Contents

- Документация по C#
- Начало работы
  - Вступление
  - Типы
  - Стандартные блоки программы
  - Основные области языка
  - Учебники
    - Выберите первый урок
    - Учебники на основе браузера
      - Здравствуй, мир
      - Числа в C#
      - Ветви и циклы
      - Коллекции списков
    - Работа в локальной среде
      - Настройка среды
      - Числа в C#
      - Ветви и циклы
      - Коллекции списков
- Базовый
  - Структура программы
    - Обзор
    - Main - метод
    - Инструкции верхнего уровня
  - Система типов
    - Обзор
    - Пространства имен
    - Классы
    - Записи
    - Интерфейсы

- Универсальные шаблоны
- Анонимные типы
- Объектно-ориентированное программирование
  - Обзор
  - Объекты
  - Наследование
  - Полиморфизм
- Функциональные методики
  - Регулярные выражения
  - Пустые переменные
  - Деконструкция кортежей и других типов
- Исключения и ошибки
  - Обзор
  - Использование исключений
  - Обработка исключений
  - Создание и генерация исключений
  - Исключения, создаваемые компилятором
- Стиль кода
  - Имена идентификаторов
  - Соглашения о написании кода на C#
- Учебники
  - Практическое руководство. Отображение аргументов командной строки
  - Общие сведения о классах
  - Объектно-ориентированный язык C#
  - Наследование в C# и .NET
  - Преобразование типов
  - Создание алгоритмов на основе данных с сопоставлением шаблонов
  - Практическое руководство. Обработка исключений с помощью блока try-catch
  - Практическое руководство. Выполнение кода очистки с использованием блока finally
- Новые возможности C#
  - C# 10
  - C# 9.0

- C# 8.0
- Критические изменения компилятора
- Журнал версий C#
- Связи с библиотекой .NET
- Совместимость версий
- Учебники
  - Подробнее о типах записей
  - Анализ с инструкциями верхнего уровня
  - Поиск закономерностей в объектах
  - Безопасное обновление интерфейсов с помощью методов интерфейса по умолчанию
  - Создание функциональных возможностей смешения с помощью методов интерфейса по умолчанию
  - Индексы и диапазоны
  - Использование ссылочных типов, допускающих значение NULL
  - Создание и использование асинхронных потоков
  - Написание пользовательского обработчика интерполяции строк
- Учебники
  - Интерполяция строк. Интерактивное руководство
  - Интерполяция строк. В вашей среде
  - Расширенные сценарии для интерполяции строки
  - Консольное приложение
  - Клиент REST
  - Работа с LINQ
  - Использование атрибутов
- Основные понятия C#
  - Ссылочные типы, допускающие значение null
  - Миграции пустых ссылок
  - Разрешение предупреждений, допускающих значения NULL
  - Методы
  - Свойства
  - Индексаторы
  - Iterators

Делегаты и события
Общие сведения о делегатах
System.Delegate и ключевое слово delegate
Строго типизированные делегаты
Общие шаблоны делегатов
Общие сведения о событиях
Стандартные шаблоны событий .NET
Обновленный шаблон событий .NET
Различия между делегатами и событиями
LINQ
Обзор LINQ
Основы выражения запроса
LINQ в C#
Создание запросов LINQ на языке C#
Запрос коллекции объектов
Возврат запроса из метода
Сохранение результатов запроса в памяти
Группировка результатов запросов
Создание вложенной группы
Вложенный запрос в операции группирования
Группирование результатов по смежным ключам
Динамическое определение фильтров предикатов во время выполнения
Выполнение внутренних соединений
Выполнение групповых соединений
Выполнение левых внешних соединений
Упорядочение результатов предложения соединения
Соединение с помощью составных ключей
Выполнение пользовательских операций соединения
Обработка значений NULL в выражениях запросов
Обработка исключений в выражениях запросов
Написание безопасного и эффективного кода
Деревья выражений
Введение в деревья выражений

- Описание деревьев выражений
- Типы платформ, поддерживающие деревья выражений
- Выполнение выражений
- Интерпретация выражений
- Построение выражений
- Преобразование выражений
- Сводка
- Взаимодействие на уровне машинного кода
- Управление версиями

Статьи с практическими инструкциями по C#

- Индекс статьи
- Разделение строк на подстроки
- Объединение строк
- Поиск по строкам
- Изменение содержимого строк
- Сравнение строк
- Практическое руководство. Перехват несовместимого с CLS исключения

Пакет SDK для платформы компилятора .NET (API-интерфейсы Roslyn)

- Обзор пакета SDK для .NET Compiler Platform (API Roslyn)
- Сведения о модели SDK для .NET Compiler Platform
- Работа с синтаксисом
- Работа с семантикой
- Использование рабочей области
- Изучение кода с помощью визуализатора синтаксиса
- Генераторы источников
- Краткие руководства
  - Анализ синтаксиса
  - Семантический анализ
  - Синтаксическое преобразование
- Учебники
  - Создание средства для анализа и исправления кода

Руководство по программированию на C#

Обзор
Основные понятия программирования
Обзор
Асинхронное программирование
Обзор
Сценарии асинхронного программирования
Асинхронная модель программирования
Асинхронные типы возвращаемых значений
Отмена задач
Отмена списка задач
Отмена задач после определенного периода времени
Обработка асинхронных задач по мере завершения
Асинхронный доступ к файлам
Атрибуты
Обзор
Создание настраиваемых атрибутов
Обращение к атрибутам с помощью отражения
Практическое руководство. Создание объединения C/C++ с помощью атрибутов
Коллекции
Ковариантность и контрвариантность
Обзор
Вариативность в универсальных интерфейсах
Создание вариативных универсальных интерфейсов
Использование вариативности в интерфейсах для универсальных коллекций
Вариативность в делегатах
Использование вариативности в делегатах
Использование вариативности в универсальных методах-делегатах Func и Action
Деревья выражений
Обзор
Выполнение деревьев выражений

- Изменение деревьев выражений
- Использование деревьев выражений для создания динамических запросов
- Отладка деревьев выражений в Visual Studio
- Синтаксис DebugView

Iterators

LINQ

- Обзор
- Приступая к работе с LINQ в C#
  - Введение в запросы LINQ
  - LINQ и универсальные типы
  - Основные операции запроса LINQ
  - Преобразование данных с помощью LINQ
  - Связи типов в операциях запросов LINQ
  - Синтаксис запросов и синтаксис методов в LINQ
  - Возможности C#, поддерживающие LINQ
  - Пошаговое руководство. Написание запросов на C# (LINQ)
- Общие сведения о стандартных операторах запроса
  - Обзор
  - Синтаксис выражений запроса для стандартных операторов запроса
  - Классификация стандартных операторов запросов по способу выполнения
  - Сортировка данных
  - Операции над множествами
  - Фильтрация данных
  - Операции, использующие квантификаторы
  - Операции проецирования
  - Секционирование данных
  - Операции соединения
  - Группировка данных
  - Операции создания
  - Операции сравнения
  - Операции с элементами
  - Преобразование типов данных

Операции объединения
Операции агрегирования
LINQ to Objects
Обзор
LINQ и строки
Статьи с пошаговыми инструкциями
Практическое руководство. Подсчет вхождений слова в строке (LINQ)
Практическое руководство. Запрос к предложениям, содержащим указанный набор слов (LINQ)
Практическое руководство. Запрос знаков в строке (LINQ)
Практическое руководство. Объединение запросов LINQ с регулярными выражениями
Практическое руководство. Нахождение разности наборов между двумя списками (LINQ)
Практическое руководство. Сортировка или фильтрация текстовых данных по любому слову или полю (LINQ)
Практическое руководство. Изменение порядка полей файла с разделителями (LINQ)
Практическое руководство. Объединение и сравнение коллекций строк (LINQ)
Практическое руководство. Заполнение коллекций объектов из нескольких источников (LINQ)
Практическое руководство. Разделение файла на несколько файлов с помощью групп (LINQ)
Практическое руководство. Объединение содержимого из файлов разных форматов (LINQ)
Практическое руководство. Вычисление значений столбцов в текстовом CSV-файле (LINQ)
LINQ и отражение
Практическое руководство. Выполнение запроса к метаданным сборки при помощи отражения (LINQ)
LINQ и каталоги файлов
Обзор
Практическое руководство. Запрос файлов с указанными атрибутами или именем

Практическое руководство. Группировка файлов по расширению (LINQ)
Практическое руководство. Запрос общего числа байтов в наборе папок (LINQ)
Практическое руководство. Сравнение содержимого двух папок (LINQ)
Практическое руководство. Запрос самого большого файла или файлов в дереве папок (LINQ)
Практическое руководство. Запрос повторяющихся файлов в дереве каталогов (LINQ)
Практическое руководство. Запрос содержимого файлов в папке (LINQ)
Практическое руководство. Выполнение запроса к ArrayList с помощью LINQ
Практическое руководство. Добавление настраиваемых методов для запросов LINQ
LINQ to ADO.NET (Страница портала)
Включение источника данных для запросов LINQ
Среда разработки Visual Studio и поддержка средств для LINQ
Отражение
Сериализация (C#)
Обзор
Практическое руководство. Запись данных объекта в XML-файл
Практическое руководство. Чтение данных объекта из XML-файла
Пошаговое руководство. Сохранение объекта в Visual Studio
Инструкции, выражения и операторы
Обзор
Операторы
Элементы, воплощающие выражение
Равенства и сравнения на равенство
Сравнения на равенство
Практическое руководство. Определение равенства значений для типа
Практическое руководство. Проверка на ссылочное равенство (идентичность)
Типы
Приведение и преобразование типов
Упаковка–преобразование и распаковка–преобразование

Практическое руководство. Преобразование массива байтов в значение типа int
Практическое руководство. Преобразование строки в число
Практическое руководство. Преобразование из шестнадцатеричных строк в числовые типы
Использование типа dynamic
Пошаговое руководство. Создание и использование динамических объектов (C# и Visual Basic)
Классы, структуры и записи
Полиморфизм
Управление версиями с помощью ключевых слов Override и New
Использование ключевых слов Override и New
Переопределение метода ToString
Участники
Обзор участников
Абстрактные и запечатанные классы и члены классов
Статические классы и члены статических классов
Модификаторы доступа
Поля
Константы
Определение абстрактных свойств
Определение констант в C#
Свойства
Общие сведения о свойствах
Использование свойств
Свойства интерфейса
Ограничение доступности методов доступа
Объявление и использование свойств чтения и записи
Автоматически реализуемые свойства
Практическое руководство. Реализация облегченного класса с автоматически реализуемыми свойствами
Методы
Обзор методов

Локальные функции
Возвращаемые ссылочные значения и ссылочные локальные переменные
Параметры
Передача параметров
Передача параметров типа значения
Передача параметров ссылочного типа
Определение различия между передачей структуры и ссылки класса в метод
Неявно типизированные локальные переменные
Использование неявно типизированных локальных переменных и массивов в выражении запроса
Методы расширения
Реализация и вызов пользовательского метода расширения
Создание нового метода для перечисления
Именованные и необязательные аргументы
Использование именованных и необязательных аргументов в программировании приложений Office
Конструкторы
Обзор конструкторов
Использование конструкторов
Конструкторы экземпляров
Закрытые конструкторы
Статические конструкторы
Создание конструктора копий
Методы завершения
Инициализаторы объектов и коллекций
Практическое руководство. Инициализация объектов с помощью инициализатора объектов
Практическое руководство. Инициализация словаря с помощью инициализатора коллекции
Вложенные типы
Разделяемые классы и методы
Возвращение подмножеств свойств элементов в запросе
Интерфейсы

Явная реализация интерфейса
Практическое руководство. Явная реализация членов интерфейса
Практическое руководство. Явная реализация членов двух интерфейсов
Делегаты
Обзор
Использование делегатов
Делегаты с именованными методами и Анонимные методы
Практическое руководство. Объединение делегатов (многоадресные делегаты) (руководство по программированию на C#)
Практическое руководство. Объявление, создание и использование делегата
Массивы
Обзор
Одномерные массивы
Многомерные массивы
Массивы массивов
Использование оператора foreach с массивами
Передача массивов в качестве аргументов
Неявно типизированные массивы
Строки
Программирование со строками
Практическое руководство. Определение представления числового значения в строке
Индексаторы
Обзор
Использование индексаторов
Индексаторы в интерфейсах
Сравнение свойств и индексаторов
События
Обзор
Практическое руководство. Подписка и отмена подписки на события
Практическое руководство. Публикация событий, соответствующих рекомендациям .NET
Практическое руководство. Создание событий базового класса в производных

классах
Практическое руководство. Реализация событий интерфейса
Практическое руководство. Реализация пользовательских методов доступа к событиям
Универсальные шаблоны
Параметры универсального типа
Ограничения параметров типа
Универсальные классы
Универсальные интерфейсы
Универсальные методы
Универсальные методы и массивы
Универсальные делегаты
Различия между шаблонами языка C++ и универсальными шаблонами языка C#
Универсальные типы во время выполнения
Универсальные типы и отражение
Универсальные шаблоны и атрибуты
Файловая система и реестр
Обзор
Практическое руководство. Перебор дерева папок
Практическое руководство. Получение сведений о файлах, папках и дисках
Практическое руководство. Создание файла или папки
Практическое руководство. Копирование, удаление и перемещение файлов и папок
Практическое руководство. Предоставление диалогового окна "Ход выполнения" для операций с файлами
Практическое руководство. Запись в текстовый файл
Практическое руководство. Чтение из текстового файла
Практическое руководство. Построчное чтение текстового файла
Практическое руководство. Создание раздела в реестре
Взаимодействие
Взаимодействие с платформой .NET
Общие сведения о взаимодействии

Практическое руководство. Доступ к объектам взаимодействия Office с помощью функций языка C#
Практическое руководство. Использование индексированных свойств в программировании COM-взаимодействия
Практическое руководство. Использование вызова неуправляемого кода для воспроизведения WAV-файла
Пошаговое руководство. Программирование для Office (C# и Visual Basic)
Пример COM-класса
Справочник по языку
Обзор
Настройка версии языка
Типы
Типы значений
Обзор
Целочисленные типы
Собственные целочисленные типы nint и nuint
Числовые типы с плавающей запятой
Встроенные числовые преобразования
bool
char
Типы перечисления.
Типы структур
Типы кортежей
Типы значений, допускающие значение NULL
Ссылочные типы
Характеристики ссылочных типов
Встроенные ссылочные типы
запись
class
interface
Ссылочные типы, допускающие значение null
void
var

- Встроенные типы
- Неуправляемые типы
- Значения по умолчанию

Keywords

- Обзор
- Модификаторы
  - Модификаторы доступа
    - Краткий справочник
    - Уровни доступности
    - Домен доступности
    - Ограничения на использование уровней доступности
    - internal
    - private
    - protected
    - public
    - protected internal
    - private protected
  - abstract
  - async
  - const
  - event
  - extern
  - in (универсальный модификатор)
  - new (модификатор члена)
  - out (универсальный модификатор)
  - override
  - readonly
  - sealed
  - static
  - unsafe
  - virtual
  - volatile

Ключевые слова операторов
Категории операторов
Операторы обработки исключений
throw
try-catch
try-finally
try-catch-finally
Checked и Unchecked
Обзор
checked
unchecked
Оператор fixed
Оператор lock
Параметры методов
Передача параметров
params
in (модификатор параметров)
ref
out (модификатор параметров)
Ключевые слова, используемые для пространств имен
namespace
using
Контексты для using
Директива using
Оператор using
Псевдоним extern
Ключевые слова ограничений универсального типа
Ограничение new
where
Ключевые слова доступа
base
this

Ключевые слова литералов
null
True и False
default
Контекстные ключевые слова
Краткий справочник
add
get
init
partial (тип)
partial (метод)
remove
set
when (условие фильтра)
значение
yield
Ключевые слова запроса
Краткий справочник
Предложение from
Предложение where
Предложение select
Предложение group
into
Предложение orderby
Предложение join
Предложение let
ascending
descending
on
equals
by
in

Операторы и выражения
Обзор
Арифметические операторы
Логические операторы
Операторы битовых операций и сдвига
Операторы равенства
Операторы сравнения
Операторы и выражения для доступа к членам
Операторы проверки типа и выражения CAST
Операторы пользовательского преобразования
Операторы, связанные с указателем
Операторы присваивания
Лямбда-выражения
Шаблоны
+ Операторы +=
- Операторы -=
Оператор ?:
! Оператор (null-forgiving)
?? и ??= — операторы
Оператор =>
:: - оператор
оператор await
Выражения значения по умолчанию
Оператор delegate
Оператор is
Выражение nameof
Оператор new
Оператор sizeof
Выражение stackalloc
Выражение switch
Операторы true и false
Выражение with

- Перегрузка операторов
- Инструкции
  - Операторы итерации
  - Операторы выбора
  - Операторы перехода
- Специальные символы
  - Обзор
  - $ — интерполяция строк
  - @ — буквальный идентификатор
- Считываемые компилятором атрибуты
  - Глобальные атрибуты
  - Сведения о вызывающем объекте
  - Статический анализ типов, допускающих значения NULL
  - Разное
- Небезопасный код и указатели
- Директивы препроцессора
- Параметры компилятора
  - Обзор
  - Параметры языка
  - Параметры вывода
  - Параметры ввода
  - Параметры ошибок и предупреждений
  - Параметры создания кода
  - Параметры безопасности
  - Параметры ресурсов
  - Прочие параметры
  - Расширенные параметры
- Комментарии XML-документации
  - Создание документации по API
  - Рекомендуемые теги
  - Примеры
- Сообщения компилятора

Спецификации
Предварительная спецификация C# 6.0
Вступление
Лексическая структура
Основные понятия
Типы
Переменные
Преобразования
Выражения
Операторы
Пространства имен
Классы
Структуры
Массивы
Интерфейсы
Перечисления
Делегаты
Исключения
Атрибуты
Небезопасный код
Комментарии для документации
Функции C# 7.0–10.0
Функции C# 7.0
Регулярные выражения
Локальные функции
Объявление выходных переменных
Выражения throw
Двоичные литералы
Разделители между цифрами
Асинхронные типы задач
Функции C# 7.1
Асинхронный метод main

Стандартные выражения
Выводимые имена кортежей
Сопоставление шаблонов с универсальными шаблонами
Функции C# 7.2
Ссылки только для чтения
Безопасность во время компиляции для типов, схожих с ссылочными
Неконечные именованные аргументы
Частный защищенный
Условная ссылочная переменная
Разделитель начальных цифр
Функции C# 7.3
Ограничения неуправляемых универсальных типов
Индексирование полей `fixed` не требует закрепления независимо от перемещаемого или неперемещаемого контекста
Инструкция `fixed` на основе шаблона
Переназначение ссылочных локальных переменных
Инициализаторы массивов stackalloc
Нацеленные на поле атрибуты автоматически реализуемых свойств
Переменные выражений в инициализаторах
Равенство (==) и неравенство (!=) кортежей
Улучшенный отбор потенциальных перегрузок
Функции C# 8.0
Ссылочные типы, допускающие значение NULL. Предложение
Рекурсивное сопоставление шаблонов
Методы интерфейса по умолчанию
Асинхронные потоки
Диапазоны
Использование шаблонов и объявлений using
Статические локальные функции
Присваивание объединения со значением NULL
Члены экземпляров с доступом только на чтение
Вложенная функция stackalloc
Функции C# 9.0

Записи
Инструкции верхнего уровня
Ссылочные типы, допускающие значение NULL. Спецификация
Улучшения сопоставления шаблонов
Методы задания только инициализации
Новые выражения целевого типа
Инициализаторы модулей
Расширение разделяемых методов
Статические анонимные функции
Условное выражение с целевым типом
Ковариантные возвращаемые типы
Расширение GetEnumerator в циклах foreach
Параметры удаления лямбда-выражения
Атрибуты локальных функций
Целые числа собственного размера
Указатели функций
Отмена создания флага localsinit
Аннотации неограниченных параметров типа
Возможности C# 10
Структуры записей
Конструкторы структур без параметров
Глобальная директива using
Пространства имен с областью действия файла
Расширенные шаблоны свойств
Улучшенные интерполированные строки
Константные интерполированные строки
Улучшения лямбда-выражений
Выражение аргумента вызывающей стороны
Расширенные директивы #line
Универсальные атрибуты
Улучшенный анализ определенного назначения
Переопределение AsyncMethodBuilder

Краткий обзор языка C#

18.12.2021 • 12 minutes to read

C# (произносится как "си шарп") — современный объектно-ориентированный и типобезопасный язык программирования. C# позволяет разработчикам создавать разные типы безопасных и надежных приложений, выполняющихся в .NET. C# относится к широко известному семейству языков C, и покажется хорошо знакомым любому, кто работал с C, C++, Java или JavaScript. Здесь представлен обзор основных компонентов языка C# 8 и более ранних версий. Если вы хотите изучить язык с помощью интерактивных примеров, рекомендуем поработать с вводными руководствами по C#.

C# — это объектно- и ***компонентно-ориентированный*** *_ язык программирования. C# предоставляет языковые конструкции для непосредственной поддержки такой концепции работы. Благодаря этому C# подходит для создания и применения программных компонентов. С момента создания язык C# обогатился функциями для поддержки новых рабочих нагрузок и современными рекомендациями по разработке ПО. C# — это _ объектно-ориентированный* язык. Вы определяете типы и их поведение.

Вот лишь несколько функций языка C#, которые позволяют создавать надежные и устойчивые приложения. ***Сборка мусора*** _ автоматически освобождает память, занятую недоступными неиспользуемыми объектами. *Типы, допускающие значение null,* обеспечивают защиту от переменных, которые не ссылаются на выделенные объекты. *Обработка исключений* предоставляет структурированный и расширяемый подход к обнаружению ошибок и восстановлению после них. *Лямбда-выражения* поддерживают приемы функционального программирования. *Синтаксис LINQ* создает общий шаблон для работы с данными из любого источника. Поддержка языков для *асинхронных операций* предоставляет синтаксис для создания распределенных систем. В C# действует _ ***единая система типов****. Все типы C#, включая типы-примитивы, такие как `int` и `double`, наследуют от одного корневого типа `object`. Все типы используют общий набор операций, а значения любого типа можно хранить, передавать и обрабатывать схожим образом. Более того, C# поддерживает как определяемые пользователями ссылочные типы, так и типы значений. C# позволяет динамически выделять объекты и хранить упрощенные структуры в стеке. C# поддерживает универсальные методы и типы, обеспечивающие повышенную безопасность типов и производительность. C# предоставляет итераторы, которые позволяют разработчикам классов коллекций определять пользовательские варианты поведения для клиентского кода.

В C# особое внимание уделяется ***управлению версиями*** для обеспечения совместимости программ и библиотек при их изменении. Вопросы управления версиями существенно повлияли на такие аспекты разработки C#, как раздельные модификаторы `virtual` и `override`, правила разрешения перегрузки методов и поддержка явного объявления членов интерфейса.

Архитектура .NET

Программы C# выполняются в .NET, виртуальной системе выполнения, вызывающей общеязыковую среду выполнения (CLR) и набор библиотек классов. Среда CLR — это реализация общеязыковой инфраструктуры языка (CLI), являющейся международным стандартом, от корпорации Майкрософт. CLI является основой для создания сред выполнения и разработки, в которых языки и библиотеки прозрачно работают друг с другом.

Исходный код, написанный на языке C# компилируется в промежуточный язык (IL), который соответствует спецификациям CLI. Код на языке IL и ресурсы, в том числе растровые изображения и строки, сохраняются в сборке, обычно с расширением *.dll*. Сборка содержит манифест с информацией о типах, версии, языке и региональных параметрах для этой сборки.

При выполнении программы C# сборка загружается в среду CLR. Среда CLR выполняет JIT-компиляцию из кода на языке IL в инструкции машинного языка. Среда CLR также выполняет другие операции, например, автоматическую сборку мусора, обработку исключений и управление ресурсами. Код, выполняемый в среде CLR, иногда называется управляемым кодом. "Неуправляемый код" преобразуется в машинный язык, предназначенный для конкретной платформы.

Обеспечение взаимодействия между языками является ключевой особенностью .NET. Код IL, созданный компилятором C#, соответствует спецификации общих типов (CTS). Код IL, созданный из кода на C#, может взаимодействовать с кодом, созданным из версий .NET для языков F#, Visual Basic, C++. Существует более 20 других языков, совместимых с CTS. Одна сборка может содержать несколько модулей, написанных на разных языках .NET, и все типы могут ссылаться друг на друга, как если бы они были написаны на одном языке.

В дополнение к службам времени выполнения .NET также включает расширенные библиотеки. Эти библиотеки поддерживают множество различных рабочих нагрузок. Они упорядочены по пространствам имен, которые предоставляют разные полезные возможности: от операций файлового ввода и вывода до управления строками и синтаксического анализа XML, от платформ веб-приложений до элементов управления Windows Forms. Обычно приложение C# активно используют библиотеку классов .NET для решения типовых задач.

Дополнительные сведения о .NET, см. в статье Обзор .NET.

Здравствуй, мир

Для первого знакомства с языком программирования традиционно используется программа "Hello, World". Вот ее пример на C#:

```
using System;

class Hello
{
    static void Main()
    {
        Console.WriteLine("Hello, World");
    }
}
```

Программа "Hello, World" начинается с директивы `using`, которая ссылается на пространство имен `System`. Пространства имен позволяют иерархически упорядочивать программы и библиотеки C#. Пространства имен содержат типы и другие пространства имен. Например, пространство имен `System` содержит несколько типов (в том числе используемый в нашей программе класс `Console`) и несколько других пространств имен, таких как `IO` и `Collections`. Директива `using`, которая ссылается на пространство имен, позволяет использовать типы из этого пространства имен без указания полного имени. Благодаря директиве `using` в коде программы можно использовать сокращенное имя `Console.WriteLine` вместо полного варианта `System.Console.WriteLine`.

Класс `Hello`, объявленный в программе "Hello, World", имеет только один член — это метод с именем `Main`. Метод `Main` объявлен с модификатором `static`. Методы экземпляра могут ссылаться на конкретный экземпляр объекта, используя ключевое слово `this`, а статические методы работают без ссылки на конкретный объект. По стандартному соглашению точкой входа программы C# является статический метод с именем `Main`.

Выходные данные программы создаются в методе `WriteLine` класса `Console` из пространства имен `System`. Этот класс предоставляется библиотеками стандартных классов, ссылки на которые компилятор по умолчанию добавляет автоматически.

Типы и переменные

Тип определяет структуру и поведение любых данных в C#. Объявление типа может включать его члены, базовый тип, интерфейсы, которые он реализует, и операции, разрешенные для этого типа. *Переменная* — это метка, которая ссылается на экземпляр определенного типа.

В C# существуют две разновидности типов: *ссылочные типы* и *типы значений*. Переменные типа значений содержат непосредственно данные, а в переменных ссылочных типов хранятся ссылки на нужные данные, которые именуются объектами. Две переменные ссылочного типа могут ссылаться на один и тот же объект, поэтому может случиться так, что операции над одной переменной затронут объект, на который ссылается другая переменная. Каждая переменная типа значения имеет собственную копию данных, и операции над одной переменной не могут затрагивать другую (за исключением переменных параметров `ref` и `out`).

Идентификатор является именем переменной. Идентификатор — это последовательность символов Юникода без пробелов. Идентификатор может быть зарезервированным словом C#, если он имеет префикс `@`. При взаимодействии с другими языками в качестве идентификатора может быть полезно использовать зарезервированное слово.

Типы значений в C# делятся на *простые типы*, *типы перечислений*, *типы структур*, *типы, допускающие значение NULL*, и *типы значений кортежей*. Ссылочные типы в C# подразделяются на *типы классов*, *типы интерфейсов*, *типы массивов* и *типы делегатов*.

Далее представлены общие сведения о системе типов в C#.

- Типы значений
  - Простые типы
    - Целочисленный со знаком: `sbyte`, `short`, `int`, `long`.
    - Целочисленный без знака: `byte`, `ushort`, `uint`, `ulong`.
    - Символы Юникода: `char`, который представляет блок кода в кодировке UTF-16.
    - Бинарный оператор IEEE с плавающей запятой: `float`, `double`.
    - Десятичное значение с повышенной точностью и плавающей запятой: `decimal`.
    - Логический: `bool`, используется для представления логических значений, которые могут иметь значение `true` или `false`.
  - Типы перечисления
    - Пользовательские типы в формате `enum E {...}`. Тип `enum` является отдельным типом со списком именованных констант. Каждый тип `enum` имеет базовый тип, в роли которого выступает один из восьми целочисленных типов. Набор значений типа `enum` аналогичен набору значений его базового типа.
  - Типы структур
    - Пользовательские типы в формате `struct S {...}`
  - Типы значений, допускающие значение NULL
    - Расширения других типов значений, допускающие значение `null`
  - Типы значений кортежей
    - Пользовательские типы в формате `(T1, T2, ...)`
- Ссылочные типы
  - Типы классов
    - Исходный базовым классом для всех типов: `object`
    - Строки в Юникоде: `string`, который представляет последовательность блоков кода в кодировке UTF-16.
    - Пользовательские типы в формате `class C {...}`

- Типы интерфейсов
  - Пользовательские типы в формате `interface I {...}`
- Типы массивов
  - Одномерные, многомерные массивы и массивы массивов. Например, `int[]`, `int[,]` и `int[][]`.
- Типы делегатов
  - Пользовательские типы в формате `delegate int D(...)`

Программы C# используют *объявления типов* для создания новых типов. В объявлении типа указываются имя и члены нового типа. Шесть категорий типов в C# определяются пользователем: типы классов, типы структур, типы интерфейсов, типы перечисления, типы делегатов и типы значений кортежей. Можно также объявлять типы `record`, либо `record struct`, либо `record class`. Типы записей имеют члены, синтезированные компилятором. Записи используются в основном для хранения значений с минимальным связанным поведением.

- Тип `class` определяет структуру данных, которая содержит данные-члены (поля) и функции-члены (методы, свойства и т. д.). Классы поддерживают механизмы одиночного наследования и полиморфизма, которые позволяют создавать производные классы, расширяющие и уточняющие определения базовых классов.
- Тип `struct` похож на тип класса тем, что он представляет структуру с данными-членами и функциями-членами. Но, в отличие от классов, структуры являются типами значений и обычно не требуют выделения памяти из кучи. Типы структуры не поддерживают определяемое пользователем наследование, и все типы структуры неявно наследуют от типа `object`.
- Тип `interface` определяет контракт в виде именованного набора открытых элементов. Объект типа `class` или `struct`, реализующий `interface`, должен предоставить реализации для всех элементов интерфейса. Тип `interface` может наследовать от нескольких базовых интерфейсов, а `class` или `struct` могут реализовывать несколько интерфейсов.
- Тип `delegate` (делегат) представляющий ссылки на методы с конкретным списком параметров и типом возвращаемого значения. Делегаты позволяют использовать методы как сущности, сохраняя их в переменные и передавая в качестве параметров. Делегаты аналогичны типам функций, которые используются в функциональных языках. Их принцип работы близок к указателям функций из некоторых языков. В отличие от указателей функций, делегаты являются объектно-ориентированными и типобезопасными.

Типы `class`, `struct`, `interface` и `delegate` поддерживают универсальные шаблоны, которые позволяют передавать им другие типы в качестве параметров.

C# поддерживает одномерные и многомерные массивы любого типа. В отличие от перечисленных выше типов, типы массивов не требуется объявлять перед использованием. Типы массивов можно сформировать, просто введя квадратные скобки после имени типа. Например, `int[]` является одномерным массивом значений типа `int`, а `int[,]` — двумерным массивом значений типа `int`, тогда как `int[][]` представляет собой одномерный массив одномерных массивов (или массив массивов) значений типа `int`.

Типы, допускающие значение NULL, не требуют отдельного определения. Для каждого обычного типа `T`, который не допускает значение NULL, существует идентичный тип `T?`, который отличается только тем, что может содержать дополнительное значение `null`. Например, `int?` является типом, который может содержать любое 32-разрядное целое число или значение `null`, а `string?` — любое значение `string` или `null`.

Система типов в C# унифицирована таким образом, что значение любого типа можно рассматривать как `object` (объект). Каждый тип в C# является прямо или косвенно производным от типа класса `object`, и

этот тип `object` является исходным базовым классом для всех типов. Чтобы значения ссылочного типа обрабатывались как объекты, им просто присваивается тип `object`. Чтобы значения типов значений обрабатывались как объекты, выполняются операции *упаковки-преобразования* и *распаковки-преобразования*. В следующем примере значение `int` преобразуется в `object`, а затем обратно в `int`.

```
int i = 123;
object o = i;    // Boxing
int j = (int)o;  // Unboxing
```

Если значение типа назначается ссылке `object`, для хранения значения выделяется упаковка. Эта упаковка является экземпляром ссылочного типа, и в нее копируется значение. И наоборот, если ссылка типа `object` используется для типа значения, для соответствующего `object` выполняется проверка, является ли он упаковкой правильного типа. Если эта проверка завершается успешно, копируется значение этой упаковки.

Унифицированная система типов C# фактически позволяет преобразовывать типы значений в ссылки `object` "по требованию". Такая унификация позволяет применять универсальные библиотеки, использующие тип `object`, со всеми типами, производными от `object`, включая как ссылочные типы, так и с типы значений.

В C# существует несколько типов *переменных*, в том числе поля, элементы массива, локальные переменные и параметры. Переменные представляют собой места хранения, и каждая переменная имеет тип, который определяет допустимые значения для хранения в этой переменной. Примеры представлены ниже.

- Тип значения, не допускающий значения Null
  - Значение такого типа
- Тип значения, допускающий значение Null
  - Значение `null` или значение такого типа
- object
  - Ссылка `null`, ссылка на объект любого ссылочного типа или ссылка на упакованное значение любого типа значения
- Тип класса
  - Ссылка `null`, ссылка на экземпляр такого типа класса или ссылка на экземпляр любого класса, производного от такого типа класса
- Тип интерфейса
  - Ссылка `null`, ссылка на экземпляр типа класса, который реализует такой тип интерфейса, или ссылка на упакованное значение типа значения, которое реализует такой тип интерфейса
- Тип массива
  - Ссылка `null`, ссылка на экземпляр такого типа массива или ссылка на экземпляр любого совместимого типа массива
- Тип делегата
  - Ссылка `null` или ссылка на экземпляр совместимого типа делегата

Структура программы

В C# основными понятиями организационной структуры являются *__программы__ _, *пространства имен*, *типы*, *элементы* и *сборки*. В программе объявляются типы, которые содержат члены. Эти типы можно организовать в пространства имен. Примерами типов являются классы, структуры и интерфейсы. К членам относятся поля, методы, свойства и события. При компиляции программы на C# упаковываются в сборки. Сборка — это файл, обычно с расширением `.exe` или `.dll`, если она реализует *приложение* или

**библиотеку**, соответственно.

В качестве небольшого примера рассмотрим сборку, содержащую следующий код:

```
namespace Acme.Collections;

public class Stack<T>
{
    Entry _top;

    public void Push(T data)
    {
        _top = new Entry(_top, data);
    }

    public T Pop()
    {
        if (_top == null)
        {
            throw new InvalidOperationException();
        }
        T result = _top.Data;
        _top = _top.Next;

        return result;
    }

    class Entry
    {
        public Entry Next { get; set; }
        public T Data { get; set; }

        public Entry(Entry next, T data)
        {
            Next = next;
            Data = data;
        }
    }
}
```

Полное имя этого класса: `Acme.Collections.Stack`. Этот класс содержит несколько членов: поле с именем `_top`, два метода с именами `Push` и `Pop`, а также вложенный класс с именем `Entry`. Класс `Entry`, в свою очередь, содержит три члена: свойство с именем `Next`, свойство с именем `Data` и конструктор. `Stack` — это *универсальный* класс. Он имеет параметр одного типа `T`, который замещается конкретным типом при использовании.

Стек — это коллекция типа FILO (прибыл первым — обслужен последним). Новые элементы добавляются в верх стека. Удаляемый элемент исключается из верхней части стека. В предыдущем примере объявляется тип `Stack`, который определяет хранилище и поведение для стека. Можно объявить переменную, которая ссылается на экземпляр типа `Stack` для использования этой возможности.

Сборки содержат исполняемый код в виде инструкций промежуточного языка (IL) и символьную информацию в виде метаданных. Перед выполнением JIT-компилятор среды CLR .NET преобразует код IL в сборке в код, зависящий от процессора.

Сборка полностью описывает сама себя и содержит весь код и метаданные, поэтому в C# не используются директивы `#include` и файлы заголовков. Чтобы использовать в программе C# открытые типы и члены, содержащиеся в определенной сборке, вам достаточно указать ссылку на эту сборку при компиляции программы. Например, эта программа использует класс `Acme.Collections.Stack` из сборки `acme.dll`:

```
class Example
{
    public static void Main()
    {
        var s = new Acme.Collections.Stack<int>();
        s.Push(1); // stack contains 1
        s.Push(10); // stack contains 1, 10
        s.Push(100); // stack contains 1, 10, 100
        Console.WriteLine(s.Pop()); // stack contains 1, 10
        Console.WriteLine(s.Pop()); // stack contains 1
        Console.WriteLine(s.Pop()); // stack is empty
    }
}
```

Для компиляции программы вам потребуется создать *ссылку* на сборку, содержащую класс стека, определенный в примере выше.

Программы C# можно хранить в нескольких исходных файлах. При компиляции программы C# все исходные файлы обрабатываются вместе, при этом они могут свободно ссылаться друг на друга. По сути, это аналогично тому, как если бы все исходные файлы были объединены в один большой файл перед обработкой. В C# никогда не используются опережающие объявления, так как порядок объявления, за редким исключением, не играет никакой роли. В C# нет требований объявлять только один открытый тип в одном исходном файле, а также имя исходного файла не обязано совпадать с типом, объявляемом в этом файле.

Такие организационные блоки описываются в других статьях этого обзора.

ВПЕРЕД

Типы и элементы

18.12.2021 • 6 minutes to read

C# является объектно-ориентированным языком, а значит поддерживает инкапсуляцию, наследование и полиморфизм. Класс наследуется непосредственно из одного родительского класса. Он может реализовывать любое число интерфейсов. Методы, которые переопределяют виртуальные методы родительского класса, должны содержать ключевое слово `override`, чтобы исключить случайное переопределение. В языке C# структура похожа на облегченный класс: это тип, распределяемый в стеке и реализующий интерфейсы, но не поддерживающий наследование. C# предоставляет `record class` и типы `record struct`. Это типы, основное назначение которых заключается в хранении данных.

Классы и объекты

Классы являются основным типом в языке C#. Класс представляет собой структуру данных, которая объединяет в себе значения (поля) и действия (методы и другие функции-члены). Класс предоставляет определение для *экземпляров* класса, которые также именуются *объектами*. Классы поддерживают механизмы *наследования* и *полиморфизма*, которые позволяют создавать *производные классы*, расширяющие и уточняющие определения *базовых классов*.

Новые классы создаются с помощью объявлений классов. Объявление класса начинается с заголовка. Заголовок указывает:

- Атрибуты и модификаторы класса
- Имя класса
- Базовый класс (при наследовании от базового класса)
- Интерфейсы, реализуемые классом.

За заголовком между разделителями `{` и `}` следует тело класса, в котором последовательно объявляются все члены класса.

Следующий код является простым примером объявления класса с именем `Point`:

```
public class Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y) => (X, Y) = (x, y);
}
```

Экземпляры классов создаются с помощью оператора `new`, который выделяет память для нового экземпляра, вызывает конструктор для инициализации этого экземпляра и возвращает ссылку на экземпляр. Следующие инструкции создают два объекта `Point` и сохраняют ссылки на эти объекты в двух переменных:

```
var p1 = new Point(0, 0);
var p2 = new Point(10, 20);
```

Занимаемая объектом память автоматически освобождается, когда объект становится недоступен. В C# нет ни необходимости, ни возможности освобождать память объектов явным образом.

Параметры типа

Универсальные классы определяют ***параметры типа***. Параметры типа — это список имен параметров типа, заключенных в угловые скобки. Параметры типа следуют за именем класса. Параметры типа можно использовать в теле класса в определениях, описывающих члены класса. В следующем примере для класса `Pair` заданы параметры типа `TFirst` и `TSecond`:

```
public class Pair<TFirst, TSecond>
{
    public TFirst First { get; }
    public TSecond Second { get; }

    public Pair(TFirst first, TSecond second) =>
        (First, Second) = (first, second);
}
```

Тип класса, для которого объявлены параметры типа, называется *универсальным типом класса*. Типы структуры, интерфейса и делегата также могут быть универсальными. Если вы используете универсальный класс, необходимо указать аргумент типа для каждого параметра типа, вот так:

```
var pair = new Pair<int, string>(1, "two");
int i = pair.First;     //TFirst int
string s = pair.Second; //TSecond string
```

Универсальный тип, для которого указаны аргументы типа, как `Pair<int,string>` в примере выше, называется *сконструированным типом*.

базовых классов;

В объявлении класса может быть указан базовый класс. Имя базового класса указывается после имени класса и параметров типа и отделяется от них двоеточием. Если спецификация базового класса не указана, класс наследуется от типа `object`. В следующем примере `Point3D` имеет базовый класс `Point`. В первом примере `Point` имеет базовый класс `object`:

```
public class Point3D : Point
{
    public int Z { get; set; }

    public Point3D(int x, int y, int z) : base(x, y)
    {
        Z = z;
    }
}
```

Класс наследует члены базового класса. Наследование означает, что класс неявно содержит почти все элементы базового класса. Класс не наследует экземпляры, статические конструкторы и метод завершения. Производный класс может добавлять новые члены к наследуемым, но не может удалить определение для наследуемого члена. В предыдущем примере `Point3D` наследует члены `X` и `Y` из `Point`, и каждый экземпляр `Point3D` содержит три свойства: `X`, `Y` и `Z`.

Используется неявное преобразование из типа класса к любому из типов соответствующего базового класса. Переменная типа класса может ссылаться как на экземпляр этого класса, так и на экземпляры любого производного класса. Например, если мы используем описанные выше объявления классов, то переменная типа `Point` может ссылаться на `Point` или `Point3D`:

```
Point a = new(10, 20);
Point b = new Point3D(10, 20, 30);
```

Структуры

Классы определяют типы, поддерживающие наследование и полиморфизм. Они позволяют создавать сложные расширения функциональности на основе иерархий производных классов. В отличие от них, типы ***struct*** — это простые типы, основное назначение которых — хранить значения данных. В структурах не может быть объявлен базовый тип; они неявно наследуются от System.ValueType. От типа `struct` не могут быть унаследованы другие типы `struct`. Они неявно запечатаны.

```
public struct Point
{
    public double X { get; }
    public double Y { get; }

    public Point(double x, double y) => (X, Y) = (x, y);
}
```

интерфейсов,

__Интерфейс__ _ определяет контракт, который может быть реализован классами и структурами. Вы определяете _интерфейс для объявления возможностей, которые являются общими для разных типов. Например, интерфейс System.Collections.Generic.IEnumerable<T> определяет согласованный способ прохода по всем элементам коллекции, таким как массив. Интерфейс может содержать методы, свойства, события и индексаторы. Интерфейс обычно не предоставляет реализацию элементов, которые в нем определены. Он лишь перечисляет элементы, которые должны быть определены в классах или структурах, реализующих этот интерфейс.

Интерфейсы могут применять ***множественное наследование***. В следующем примере интерфейс `IComboBox` наследует одновременно от `ITextBox` и `IListBox`.

```
interface IControl
{
    void Paint();
}

interface ITextBox : IControl
{
    void SetText(string text);
}

interface IListBox : IControl
{
    void SetItems(string[] items);
}

interface IComboBox : ITextBox, IListBox { }
```

Классы и структуры могут реализовывать несколько интерфейсов. В следующем примере класс `EditBox` реализует одновременно `IControl` и `IDataBound`.

```
interface IDataBound
{
    void Bind(Binder b);
}

public class EditBox : IControl, IDataBound
{
    public void Paint() { }
    public void Bind(Binder b) { }
}
```

Если класс или структура реализует конкретный интерфейс, любой экземпляр этого класса или структуры можно неявно преобразовать в такой тип интерфейса. Например.

```
EditBox editBox = new();
IControl control = editBox;
IDataBound dataBound = editBox;
```

Перечисления

Тип *Enum* определяет набор константных значений. В следующем объявлении `enum` определяются константы, соответствующие различным корнеплодам:

```
public enum SomeRootVegetable
{
    HorseRadish,
    Radish,
    Turnip
}
```

Также можно определить `enum`, которое будет использоваться в виде сочетания флагов. В следующем объявлении определяется набор флагов для четырех времен года. Можно применить любое сочетание времен года, в том числе значение `All`, включающее все времена года:

```
[Flags]
public enum Seasons
{
    None = 0,
    Summer = 1,
    Autumn = 2,
    Winter = 4,
    Spring = 8,
    All = Summer | Autumn | Winter | Spring
}
```

В следующем примере показаны объявления для обоих указанных выше перечислений:

```
var turnip = SomeRootVegetable.Turnip;

var spring = Seasons.Spring;
var startingOnEquinox = Seasons.Spring | Seasons.Autumn;
var theYear = Seasons.All;
```

Типы, допускающие значение NULL

Переменные любого типа могут быть объявлены как ***не допускающие значение*** *NULL _ или*

допускающие значение NULL. Переменная, допускающая значение NULL, может содержать дополнительное значение `null`*, указывающее на отсутствие значения. Типы значений, допускающие значение NULL (структуры или перечисления), представляются System.Nullable<T>. Ссылочные типы, не допускающие значение NULL и допускающие значение NULL, представляются базовым ссылочным типом. Различие представляется метаданными, которые считываются компилятором и некоторыми библиотеками. Если ссылки, допускающие значение NULL, разыменовываются без предварительной проверки значения на* `null`*, компилятор выдает предупреждения. Компилятор также выдает предупреждения, если ссылкам, не допускающим значение NULL, присваивается значение, которое может быть равно* `null`*. В следующем примере объявляется _переменная целочисленного типа, допускающая значение NULL_, и эта переменная инициализируется значением* `null`*. Затем ей присваивается значение* `5`*. Затем те же действия выполняются для _строки, допускающей значение NULL**.* Дополнительные сведения см. в статьях Типы значений, допускающие значение NULL и Ссылочные типы, допускающие значение NULL.

```
int? optionalInt = default;
optionalInt = 5;
string? optionalText = default;
optionalText = "Hello World.";
```

Кортежи

C# поддерживает ***кортежи***, которые предоставляют краткий синтаксис для группирования нескольких элементов данных в упрощенную структуру данных. Чтобы создать экземпляр кортежа, необходимо объявить типы и имена элементов между `(` и `)`, как показано в следующем примере:

```
(double Sum, int Count) t2 = (4.5, 3);
Console.WriteLine($"Sum of {t2.Count} elements is {t2.Sum}.");
//Output:
//Sum of 3 elements is 4.5.
```

Кортежи являются альтернативой для структуры данных с несколькими элементами. В них не используются стандартные блоки, описанные в следующей статье.

Стандартные блоки программы

18.12.2021 • 21 minutes to read

Типы, описанные в предыдущей статье, создаются с использованием следующих стандартных блоков: *__элементы__ _, *выражения* и _ **инструкции***.

Элементы

Элементы `class` могут быть ***статическими элементами*** _ *или _элементами экземпляра*_*. Статические члены принадлежат классу в целом, а члены экземпляра принадлежат конкретным объектам (экземплярам классов).

Ниже перечислены виды членов, которые могут содержаться в классе.

- **Константы**. Константные значения, связанные с классом.
- **Поля**. Переменные, связанные с классом.
- **Методы**. Действия, которые может выполнять класс.
- **Свойства**. Действия, связанные с чтением и записью именованных свойств класса.
- **Индексаторы**. Действия, реализующие индексирование экземпляров класса, чтобы обращаться к ним как к массиву.
- **События**. Уведомления, которые могут быть созданы этим классом.
- **Операторы**. Поддерживаемые классом операторы преобразования и выражения.
- **Конструкторы**. Действия, необходимые для инициализации экземпляров класса или класса в целом.
- **Методы завершения**: действия, выполняемые перед окончательным удалением экземпляров класса.
- **Типы**. Вложенные типы, объявленные в классе.

Возможности доступа

Каждый член класса имеет определенный уровень доступности. Он определяет, из какой области программы можно обращаться к этому члену. Существует шесть уровней доступности. Ниже приведены модификаторы доступа.

- `public`. Доступ не ограничен.
- `private`. Доступ возможен только из этого класса.
- `protected`. Доступ возможен из этого класса и из классов, производных от него.
- `internal`. Доступ ограничен текущей сборкой (`.exe` или `.dll`).
- `protected internal`. Доступ ограничен данным классом, классами, производными от данного класса, либо классами в той же сборке.
- `private protected`. Доступ ограничен данным классом либо классами, производными от данного типа в той же сборке.

Поля

Поле является переменной, связанной с определенным классом или экземпляром класса.

Поле, объявленное с модификатором static, является статическим. Статическое поле определяет строго одно место хранения. Независимо от того, сколько будет создано экземпляров этого класса, существует только одна копия статического поля.

Поле, объявленное без модификатора static, является полем экземпляра. Каждый экземпляр класса содержит отдельные копии всех полей экземпляра, определенных для этого класса.

В следующем примере каждый экземпляр класса `Color` содержит отдельную копию полей экземпляра `R`, `G` и `B`, но для каждого из статических полей `Black`, `White`, `Red`, `Green` и `Blue` существует только одна копия:

```
public class Color
{
    public static readonly Color Black = new(0, 0, 0);
    public static readonly Color White = new(255, 255, 255);
    public static readonly Color Red = new(255, 0, 0);
    public static readonly Color Green = new(0, 255, 0);
    public static readonly Color Blue = new(0, 0, 255);

    public byte R;
    public byte G;
    public byte B;

    public Color(byte r, byte g, byte b)
    {
        R = r;
        G = g;
        B = b;
    }
}
```

Как показано в предыдущем примере, можно объявить *поля только для чтения*, используя модификатор `readonly`. Присвоение значения доступному только для чтения полю может происходить только при объявлении этого поля или в конструкторе этого класса.

Методы

Метод — это член, реализующий вычисление или действие, которое может выполнять объект или класс. Доступ к *статическим методам* осуществляется через класс. Доступ к *методам экземпляра* осуществляется через экземпляр класса.

Для метода можно определить список *параметров*, которые представляют передаваемые методу значения или ссылки на переменные. Методы имеют *возвращаемый тип*, который задает тип значения, вычисляемого и возвращаемого методом. Если метод не возвращает значение, для него устанавливается тип возвращаемого значения `void`.

Как и типы, методы могут иметь набор параметров типа, для которых при вызове метода необходимо указывать аргументы типа. В отличие от типов, аргументы типа зачастую могут выводиться из аргументов вызова метода, и тогда их не обязательно задавать явным образом.

Сигнатура метода должна быть уникальной в пределах класса, в котором объявлен этот метод. Сигнатура метода включает имя метода, количество параметров типа, а также количество, модификаторы и типы параметров метода. Сигнатура метода не включает тип возвращаемого значения.

Если тело метода является одиночным выражением, метод можно определить с помощью компактного формата выражения, как показано в следующем примере:

```
public override string ToString() => "This is an object";
```

Параметры

Параметры позволяют передать в метод значения или ссылки на переменные. Фактические значения

параметрам метода присваиваются на основе *аргументов*, заданных при вызове метода. Существует четыре типа параметров: параметры значения, ссылочные параметры, параметры вывода и массивы параметров.

Параметр значения используется для передачи входных аргументов. Параметр значения сопоставляется с локальной переменной, которая получит начальное значение из значения аргумента, переданного в этом параметре. Изменения параметра значения не влияют на аргумент, переданный для этого параметра.

Параметры значения можно сделать необязательными, указав для них значения по умолчанию. Тогда соответствующие аргументы можно не указывать.

Ссылочный параметр используется для передачи аргументов по ссылке. Аргумент, передаваемый ссылочному параметру, должен являться переменной с определенным значением. При выполнении метода ссылочный параметр указывает на то же место хранения, где размещена переменная аргумента. Чтобы объявить ссылочный параметр, используйте модификатор `ref`. Следующий пример кода демонстрирует использование параметров `ref`.

```
static void Swap(ref int x, ref int y)
{
    int temp = x;
    x = y;
    y = temp;
}

public static void SwapExample()
{
    int i = 1, j = 2;
    Swap(ref i, ref j);
    Console.WriteLine($"{i} {j}");    // "2 1"
}
```

Параметр вывода используется для передачи аргументов по ссылке. Он похож на ссылочный параметр, однако не требует явно присваивать значение аргумента, предоставляемого вызывающим объектом. Чтобы объявить параметр вывода, используйте модификатор `out`. В следующем примере показано использование параметров `out` с помощью синтаксиса, появившегося в C# 7.

```
static void Divide(int x, int y, out int result, out int remainder)
{
    result = x / y;
    remainder = x % y;
}

public static void OutUsage()
{
    Divide(10, 3, out int res, out int rem);
    Console.WriteLine($"{res} {rem}"); // "3 1"
}
```

Массив параметров позволяет передавать в метод переменное число аргументов. Чтобы объявить массив параметров, используйте модификатор `params`. Массив параметров может быть только последним параметром в методе. Для него можно использовать только тип одномерного массива. В качестве примера правильного использования массива параметров можно назвать методы `Write` и `WriteLine`, реализованные в классе System.Console. Ниже представлены объявления этих методов.

```
public class Console
{
    public static void Write(string fmt, params object[] args) { }
    public static void WriteLine(string fmt, params object[] args) { }
    // ...
}
```

Внутри метода массив параметров полностью идентичен обычному параметру типа массив. Но зато при вызове метода, использующего массив параметров, ему можно передать либо один аргумент типа массив, либо любое количество аргументов типа элемент для массива параметров. В последнем случае экземпляр массива автоматически создается и инициализируется с заданными аргументами. Код из этого примера...

```
int x, y, z;
x = 3;
y = 4;
z = 5;
Console.WriteLine("x={0} y={1} z={2}", x, y, z);
```

...эквивалентен следующей конструкции:

```
int x = 3, y = 4, z = 5;

string s = "x={0} y={1} z={2}";
object[] args = new object[3];
args[0] = x;
args[1] = y;
args[2] = z;
Console.WriteLine(s, args);
```

Тело метода и локальные переменные

Тело метода содержит инструкции, которые будут выполнены при вызове метода.

В теле метода можно объявлять переменные, относящиеся к выполнению этого метода. Такие переменные называются *локальными переменными*. В объявлении локальной переменной нужно указать имя типа и имя переменной. Также можно задать ее начальное значение. Следующий пример кода объявляет локальную переменную `i` с нулевым начальным значением, и еще одну локальную переменную `j` без начального значения.

```
class Squares
{
    public static void WriteSquares()
    {
        int i = 0;
        int j;
        while (i < 10)
        {
            j = i * i;
            Console.WriteLine($"{i} x {i} = {j}");
            i++;
        }
    }
}
```

C# требует, чтобы локальной переменной было *явно присвоено значение*, прежде чем можно будет получить это значение. Например, если в предложенное выше объявление `i` не включить начальное значение, компилятор сообщит об ошибке при последующем использовании `i`, так как для `i` нет явно

присвоенного значения.

Метод может использовать инструкцию `return`, чтобы вернуть управление вызывающему объекту. Если метод возвращает `void`, инструкции `return` не могут указывать выражение. В методе, выходное значение которого имеет любой другой тип, инструкции `return` должны содержать выражение, которое вычисляет возвращаемое значение.

Статические методы и методы экземпляра

Метод, объявленный с модификатором `static`, является *статическим методом*. Статический метод не работает с конкретным экземпляром и может напрямую обращаться только к статическим членам.

Метод, объявленный с модификатором `static`, является *методом экземпляра*. Метод экземпляра работает в определенном экземпляре и может обращаться как к статическим методам, так и к методам этого экземпляра. В методе можно напрямую обратиться к экземпляру, для которого этот метод был вызван, используя дескриптор `this`. Использование ссылки на `this` в статическом методе является недопустимым.

Следующий класс `Entity` содержит статические члены и члены экземпляра.

```
class Entity
{
    static int s_nextSerialNo;
    int _serialNo;

    public Entity()
    {
        _serialNo = s_nextSerialNo++;
    }

    public int GetSerialNo()
    {
        return _serialNo;
    }

    public static int GetNextSerialNo()
    {
        return s_nextSerialNo;
    }

    public static void SetNextSerialNo(int value)
    {
        s_nextSerialNo = value;
    }
}
```

Каждый экземпляр `Entity` содержит серийный номер (и может содержать другие данные, которые здесь не показаны). Конструктор объекта `Entity` (который рассматривается как метод экземпляра) задает для нового экземпляра следующий доступный серийный номер. Поскольку конструктор является членом экземпляра, он может обращаться как к полю экземпляра `_serialNo`, так и к статическому полю `s_nextSerialNo`.

Статические методы `GetNextSerialNo` и `SetNextSerialNo` могут обращаться к статическому полю `s_nextSerialNo`, но прямое обращение из них к полю экземпляра `_serialNo` приводит к ошибке.

В приведенном ниже примере показано использование класса `Entity`.

```
Entity.SetNextSerialNo(1000);
Entity e1 = new();
Entity e2 = new();
Console.WriteLine(e1.GetSerialNo());          // Outputs "1000"
Console.WriteLine(e2.GetSerialNo());          // Outputs "1001"
Console.WriteLine(Entity.GetNextSerialNo());  // Outputs "1002"
```

Статические методы `SetNextSerialNo` и `GetNextSerialNo` вызываются для класса, а метод экземпляра `GetSerialNo` вызывается для экземпляров класса.

Виртуальные, переопределяющие и абстрактные методы

Для определения поведения иерархии типов классов используются виртуальные методы, методы переопределения и абстрактные методы. Поскольку класс может быть производным от базового класса, этому производному классу может потребоваться изменить поведение, реализованное в базовом классе. ***Виртуальный*** *_ метод — это метод, объявленный и реализованный в базовом классе, где любой производный класс может предоставить более конкретную реализацию. Метод _переопределения_ — это метод, реализованный в производном классе, который изменяет поведение реализации базового класса. _Абстрактный_ метод — это метод, объявленный в базовом классе, который _должен* быть переопределен во всех производных классах. Фактически, абстрактные методы не определяют реализацию в базовом классе.

Вызовы методов к методам экземпляров могут разрешаться в реализации базового класса или производного класса. Тип переменной определяет его *тип времени компиляции. Тип времени компиляции* — это тип, используемый компилятором для определения его членов. Однако переменная может быть назначена экземпляру любого типа, производного от его *типа времени компиляции. Тип времени выполнения* является типом собственно экземпляра, на который ссылается переменная.

При вызове виртуального метода могут быть вызваны разные его реализации в зависимости от того, какой *тип среды выполнения* имеет экземпляр, для которого вызван этот метод. При вызове невиртуального метода решающим фактором является *тип во время компиляции* для этого экземпляра.

Виртуальный метод можно *переопределить* в производном классе. Если объявление метода экземпляра содержит модификатор override, этот метод переопределяет унаследованный виртуальный метод с такой же сигнатурой. Объявление виртуального метода создает новый метод. Переопределение этого метода создает специализированный виртуальный метод с новой реализацией.

Абстрактным методом называется виртуальный метод без реализации. Абстрактный метод объявляется с модификатором `abstract`. Его можно объявить только в абстрактном классе. Абстрактный метод должен обязательно переопределяться в каждом производном классе, не являющемся абстрактным.

Следующий пример кода объявляет абстрактный класс `Expression`, который представляет узел дерева выражений, а также три производных класса: `Constant`, `VariableReference` и `Operation`, которые реализуют узлы дерева выражений для констант, ссылок на переменные и арифметических операций. (Этот пример похож на типы дерева выражений, но не связан с ними.)

```
public abstract class Expression
{
    public abstract double Evaluate(Dictionary<string, object> vars);
}

public class Constant : Expression
{
    double _value;

    public Constant(double value)
    {
        _value = value;
    }

    public override double Evaluate(Dictionary<string, object> vars)
    {
        return _value;
    }
}

public class VariableReference : Expression
{
    string _name;

    public VariableReference(string name)
    {
        _name = name;
    }

    public override double Evaluate(Dictionary<string, object> vars)
    {
        object value = vars[_name] ?? throw new Exception($"Unknown variable: {_name}");
        return Convert.ToDouble(value);
    }
}

public class Operation : Expression
{
    Expression _left;
    char _op;
    Expression _right;

    public Operation(Expression left, char op, Expression right)
    {
        _left = left;
        _op = op;
        _right = right;
    }

    public override double Evaluate(Dictionary<string, object> vars)
    {
        double x = _left.Evaluate(vars);
        double y = _right.Evaluate(vars);
        switch (_op)
        {
            case '+': return x + y;
            case '-': return x - y;
            case '*': return x * y;
            case '/': return x / y;

            default: throw new Exception("Unknown operator");
        }
    }
}
```

Четыре приведенных выше класса можно использовать для моделирования арифметических выражений.

Например, с помощью экземпляров этих классов выражение `x + 3` можно представить следующим образом.

```
Expression e = new Operation(
    new VariableReference("x"),
    '+',
    new Constant(3));
```

Метод `Evaluate` экземпляра `Expression` вызывается для вычисления данного выражения и создает значение `double`. Этот метод принимает аргумент `Dictionary`, который содержит имена переменных (в качестве ключей записей) и значения переменных (в качестве значений записей). Так как `Evaluate` — абстрактный метод, то в неабстрактных классах, производных от `Expression`, необходимо переопределить `Evaluate`.

В `Constant` реализация метода `Evaluate` просто возвращает хранимую константу. В `VariableReference` реализация этого метода выполняет поиск имени переменной в словаре и возвращает полученное значение. В `Operation` реализация этого метода сначала вычисляет левый и правый операнды (рекурсивно вызывая их методы `Evaluate`), а затем выполняет предоставленную арифметическую операцию.

В следующей программе классы `Expression` используются для вычисления выражения `x * (y + 2)` с различными значениями `x` и `y`.

```
Expression e = new Operation(
    new VariableReference("x"),
    '*',
    new Operation(
        new VariableReference("y"),
        '+',
        new Constant(2)
    )
);
Dictionary<string, object> vars = new();
vars["x"] = 3;
vars["y"] = 5;
Console.WriteLine(e.Evaluate(vars)); // "21"
vars["x"] = 1.5;
vars["y"] = 9;
Console.WriteLine(e.Evaluate(vars)); // "16.5"
```

Перегрузка методов

Перегрузка метода позволяет использовать в одном классе несколько методов с одинаковыми именами, если они имеют уникальные сигнатуры. Когда при компиляции встречается вызов перегруженного метода, компилятор использует принцип *разрешения перегрузки*, чтобы определить, какой из методов следует вызвать. Разрешение перегрузки выбирает из методов тот, который лучше всего соответствует предоставленным аргументам. Если не удается выбрать конкретный подходящий метод, возвращается ошибка. В следующем примере показано, как работает разрешение перегрузки. Комментарий к каждому вызову метода `UsageExample` указывает, какой именно метод вызывается.

```
class OverloadingExample
{
    static void F() => Console.WriteLine("F()");
    static void F(object x) => Console.WriteLine("F(object)");
    static void F(int x) => Console.WriteLine("F(int)");
    static void F(double x) => Console.WriteLine("F(double)");
    static void F<T>(T x) => Console.WriteLine("F<T>(T)");
    static void F(double x, double y) => Console.WriteLine("F(double, double)");

    public static void UsageExample()
    {
        F();            // Invokes F()
        F(1);           // Invokes F(int)
        F(1.0);         // Invokes F(double)
        F("abc");       // Invokes F<string>(string)
        F((double)1);   // Invokes F(double)
        F((object)1);   // Invokes F(object)
        F<int>(1);      // Invokes F<int>(int)
        F(1, 1);        // Invokes F(double, double)
    }
}
```

Как видно из этого примера, вы всегда можете выбрать конкретный метод, явным образом приведя типы аргументов к соответствующим типам параметров и аргументам типов.

Другие функции-члены

Все члены класса, содержащие исполняемый код, совокупно называются *функции-члены*. В предыдущем разделе описаны основные варианты методов, используемых в качестве функций-членов. В этом разделе описываются другие типы функций-членов, поддерживаемые в языке C#: конструкторы, свойства, индексаторы, события, операторы и методы завершения.

В следующем примере показан универсальный класс с именем `MyList<T>`, который реализует расширяемый список объектов. Этот класс содержит несколько наиболее распространенных типов функций-членов.

```
public class MyList<T>
{
    const int DefaultCapacity = 4;

    T[] _items;
    int _count;

    public MyList(int capacity = DefaultCapacity)
    {
        _items = new T[capacity];
    }

    public int Count => _count;

    public int Capacity
    {
        get =>  _items.Length;
        set
        {
            if (value < _count) value = _count;
            if (value != _items.Length)
            {
                T[] newItems = new T[value];
                Array.Copy(_items, 0, newItems, 0, _count);
                _items = newItems;
            }
        }
```

```
    }

    public T this[int index]
    {
        get => _items[index];
        set
        {
            _items[index] = value;
            OnChanged();
        }
    }

    public void Add(T item)
    {
        if (_count == Capacity) Capacity = _count * 2;
        _items[_count] = item;
        _count++;
        OnChanged();
    }
    protected virtual void OnChanged() =>
        Changed?.Invoke(this, EventArgs.Empty);

    public override bool Equals(object other) =>
        Equals(this, other as MyList<T>);

    static bool Equals(MyList<T> a, MyList<T> b)
    {
        if (Object.ReferenceEquals(a, null)) return Object.ReferenceEquals(b, null);
        if (Object.ReferenceEquals(b, null) || a._count != b._count)
            return false;
        for (int i = 0; i < a._count; i++)
        {
            if (!object.Equals(a._items[i], b._items[i]))
            {
                return false;
            }
        }
        return true;
    }

    public event EventHandler Changed;

    public static bool operator ==(MyList<T> a, MyList<T> b) =>
        Equals(a, b);

    public static bool operator !=(MyList<T> a, MyList<T> b) =>
        !Equals(a, b);
}
```

Конструкторы

C# поддерживает конструкторы экземпляров и статические конструкторы. *Конструктор экземпляра* является членом, который реализует действия для инициализации нового экземпляра класса. *Статический конструктор* является членом, который реализует действия для инициализации самого класса при первоначальной его загрузке.

Конструктор объявляется в виде метода без возвращаемого типа, имя которого совпадает с именем класса, в котором он определен. Если объявление конструктора содержит модификатор `static`, объявляется статический конструктор. В противном случае это объявление считается конструктором экземпляра.

Конструкторы экземпляров можно перегружать, и для них можно указать необязательные параметры. Например, класс `MyList<T>` объявляет один конструктор экземпляра с одним необязательным параметром `int`. Конструкторы экземпляров вызываются с помощью оператора `new`. Следующий пример кода выделяет два экземпляра `MyList<string>` с помощью конструкторов класса `MyList`: один с

необязательным аргументом, а второй — без.

```
MyList<string> list1 = new();
MyList<string> list2 = new(10);
```

В отличие от других членов, конструкторы экземпляров не наследуются. Класс не имеет конструкторов экземпляров, кроме объявленных в самом этом классе. Если в классе не объявлен конструктор экземпляра, для него автоматически создается пустой конструктор без параметров.

Свойства

Свойства естественным образом дополняют поля. И те, и другие являются именованными членами со связанными типами, и для доступа к ним используется одинаковый синтаксис. Однако свойства, в отличие от полей, не указывают места хранения. Вместо этого свойства содержат *методы доступа*, в которых описаны операторы, выполняемые при чтении или записи значений. *Метод доступа Get* считывает значение. *Метод доступа Set* записывает значение.

Свойство объявляется так же, как поле, за исключением того, что объявление заканчивается не точкой с запятой, а парой разделителей `{` и `}`, между которыми указан метод доступа get или set. Свойство, которое имеет и метод доступа get, и метод доступа set, является *свойством для чтения и записи*. Свойство, имеющее только метод доступа get, является *свойством только для чтения*. Свойство, имеющее только метод доступа set, является *свойством только для записи*.

Акцессор get оформляется как метод без параметров, у которого тип возвращаемого значения совпадает с типом, установленным для этого свойства. Метод доступа set соответствует методу с одним именованным значением параметра и без возвращаемого типа. Метод доступа get вычисляет значение свойства. Метод доступа set предоставляет новое значение для свойства. При ссылке на свойство в качестве назначения в операторе присваивания или в качестве операнда для `++` или `--` вызывается метод доступа set. В остальных случаях при ссылке на свойство вызывается метод доступа get.

Класс `MyList<T>` объявляет два свойства: `Count` (только для чтения) и `Capacity` (только для записи). Следующий код содержит пример использования этих свойств:

```
MyList<string> names = new();
names.Capacity = 100;   // Invokes set accessor
int i = names.Count;    // Invokes get accessor
int j = names.Capacity; // Invokes get accessor
```

Как и в отношении полей и методов, C# поддерживает свойства экземпляра и статические свойства. Статические свойства объявляются с модификатором static, а свойства экземпляра — без него.

Акцессоры свойства могут быть виртуальными. Если объявление свойства содержит модификатор `virtual`, `abstract` или `override`, этот модификатор применяется к акцессорам свойства.

Индексаторы

Индексатор является членом, позволяющим индексировать объекты так, как будто они включены в массив. Индексатор объявляется так же, как свойство, за исключением того, что именем элемента является `this`, а за этим именем следует список параметров, находящийся между разделителями `[` и `]`. Эти параметры доступны в акцессорах индексатора. Как и свойства, можно объявить индексаторы для чтения и записи, только для чтения или только для записи. Кроме того, поддерживаются виртуальные акцессоры индексатора.

Класс `MyList<T>` объявляет один индексатор для чтения и записи, который принимает параметр `int`. Индексатор позволяет индексировать экземпляры `MyList<T>` значениями с типом `int`. Например:

```
MyList<string> names = new();
names.Add("Liz");
names.Add("Martha");
names.Add("Beth");
for (int i = 0; i < names.Count; i++)
{
    string s = names[i];
    names[i] = s.ToUpper();
}
```

Индексаторы могут быть перегружены. В одном классе можно объявить несколько индексаторов, если у них различаются количество или типы параметров.

События

Событие — это член, с помощью которого класс или объект предоставляют уведомления. Объявление события выглядит так же, как объявление поля, но содержит ключевое слово `event` и обязано иметь тип делегата.

В классе, который объявляет член события, это событие действует как обычное поле с типом делегата (если это событие не является абстрактным и не объявляет методы доступа). Это поле хранит ссылку на делегат, который представляет добавленные к событию обработчики событий. Если обработчики событий отсутствуют, это поле имеет значение `null`.

Класс `MyList<T>` объявляет один член события с именем `Changed`, который обрабатывает добавление нового элемента. Событие Changed вызывается виртуальным методом `OnChanged`, который сначала проверяет, не имеет ли это событие значение `null` (это означает, что обработчики отсутствуют). Концепция создания события в точности соответствует вызову делегата, представленного этим событием. Это позволяет обойтись без особой языковой конструкции для создания событий.

Клиенты реагируют на события посредством *обработчиков событий*. Обработчики событий можно подключать с помощью оператора `+=` и удалять с помощью оператора `-=`. Следующий пример кода подключает обработчик события `Changed` к событию `MyList<string>`.

```
class EventExample
{
    static int s_changeCount;

    static void ListChanged(object sender, EventArgs e)
    {
        s_changeCount++;
    }

    public static void Usage()
    {
        var names = new MyList<string>();
        names.Changed += new EventHandler(ListChanged);
        names.Add("Liz");
        names.Add("Martha");
        names.Add("Beth");
        Console.WriteLine(s_changeCount); // "3"
    }
}
```

Для более сложных сценариев, требующих контроля над базовым хранилищем события, в объявлении события можно явным образом предоставить методы доступа `add` и `remove`. Они будут действовать аналогично методу доступа `set` для свойства.

Операторы

Оператор является членом, который определяет правила применения определенного выражения к экземплярам класса. Вы можете определить операторы трех типов: унарные операторы, двоичные операторы и операторы преобразования. Все операторы объявляются с модификаторами `public` и `static`.

В классе `MyList<T>` объявляются два оператора: `operator ==` и `operator !=`. Эти переопределенные операторы придают новое значение выражениям, которые применяют эти операторы к экземплярам `MyList`. В частности, они определяют, что равенство двух экземпляров `MyList<T>` проверяется путем сравнения всех содержащихся в них объектов с помощью определенных для них методов `Equals`. Следующий пример кода использует оператор `==` для сравнения двух экземпляров `MyList<int>`.

```
MyList<int> a = new();
a.Add(1);
a.Add(2);
MyList<int> b = new();
b.Add(1);
b.Add(2);
Console.WriteLine(a == b);  // Outputs "True"
b.Add(3);
Console.WriteLine(a == b);  // Outputs "False"
```

Первый `Console.WriteLine` выводит `True`, поскольку два списка содержат одинаковое число объектов с одинаковыми значениями в том же порядке. Если бы в `MyList<T>` не было определения `operator ==`, первый `Console.WriteLine` возвращал бы `False`, поскольку `a` и `b` указывают на различные экземпляры `MyList<int>`.

Методы завершения

Метод завершения является членом, который реализует действия для завершения существования экземпляра класса. Как правило, метод завершения необходим для освобождения неуправляемых ресурсов. Методы завершения не могут иметь параметры, не могут содержать модификаторы доступа, и их невозможно вызвать явным образом. Метод завершения для экземпляра вызывается автоматически в процессе сборки мусора. Дополнительные сведения см. в статье, посвященной методам завершения.

Сборщик мусора имеет широкую степень свободы в выборе времени уничтожения объектов и вызова методов завершения. В частности, время вызова методов завершения не является детерминированным, и эти методы могут выполняться в любом потоке. По этим и некоторым другим причинам методы завершения следует использовать в классах только в крайнем случае, когда невозможны другие решения.

Уничтожение объектов лучше контролировать с помощью инструкции `using`.

Выражения

Выражения создаются из *операндов* и *операторов*. Операторы в выражении указывают, какие действия нужно применить к операндам. Примеры операторов: `+`, `-`, `*`, `/` и `new`. Операндами могут являться литералы, поля, локальные переменные, выражения и т. п.

Если выражение содержит несколько операторов, их *приоритет* определяет порядок, в котором они оцениваются. Например, выражение `x + y * z` вычисляется как `x + (y * z)`, поскольку оператор `*` имеет более высокий приоритет, чем оператор `+`.

Если операнд располагается между двумя операторами с одинаковым приоритетом, порядок их выполнения определяется *ассоциативностью* операторов.

- Все бинарные операторы, за исключением операторов объединения со значением NULL и операторов присваивания, являются *левоассоциативными*, т. е. эти операции выполняются слева направо.

Например, выражение `x + y + z` вычисляется как `(x + y) + z`.

- Операторы присваивания, операторы объединения со значением NULL `??` и `??=`, а также условный оператор `?:` являются *правоассоциативными*, т. е. эти операции выполняются справа налево. Например, выражение `x = y = z` вычисляется как `x = (y = z)`.

Приоритет и ассоциативность операторов можно изменять, используя скобки. Например, в выражении `x + y * z` сначала `y` умножается на `z`, а результат прибавляется к `x`, а в выражении `(x + y) * z` сначала суммируются `x` и `y`, а результат умножается на `z`.

Большинство операторов могут быть *перегружены*. Перегрузка операторов позволяет создать пользовательскую реализацию оператора для таких операций, в которых один или оба операнда имеют определяемый пользователем тип класса или структуры.

C# предоставляет несколько операторов для выполнения арифметических, логических операций, побитовых операций и сдвигов, а также сравнений на равенство и порядок.

Полный список операторов C#, упорядоченных по уровню приоритета, см. в статье Операторы C#.

Операторы

Действия программы выражаются с помощью *операторов*. C# поддерживает несколько типов операторов, некоторые из которых определяются как внедренные операторы.

- С помощью *блоков* можно использовать несколько операторов в таких контекстах, где ожидается только один оператор. Блок состоит из списка инструкций, заключенных между разделителями `{` и `}`.
- *Операторы объявления* используются для объявления локальных переменных и констант.
- *Операторы выражений* позволяют вычислять выражения. В качестве оператора можно использовать такие выражения, как вызовы методов, выделение объектов с помощью оператора `new`, назначения с помощью `=` и составных операторов присваивания, операторы `++` и `--` для приращения и уменьшения, а также выражения `await`.
- *Операторы выбора* используются для выбора одного оператора из нескольких возможных вариантов в зависимости от значения какого-либо выражения. Эта группа содержит операторы `if` и `switch`.
- *Операторы итерации* используются для многократного выполнения внедренного оператора. Эта группа содержит операторы `while`, `do`, `for` и `foreach`.
- *Операторы перехода* используются для передачи управления. Эта группа содержит операторы `break`, `continue`, `goto`, `throw`, `return` и `yield`.
- Операторы `try` ... `catch` позволяют перехватывать исключения, создаваемые при выполнении блока кода, а оператор `try` ... `finally` используется для указания кода завершения, который выполняется всегда, независимо от появления исключений.
- Операторы `checked` и `unchecked` операторы позволяют управлять контекстом проверки переполнения для целочисленных арифметических операций и преобразований.
- Оператор `lock` позволяет создать взаимоисключающую блокировку заданного объекта перед выполнением определенных операторов, а затем снять блокировку.
- Оператор `using` используется для получения ресурса перед определенным оператором, и для удаления ресурса после его завершения.

Ниже перечислены виды операторов, которые можно использовать.

- Объявление локальной переменной.
- Объявление локальной константы.
- Оператор выражений.
- Оператор `if`.

- Оператор `switch`.
- Оператор `while`.
- Оператор `do`.
- Оператор `for`.
- Оператор `foreach`.
- Оператор `break`.
- Оператор `continue`.
- Оператор `goto`.
- Оператор `return`.
- Оператор `yield`.
- Операторы `throw` и `try`.
- Операторы `checked` и `unchecked`.
- Оператор `lock`.
- Оператор `using`.

Основные области языка

18.12.2021 • 7 minutes to read

Массивы, коллекции и LINQ

В C# и .NET имеется множество различных типов коллекций. Синтаксис массивов определяется языком. Универсальные типы коллекций перечислены в пространстве имен System.Collections.Generic. К специализированным коллекциям относятся System.Span<T> для доступа к непрерывной памяти в кадре стека и System.Memory<T> для доступа к непрерывной памяти в управляемой куче. Все коллекции, включая массивы, Span<T> и Memory<T>, используют общий принцип итерации. Используется интерфейс System.Collections.Generic.IEnumerable<T>. Этот единый принцип означает, что любой из типов коллекций можно использовать с запросами LINQ или другими алгоритмами. Методы пишутся с помощью IEnumerable<T>, и алгоритмы работают с любой коллекцией.

Массивы

*__Массив__ _ — это структура данных, содержащая несколько переменных, доступ к которым осуществляется по вычисляемым индексам. Все содержащиеся в массиве переменные, также называемые *элементами массива*, относятся к одному типу. Он называется _ *типом элементов** массива.

Сами массивы имеют ссылочный тип, и объявление переменной массива только выделяет память для ссылки на экземпляр массива. Фактические экземпляры массива создаются динамически во время выполнения с помощью оператора `new`. Операция `new` указывает ***длину*** нового экземпляра массива, которая остается неизменной в течение всего времени существования этого экземпляра. Элементы массива имеют индексы в диапазоне от `0` до `Length - 1`. Оператор `new` автоматически инициализирует все элементы массива значением по умолчанию. Например, для всех числовых типов устанавливается нулевое значение, а для всех ссылочных типов — значение `null`.

Следующий пример кода создает массив из `int` элементов, затем инициализирует этот массив и выводит содержимое массива.

```
int[] a = new int[10];
for (int i = 0; i < a.Length; i++)
{
    a[i] = i * i;
}
for (int i = 0; i < a.Length; i++)
{
    Console.WriteLine($"a[{i}] = {a[i]}");
}
```

Этот пример создает и использует ***одномерный массив*** _. *Кроме этого, C# поддерживает _многомерные массивы_. Число измерений массива, которое именуется _ рангом* для типа массива, всегда на единицу больше числа запятых, включенных в квадратные скобки типа массива. Следующий пример кода поочередно создает одномерный, двухмерный и трехмерный массивы.

```
int[] a1 = new int[10];
int[,] a2 = new int[10, 5];
int[,,] a3 = new int[10, 5, 2];
```

Массив `a1` содержит 10 элементов, массив `a2` — 50 элементов (10 × 5), и наконец `a3` содержит 100 элементов (10 × 5 × 2). Элементы массива могут иметь любой тип, в том числе тип массива. Массив с

элементами типа массива иногда называют ***массивом массивов***, так как элементы такого массива не обязаны иметь одинаковую длину. Следующий пример создает массив массивов `int`.

```
int[][] a = new int[3][];
a[0] = new int[10];
a[1] = new int[5];
a[2] = new int[20];
```

В первой строке создается массив с тремя элементами, каждый из которых имеет тип `int[]` и начальное значение `null`. В следующих строках эти три элемента инициализируются ссылками на отдельные экземпляры массивов различной длины.

Оператор `new` позволяет задать начальные значения элементов массива, используя ***инициализатор массива***. Так называется список выражений, записанных между разделителями `{` и `}`. Следующий пример создает и инициализирует массив `int[]` с тремя элементами.

```
int[] a = new int[] { 1, 2, 3 };
```

Длина массива определяется по числу выражений между скобками `{` и `}`. Инициализацию массива можно сократить, так как тип массива не обязательно объявлять повторно.

```
int[] a = { 1, 2, 3 };
```

Оба приведенных выше примера дают результат, эквивалентный следующему коду:

```
int[] t = new int[3];
t[0] = 1;
t[1] = 2;
t[2] = 3;
int[] a = t;
```

Оператор `foreach` можно использовать для перечисления элементов любой коллекции. Следующий код перечисляет массив из предыдущего примера:

```
foreach (int item in a)
{
    Console.WriteLine(item);
}
```

Оператор `foreach` использует интерфейс IEnumerable<T>, поэтому может работать с любой коллекцией.

Интерполяция строк

Интерполяция строк в C# позволяет форматировать строки, определяя выражения, результаты которых помещаются в строку формата. Например, в следующем примере выводится температура в указанный день из набора данных о погоде:

```
Console.WriteLine($"The low and high temperature on {weatherData.Date:MM-DD-YYYY}");
Console.WriteLine($"    was {weatherData.LowTemp} and {weatherData.HighTemp}.");
// Output (similar to):
// The low and high temperature on 08-11-2020
//     was 5 and 30.
```

Интерполированная строка объявляется с помощью токена `$`. При интерполяции строк вычисляются выражения между `{` и `}`, результат преобразуется в `string`, а затем текст в квадратных скобках заменяется строковым результатом выражения. `:` в первом выражении `{weatherData.Date:MM-DD-YYYY}` указывает *строку формата*. В предыдущем примере указывается, что дата должна выводиться в формате "ММ-ДД-ГГГГ".

Регулярные выражения

В языке C# есть выражения ***сопоставления шаблонов*** для запроса состояния объекта и выполнения кода в зависимости от него. Чтобы определить, какое действие следует предпринять, можно проверить типы и значения свойств и полей. Выражение `switch` является основным для сопоставления шаблонов.

Делегаты и лямбда-выражения

***Тип** delegate* представляет ссылки на методы с конкретным списком параметров и типом возвращаемого значения. Делегаты позволяют использовать методы как сущности, сохраняя их в переменные и передавая в качестве параметров. Принцип работы делегатов близок к указателям функций из некоторых языков. В отличие от указателей функций, делегаты являются объектно-ориентированными и типобезопасными.

Следующий пример кода объявляет и использует тип делегата с именем `Function`.

```
delegate double Function(double x);

class Multiplier
{
    double _factor;

    public Multiplier(double factor) => _factor = factor;

    public double Multiply(double x) => x * _factor;
}

class DelegateExample
{
    static double[] Apply(double[] a, Function f)
    {
        var result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }

    public static void Main()
    {
        double[] a = { 0.0, 0.5, 1.0 };
        double[] squares = Apply(a, (x) => x * x);
        double[] sines = Apply(a, Math.Sin);
        Multiplier m = new(2.0);
        double[] doubles = Apply(a, m.Multiply);
    }
}
```

Экземпляр `Function` с типом делегата может ссылаться на любой метод, который принимает аргумент `double` и возвращает значение `double`. Метод `Apply` применяет заданный `Function` к элементам `double[]` и возвращает `double[]` с результатами. В методе `Main` используется `Apply` для применения трех различных функций к `double[]`.

Делегат может ссылаться на статический метод (например, `Square` или `Math.Sin` в предыдущем примере) или метод экземпляра (например, `m.Multiply` в предыдущем примере). Делегат, который ссылается на

метод экземпляра, также содержит ссылку на конкретный объект. Когда метод экземпляра вызывается через делегат, этот объект превращается в `this` в вызове.

Делегаты могут также создаваться с использованием анонимных функций или лямбда-выражений, то есть "встроенных методов", создаваемых при объявлении. Анонимные функции могут использовать локальные переменные соседних методов. В следующем примере не создается класс:

```
double[] doubles = Apply(a, (double x) => x * 2.0);
```

Делегат не имеет информации или ограничений в отношении того, к какому классу относится метод, на который он ссылается. Метод, на который указывает ссылка, должен иметь те же параметры и тип возвращаемого значения, что и делегат.

async и await

C# поддерживает асинхронные программы с двумя ключевыми словами: `async` и `await`. Чтобы объявить метод как асинхронный, нужно добавить модификатор `async` в его объявление. Оператор `await` предписывает компилятору асинхронно ожидать результата для завершения выполнения. Управление возвращается вызывающему объекту, а метод возвращает структуру, которая управляет состоянием асинхронной работы. Обычно структурой является System.Threading.Tasks.Task<TResult>, но это также может быть любой тип, поддерживающий шаблон ожидания. Эти возможности позволяют писать код, который считывается как синхронный аналог, но выполняется асинхронно. Например, следующий код скачивает домашнюю страницу документации Майкрософт:

```
public async Task<int> RetrieveDocsHomePage()
{
    var client = new HttpClient();
    byte[] content = await client.GetByteArrayAsync("https://docs.microsoft.com/");

    Console.WriteLine($"{nameof(RetrieveDocsHomePage)}: Finished downloading.");
    return content.Length;
}
```

В этом небольшом примере показаны основные особенности асинхронного программирования:

- Объявление метода включает в себя модификатор `async`.
- Тело метода (`await`) ожидает возврата управления методом `GetByteArrayAsync`.
- Тип, указанный в операторе `return`, соответствует аргументу типа в объявлении `Task<T>` для метода. (Метод, возвращающий `Task`, будет использовать операторы `return` без аргументов.)

Атрибуты

Типы, члены и другие сущности в программе C# поддерживают модификаторы, которые управляют некоторыми аспектами их поведения. Например, доступность метода определяется с помощью модификаторов `public`, `protected`, `internal` и `private`. C# обобщает эту возможность, позволяя пользователям определять собственные типы декларативных сведений, назначать их для сущностей программы и извлекать во время выполнения. В программах эти дополнительные декларативные сведения определяются и используются посредством ***атрибутов***.

Следующий пример кода объявляет атрибут `HelpAttribute`, который можно поместить в сущности программы для указания связей с соответствующей документацией.

```
public class HelpAttribute : Attribute
{
    string _url;
    string _topic;

    public HelpAttribute(string url) => _url = url;

    public string Url => _url;

    public string Topic
    {
        get => _topic;
        set => _topic = value;
    }
}
```

Все классы атрибутов являются производными от базового класса Attribute, который предоставляется в библиотеке .NET. Чтобы задать атрибут, его имя и возможные аргументы указываются в квадратных скобках непосредственно перед объявлением соответствующей сущности. Если имя атрибута заканчивается на `Attribute`, этот суффикс можно опускать при указании ссылки на этот атрибут. Например, атрибут с именем `HelpAttribute` можно использовать так:

```
[Help("https://docs.microsoft.com/dotnet/csharp/tour-of-csharp/features")]
public class Widget
{
    [Help("https://docs.microsoft.com/dotnet/csharp/tour-of-csharp/features",
    Topic = "Display")]
    public void Display(string text) { }
}
```

Этот пример кода присоединяет атрибут `HelpAttribute` к классу `Widget`. Также он добавляет другой атрибут `HelpAttribute` для метода `Display` в этом классе. Открытые конструкторы класса атрибута указывают, какие сведения необходимо указать при назначении атрибута некоторой сущности программы. Дополнительные сведения можно предоставить через обращения к открытым свойствам класса атрибута, доступным для чтения и записи (например, как указанная выше ссылка на свойство `Topic`).

Метаданные, определенные атрибутами, можно считывать и использовать во время выполнения с помощью отражения. Когда с помощью этого метода выполняется запрос конкретного атрибута, вызывается конструктор для класса атрибута с указанием сведений, представленных в исходном коде программы, а затем возвращается созданный экземпляр атрибута. Если дополнительные сведения предоставляются через свойства, перед возвращением экземпляра атрибута этим свойствам присваиваются указанные значения.

В следующем примере кода показано, как получить экземпляр класса `HelpAttribute`, связанный с классом `Widget` и его методом `Display`.

```
Type widgetType = typeof(Widget);

object[] widgetClassAttributes = widgetType.GetCustomAttributes(typeof(HelpAttribute), false);

if (widgetClassAttributes.Length > 0)
{
    HelpAttribute attr = (HelpAttribute)widgetClassAttributes[0];
    Console.WriteLine($"Widget class help URL : {attr.Url} - Related topic : {attr.Topic}");
}

System.Reflection.MethodInfo displayMethod = widgetType.GetMethod(nameof(Widget.Display));

object[] displayMethodAttributes = displayMethod.GetCustomAttributes(typeof(HelpAttribute), false);

if (displayMethodAttributes.Length > 0)
{
    HelpAttribute attr = (HelpAttribute)displayMethodAttributes[0];
    Console.WriteLine($"Display method help URL : {attr.Url} - Related topic : {attr.Topic}");
}
```

Дополнительные сведения

Узнать больше о C# можно с помощью наших учебников.

НАЗАД

Знакомство с C#

18.12.2021 • 2 minutes to read

Приветствуем вас в ознакомительном разделе руководств по C#. Занятия начинаются с интерактивного кода, который можно запускать в браузере. Прежде чем приступить к интерактивным урокам, вы можете ознакомиться с основами C#, просмотрев серию видеороликов "C# для начинающих".

В первых занятиях с помощью небольших фрагментов кода объясняются основные понятия языка C#. Вы изучите основы синтаксиса C# и научитесь работать с такими типами данных, как строки, числа и логические значения. Вся серия интерактивна, и уже через считанные минуты вы будете писать и запускать собственный код. Для первых занятий не требуются какие-либо знания в области программировании или опыт работы с языком C#.

Эти руководства можно проходить в разных средах. В них описываются одни и те же основные понятия. Различие в том, где именно вы будете работать.

- В браузере на платформе документации. В этом случае окно готового к запуску кода C# внедряется на страницы документации. Вы пишете и выполняете C# код в браузере.
- В среде Microsoft Learn. Эта схема обучения содержит несколько модулей для изучения основ C# .
- В Jupyter в проекте Binder. Вы можете поэкспериментировать с кодом C# в записной книжке Jupyter в проекте Binder.
- На локальном компьютере. После изучения кода в интерактивном режиме можно скачать пакет SDK для .NET и создавать программы на компьютере.

Все ознакомительные руководства, в которых используется пример приложения "Hello World", можно проходить в веб-браузере или в локальной среде разработки. В конце каждого руководства вам предлагается на выбор возможность пройти следующее занятие в браузере или на локальном компьютере. Чтобы настроить среду и продолжить изучение следующего руководства на компьютере, можно воспользоваться соответствующими ссылками.

Hello world

В руководстве Hello World вы создадите самую простую программу на C#. Вы ознакомитесь с типом `string` и способами работы с текстом. Можно также использовать схему обучения в Microsoft Learn или Jupyter в проекте Binder.

Числа в C#

Из руководства Числа в C# вы узнаете, как на компьютере хранятся числа и как выполнять вычисления с разными числовыми типами. Вы ознакомитесь с основами округления и научитесь выполнять математические вычисления с помощью C#. Это руководство можно изучить, используя локальный компьютер.

В этом руководстве предполагается, что вы уже прошли занятие Hello World.

Ветви и циклы

В руководстве Ветви и циклы представлены общие принципы организации ветвления кода в зависимости от значений, хранящихся в переменных. Вы узнаете, что такое поток управления, являющийся основой принятия решений и выбора различных действий в программах. Это руководство можно изучить,

используя локальный компьютер.

В этом руководстве предполагается, что вы уже прошли занятия Hello World и Числа в C#.

Коллекция списков

Занятие Коллекция списков содержит обзор типа "Коллекция списков", в котором хранятся последовательности данных. Вы узнаете, как добавлять и удалять элементы, выполнять их поиск и сортировать списки. Вы ознакомитесь с различными типами списков. Это руководство можно изучить, используя локальный компьютер.

В этом руководстве предполагается, что вы уже прошли перечисленные выше занятия.

101 пример по LINQ

Для работы с этим примером требуется глобальный инструмент dotnet-try. После установки инструмента и клонирования репозитория try-samples можно изучить LINQ с помощью набора из 101 примера, которые можно запускать в интерактивном режиме. Вы можете исследовать различные способы отправки запросов к данным, исследования данных и преобразования последовательностей данных.

Настройка локальной среды

18.12.2021 • 2 minutes to read

Для выполнения этого руководства прежде всего нужно настроить на компьютере среду разработки. Выберите один из следующих вариантов:

- Сведения об использовании .NET CLI и любого текстового редактора или редактора кода см. в статье Руководство по .NET — Hello World за 10 минут. В этом руководстве содержатся инструкции по настройке локальной среды разработки в macOS, Windows или Linux.
- Чтобы использовать .NET CLI и Visual Studio Code, установите пакет SDK для .NET и Visual Studio Code.
- Сведения об использовании Visual Studio 2019, см. в статье Руководство. Создание простого консольного приложения C# в Visual Studio.

Базовый поток разработки приложения

В инструкциях, приведенных в этих руководствах, предполагается, что вы используете .NET CLI для создания, сборки и запуска приложений. Вы будете использовать следующие команды:

- `dotnet new` создает приложение. Эта команда создает файлы и ресурсы, необходимые для приложения. Во всех ознакомительных руководствах по C# используется тип приложения `console`. Освоив описанные здесь основы, вы сможете перейти и к другим типам приложений.
- `dotnet build` выполняет сборку исполняемого файла.
- `dotnet run` запускает исполняемый файл.

Если вы используете Visual Studio 2019 для работы с этими руководствами, выберите вариант с меню Visual Studio, когда в руководстве будет предложено выполнить одну из следующих команд CLI:

- **Файл** > **Создать** > **Проект** — создание приложения;
  - Рекомендуется использовать шаблон проекта `Console Application`.
  - Вам будет предоставлена возможность указать целевую платформу. Приведенные ниже руководства лучше подходят для .NET 5 и более поздних версий.
- **Сборка** > **Собрать решение** — сборка исполняемого файла;
- **Отладка** > **Запуск без отладки** — запуск исполняемого файла.

Выбор руководства

Вы можете начать изучение с любого из следующих руководств:

Числа в C#

Из руководства Числа в C# вы узнаете, как на компьютере хранятся числа и как выполнять вычисления с разными числовыми типами. Вы ознакомитесь с основами округления и научитесь выполнять математические вычисления с помощью C#.

В этом руководстве предполагается, что вы уже прошли занятие Hello World.

Ветви и циклы

В руководстве Ветви и циклы представлены общие принципы организации ветвления кода в зависимости от значений, хранящихся в переменных. Вы узнаете, что такое поток управления, являющийся основой

принятия решений и выбора различных действий в программах.

В этом руководстве предполагается, что вы уже прошли занятия Hello World и Числа в C#.

Коллекция списков

Занятие Коллекция списков содержит обзор типа "Коллекция списков", в котором хранятся последовательности данных. Вы узнаете, как добавлять и удалять элементы, выполнять их поиск и сортировать списки. Вы ознакомитесь с различными типами списков.

В этом руководстве предполагается, что вы уже прошли перечисленные выше занятия.

Управление целыми числами и числами с плавающей запятой в C#

18.12.2021 • 8 minutes to read

Это руководство поможет в интерактивном изучении числовых типов в C#. Вы напишете небольшие фрагменты кода, затем скомпилируете и выполните этот код. Руководство содержит ряд уроков, в которых рассматриваются числа и математические операции в C#. В рамках этих занятий вы ознакомитесь с основами языка C#.

Предварительные требования

Для работы с руководством вам потребуется компьютер, настроенный для разработки в локальной среде. В Windows, Linux или macOS для создания, сборки и запуска приложений можно использовать .NET CLI. Для Mac или Windows можно использовать Visual Studio 2019. Инструкции по настройке см. в статье Настройка локальной среды.

Вычисления с целыми числами

Создайте каталог с именем *numbers-quickstart*. Сделайте его текущим и выполните следующую команду:

```
dotnet new console -n NumbersInCSharp -o .
```

IMPORTANT

В шаблонах C# для .NET 6 используются *операторы верхнего уровня*. Ваше приложение может не соответствовать коду в этой статье, если вы уже выполнили обновление до предварительных версий .NET 6. Дополнительные сведения см. в статье Новые шаблоны C# для создания инструкций верхнего уровня.

Пакет SDK для .NET 6 также добавляет набор *неявных* директив `global using` для проектов, использующих следующие пакеты SDK:

- Microsoft.NET.Sdk
- Microsoft.NET.Sdk.Web
- Microsoft.NET.Sdk.Worker

Эти неявные директивы `global using` включают наиболее распространенные пространства имен для соответствующего типа проектов.

Откройте файл *Program.cs* в любом редакторе и замените содержимое файла следующим кодом:

```
using System;

int a = 18;
int b = 6;
int c = a + b;
Console.WriteLine(c);
```

Чтобы выполнить этот код, введите `dotnet run` в окно командной строки.

Вы увидели одну из основных математических операций с целыми числами. Тип `int` представляет

целое положительное или отрицательное число или ноль. Для сложения используйте символ `+`. Другие стандартные математические операции с целыми числами включают:

- `-` — вычитание;
- `*` — умножение;
- `/` — деление.

Начните с ознакомления с различными операциями. Добавьте следующие строки после строки, с помощью которой записывается значение `c`:

```
// subtraction
c = a - b;
Console.WriteLine(c);

// multiplication
c = a * b;
Console.WriteLine(c);

// division
c = a / b;
Console.WriteLine(c);
```

Чтобы выполнить этот код, введите `dotnet run` в окно командной строки.

Можно также поэкспериментировать, написав несколько математических операций в одной строке. Например, выполните `c = a + b - 12 * 17;`. Допускается сочетание переменных и постоянных чисел.

> **TIP**
> Вероятнее всего, при изучении C# (как и любого другого языка программирования) вы будете допускать ошибки в коде. **Компилятор** найдет эти ошибки и сообщит вам о них. Если результат содержит сообщения об ошибках, внимательно просмотрите пример кода и код в окне, чтобы понять, что нужно исправить. Это упражнение поможет вам изучить структуру кода C#.

Вы завершили первый этап. Прежде чем перейти к следующему разделу, переместим текущий код в отдельный *метод*. Метод — это последовательность инструкций, сгруппированных под одним именем. Чтобы вызвать метод, нужно указать имя метода и `()` после него. Упорядочение кода в методы упростит работу с новым примером. В результате ваш код должен выглядеть примерно следующим образом:

```
using System;

WorkWithIntegers();

void WorkWithIntegers()
{
    int a = 18;
    int b = 6;
    int c = a + b;
    Console.WriteLine(c);


    // subtraction
    c = a - b;
    Console.WriteLine(c);

    // multiplication
    c = a * b;
    Console.WriteLine(c);

    // division
    c = a / b;
    Console.WriteLine(c);
}
```

Строка `WorkWithIntegers();` вызывает метод. Код после этой строки объявляет метод и определяет его.

Изучение порядка операций

Закомментируйте вызов `WorkingWithIntegers()`. Это поможет упорядочить выходные данные в этом разделе.

```
//WorkWithIntegers();
```

`//` запускает **комментарий** в C#. Комментарии — это любой текст, который должен быть сохранен в исходном коде, но не должен выполняться как код. Компилятор не создает исполняемый код из комментариев. Так как `WorkWithIntegers()` является методом, необходимо закомментировать только одну строку.

Язык C# определяет приоритет математических операций в соответствии с правилами математики. Умножение и деление имеют приоритет над сложением и вычитанием. Убедитесь в этом, добавив следующий код после вызова `WorkWithIntegers()` и выполнив `dotnet run`:

```
int a = 5;
int b = 4;
int c = 2;
int d = a + b * c;
Console.WriteLine(d);
```

В выходных данных видно, что умножение выполняется раньше сложения.

Можно применить другую последовательность операций. Для этого операции, которые должны выполняться первыми, нужно заключить в скобки. Добавьте приведенные ниже строки и выполните код еще раз.

```
d = (a + b) * c;
Console.WriteLine(d);
```

Поэкспериментируйте, объединяя различные операции. Добавьте строки, как показано ниже. Выполните `dotnet run` еще раз.

```
d = (a + b) - 6 * c + (12 * 4) / 3 + 12;
Console.WriteLine(d);
```

Возможно, вы заметили интересное поведение целых чисел. Деление целых чисел всегда дает результат в виде целого числа, даже если ожидаемый результат содержит десятичную или дробную часть.

Если вы еще не видели пример такого поведения, выполните следующий код:

```
int e = 7;
int f = 4;
int g = 3;
int h = (e + f) / g;
Console.WriteLine(h);
```

Выполните `dotnet run` еще раз, чтобы просмотреть результаты.

Прежде чем продолжить, давайте поместив весь код, который вы написали в этом разделе, в новый метод. Вызовите этот новый метод `OrderPrecedence`. Код должен выглядеть следующим образом:

```
using System;

// WorkWithIntegers();
OrderPrecedence();

void WorkWithIntegers()
{
    int a = 18;
    int b = 6;
    int c = a + b;
    Console.WriteLine(c);


    // subtraction
    c = a - b;
    Console.WriteLine(c);

    // multiplication
    c = a * b;
    Console.WriteLine(c);

    // division
    c = a / b;
    Console.WriteLine(c);
}

void OrderPrecedence()
{
    int a = 5;
    int b = 4;
    int c = 2;
    int d = a + b * c;
    Console.WriteLine(d);

    d = (a + b) * c;
    Console.WriteLine(d);

    d = (a + b) - 6 * c + (12 * 4) / 3 + 12;
    Console.WriteLine(d);

    int e = 7;
    int f = 4;
    int g = 3;
    int h = (e + f) / g;
    Console.WriteLine(h);
}
```

Изучение точности и ограничений для целых чисел

В последнем примере вы увидели, что при делении целых чисел результат усекается. Вы можете получить **остаток** с помощью оператора **остатка от деления**, который обозначается символом `%`. Попробуйте выполнить следующий код после вызова метода `OrderPrecedence()`:

```
int a = 7;
int b = 4;
int c = 3;
int d = (a + b) / c;
int e = (a + b) % c;
Console.WriteLine($"quotient: {d}");
Console.WriteLine($"remainder: {e}");
```

Тип целых чисел C# характеризуется еще одним отличием от математических целых: тип `int` имеет

минимальные и максимальные ограничения. Чтобы просмотреть эти ограничения, добавьте такой код:

```
int max = int.MaxValue;
int min = int.MinValue;
Console.WriteLine($"The range of integers is {min} to {max}");
```

Если при вычислении выводится значение вне этих пределов, возникает условие **потери значимости** или **переполнения**. Ответ должен находиться в диапазоне от минимального до максимального значения. Добавьте следующие две строки, чтобы увидеть пример:

```
int what = max + 3;
Console.WriteLine($"An example of overflow: {what}");
```

Обратите внимание, что ответ очень близок к минимальному целому числу (отрицательное значение). Он совпадает со значением `min + 2`. Оператор сложения **вызвал переполнение** допустимых значений для целых чисел. Ответ является очень большим отрицательным числом, так как переполнение покрывает диапазон от наибольшего целого числа до наименьшего.

Существуют другие числовые типы с различными ограничениями и точностью, которые можно использовать, если тип `int` не соответствует вашим требованиям. Далее рассмотрим другие типы. Прежде чем перейти к следующему разделу, переместите код, написанный согласно инструкциям из этого раздела, в отдельный метод. Присвойте обработчику события имя `TestLimits`.

Работа с типом double

Числовой тип `double` представляет число с плавающей запятой двойной точности. Эти термины могут быть новыми для вас. Число **с плавающей запятой** можно использовать для представления нецелых чисел, которые могут быть очень большими или малыми. **Число двойной точности** — это относительный термин, описывающий количество двоичных разрядов, используемых для хранения значения. **Числа двойной точности** имеют в два раза больше двоичных символов по сравнению с **числами одиночной точности**. На современных компьютерах числа двойной точности используются чаще, чем одиночной. **Числа одиночной точности** объявляются с помощью ключевого слова `float`. Рассмотрим их. Добавьте следующий код и просмотрите результат:

```
double a = 5;
double b = 4;
double c = 2;
double d = (a + b) / c;
Console.WriteLine(d);
```

Обратите внимание, что ответ включает десятичную долю частного. Попробуйте более сложное выражение с типом double:

```
double e = 19;
double f = 23;
double g = 8;
double h = (e + f) / g;
Console.WriteLine(h);
```

Диапазон значений типа double гораздо больше, чем диапазон значений целых чисел. Добавьте следующий фрагмент после написанного кода:

```
double max = double.MaxValue;
double min = double.MinValue;
Console.WriteLine($"The range of double is {min} to {max}");
```

Значения выводятся в экспоненциальном представлении. Число слева от символа `E` является значащим. Число справа — это показатель степени, который равен 10. Так же, как десятичные числа в математике, значения double в C# могут содержать ошибки округления. Выполните этот код:

```
double third = 1.0 / 3.0;
Console.WriteLine(third);
```

Вы знаете, что периодическая десятичная дробь `0.3` не равняется `1/3`.

Задача

Выполните другие вычисления с большими числами, малыми числами, умножением и делением с помощью типа `double`. Попробуйте выполнить более сложные вычисления. После того как вы решите сложную задачу, поместите написанный код в новый метод. Присвойте этому методу имя `WorkWithDoubles`.

Работа с десятичными типами

Вы уже ознакомились с базовыми числовыми типами в C# — целыми числами и числами типа double. Осталось изучить еще один тип: `decimal`. Тип `decimal` имеет меньший диапазон, но большую точность, чем `double`. Например:

```
decimal min = decimal.MinValue;
decimal max = decimal.MaxValue;
Console.WriteLine($"The range of the decimal type is {min} to {max}");
```

Обратите внимание, что диапазон меньше, чем для типа `double`. Вы можете убедиться в повышении точности при использовании типа decimal, выполнив следующий код:

```
double a = 1.0;
double b = 3.0;
Console.WriteLine(a / b);

decimal c = 1.0M;
decimal d = 3.0M;
Console.WriteLine(c / d);
```

Суффикс `M` возле чисел указывает, что для константы должен использоваться тип `decimal`. В противном случае компилятор предполагает тип `double`.

> **NOTE**
> Буква `M` была выбрана потому, что визуально показывает различия между ключевыми словами `double` и `decimal`.

Обратите внимание, что при вычислении с использованием типа decimal справа от запятой содержится больше цифр.

Задача

Теперь, когда вы ознакомились с разными числовыми типами, напишите код, который позволяет вычислить площадь круга с радиусом 2,50 см. Помните, что площадь круга равна квадрату радиуса, умноженному на число пи. Подсказка: в .NET есть константа пи Math.PI, которую можно использовать. Math.PI, как и все константы, объявленные в пространстве имен `System.Math`, — это значение `double`. По этой причине вместо значений `decimal` для этой задачи следует использовать `double`.

Вы должны получить ответ от 19 до 20. Ответ можно просмотреть в готовом примере кода на GitHub.

При желании поэкспериментируйте с другими формулами.

Вы выполнили все задачи краткого руководства по числам в C#. Теперь вы можете выполнить руководство по ветвям и циклам в своей среде разработки.

Дополнительные сведения о числах в C# см. в следующих статьях:

- Целочисленные типы
- Числовые типы с плавающей запятой
- Встроенные числовые преобразования

Изучение условной логики с операторами ветви и цикла

18.12.2021 • 9 minutes to read

В этом руководстве объясняется, как написать код, который позволяет проверить переменные и изменить путь выполнения на основе этих переменных. Вы напишете код C# и сможете просмотреть результаты его компиляции и выполнения. Это руководство содержит ряд уроков, в которых рассматриваются конструкции ветвления и циклов в C#. В рамках этих занятий вы ознакомитесь с основами языка C#.

Предварительные требования

Для работы с руководством вам потребуется компьютер, настроенный для разработки в локальной среде. В Windows, Linux или macOS для создания, сборки и запуска приложений можно использовать .NET CLI. Для Mac и Windows можно использовать Visual Studio 2019. Инструкции по настройке см. в статье Настройка локальной среды.

Принятие решений с помощью оператора `if`

Создайте каталог с именем *branches-tutorial*. Сделайте его текущим, выполнив следующую команду:

```
dotnet new console -n BranchesAndLoops -o .
```

> **IMPORTANT**
> В шаблонах C# для .NET 6 используются *операторы верхнего уровня*. Ваше приложение может не соответствовать коду в этой статье, если вы уже выполнили обновление до предварительных версий .NET 6. Дополнительные сведения см. в статье Новые шаблоны C# для создания инструкций верхнего уровня.
>
> Пакет SDK для .NET 6 также добавляет набор *неявных* директив `global using` для проектов, использующих следующие пакеты SDK:
>
> - Microsoft.NET.Sdk
> - Microsoft.NET.Sdk.Web
> - Microsoft.NET.Sdk.Worker
>
> Эти неявные директивы `global using` включают наиболее распространенные пространства имен для соответствующего типа проектов.

Эта команда создает консольное приложение .NET в текущем каталоге. Откройте файл *Program.cs* в любом редакторе и замените содержимое следующим кодом:

```
using System;

int a = 5;
int b = 6;
if (a + b > 10)
    Console.WriteLine("The answer is greater than 10.");
```

Чтобы выполнить этот код, введите `dotnet run` в окне консоли. В консоли должно появиться сообщение The answer is greater than 10 (Ответ больше 10). Измените объявление `b`, чтобы сумма была меньше 10:

```
int b = 3;
```

Введите `dotnet run` еще раз. Так как ответ меньше 10, никакие данные не выводятся. Проверяемое **условие** имеет значение false. У вас еще нет кода для выполнения, так как вы написали только одну из возможных ветвей для оператора `if` — ветвь true.

> **TIP**
> Вероятнее всего, при изучении C# (как и любого другого языка программирования) вы будете допускать ошибки в коде. Компилятор найдет ошибки и сообщит о них. Внимательно просмотрите выходные данные ошибки и код, вызвавший ошибку. Как правило, сведения о причине ошибки можно найти в сообщении об ошибке компилятора.

В первом примере показаны возможности `if` и логические типы. *Логическое значение* — это переменная, которая может иметь одно из двух значений: `true` или `false`. Логические переменные в C# определяются особым типом — `bool`. Оператор `if` проверяет значение `bool`. Если значение `true`, выполняется оператор, следующий после `if`. В противном случае он пропускается. Этот процесс проверки условий и выполнения операторов на основе этих условий предоставляет широкие возможности.

Объединение операторов if и else

Чтобы выполнить разный код в ветвях true и false, создайте ветвь `else`, которая будет выполняться, если условие имеет значение false. Попробуйте создать ветвь `else`. Добавьте две последние строки из приведенного ниже кода (первые четыре должны быть уже добавлены):

```
int a = 5;
int b = 3;
if (a + b > 10)
    Console.WriteLine("The answer is greater than 10");
else
    Console.WriteLine("The answer is not greater than 10");
```

Оператор после ключевого слова `else` выполняется, только если проверяемое условие имеет значение `false`. Объединив операторы `if` и `else` с логическими условиями, вы получите все необходимые возможности для обработки условий `true` и `false`.

> **IMPORTANT**
> Отступы под операторами `if` и `else` предназначены только для удобства чтения. В языке C# необязательно ставить отступы или пробелы. Операторы после ключевого слова `if` или `else` будут выполняться на основе условия. Во всех строках в примерах кода, представленных в этом руководстве, отступы традиционно соответствуют потоку управления операторов.

Так как отступ не обязателен, используйте скобки `{` и `}`, если нужно указать несколько операторов в блоке кода, который выполняется в зависимости от условий. Программисты C# обычно используют эти фигурные скобки во всех предложениях `if` и `else`. Следующий пример аналогичен тому, который вы создали. Измените код выше, чтобы он соответствовал следующему коду:

```
int a = 5;
int b = 3;
if (a + b > 10)
{
    Console.WriteLine("The answer is greater than 10");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
}
```

TIP

Все примеры кода в следующих разделах руководства содержат фигурные скобки в соответствии с принятой практикой.

Можно проверить более сложные условия. Добавьте следующий код после написанного кода:

```
int c = 4;
if ((a + b + c > 10) && (a == b))
{
    Console.WriteLine("The answer is greater than 10");
    Console.WriteLine("And the first number is equal to the second");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
    Console.WriteLine("Or the first number is not equal to the second");
}
```

Символ `==` позволяет проверить *равенство*. С помощью `==` обозначается отличие проверки равенства от назначения, которое показано в `a = 5`.

`&&` представляет оператор and. То есть для выполнения оператора в ветви true оба условия должны иметь значение true. В этих примерах также показано, что в каждой условной ветви можно задать несколько операторов. Нужно лишь заключить их в скобки `{` и `}`. Вы также можете использовать оператор `||`, который представляет оператор or. Добавьте следующий фрагмент после написанного кода:

```
if ((a + b + c > 10) || (a == b))
{
    Console.WriteLine("The answer is greater than 10");
    Console.WriteLine("Or the first number is equal to the second");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
    Console.WriteLine("And the first number is not equal to the second");
}
```

Измените значения `a`, `b` и `c`, а также переключитесь между `&&` и `||` для изучения. Так вы лучше поймете, как работают операторы `&&` и `||`.

Вы завершили первый этап. Прежде чем перейти к следующему разделу, переместим текущий код в отдельный метод. Это упростит начало работы с новым примером. Поместите существующий код в метод `ExploreIf()`. Вызовите этот метод в начале программы. После внесения этих изменений код должен выглядеть так:

```
using System;

ExploreIf();

void ExploreIf()
{
    int a = 5;
    int b = 3;
    if (a + b > 10)
    {
        Console.WriteLine("The answer is greater than 10");
    }
    else
    {
        Console.WriteLine("The answer is not greater than 10");
    }

    int c = 4;
    if ((a + b + c > 10) && (a > b))
    {
        Console.WriteLine("The answer is greater than 10");
        Console.WriteLine("And the first number is greater than the second");
    }
    else
    {
        Console.WriteLine("The answer is not greater than 10");
        Console.WriteLine("Or the first number is not greater than the second");
    }

    if ((a + b + c > 10) || (a > b))
    {
        Console.WriteLine("The answer is greater than 10");
        Console.WriteLine("Or the first number is greater than the second");
    }
    else
    {
        Console.WriteLine("The answer is not greater than 10");
        Console.WriteLine("And the first number is not greater than the second");
    }
}
```

Закомментируйте вызов `ExploreIf()`. Это поможет упорядочить выходные данные в этом разделе.

```
//ExploreIf();
```

`//` запускает **комментарий** в C#. Комментарии — это любой текст, который должен быть сохранен в исходном коде, но не должен выполняться как код. Компилятор не создает исполняемый код из комментариев.

Использование циклов для повторения операций

В этом разделе **циклы** используются для повторения операторов. Добавьте следующий код после вызова `ExploreIf`:

```
int counter = 0;
while (counter < 10)
{
    Console.WriteLine($"Hello World! The counter is {counter}");
    counter++;
}
```

Оператор `while` проверяет условие и выполняет инструкцию или блок инструкций, следующий после `while`. Проверка условия и выполнение этих операторов будут повторяться, пока условие не примет значение false.

В этом примере представлен еще один новый оператор. Объект `++` после переменной `counter` представляет собой оператор **инкремента**. Он добавляет 1 к значению `counter` и сохраняет это значение в переменной `counter`.

> **IMPORTANT**
> Напишите такой код, при выполнении которого значение условия цикла `while` изменится на false. В противном случае будет создан **бесконечный цикл**, в котором выполнение программы никогда не закончится. Это не показано в примере, так как нужно принудительно закрыть программу, нажав клавиши CTRL+C, или другим способом.

В цикле `while` условие проверяется, прежде чем выполнить код, который следует после `while`. А в цикле `do` ... `while` сначала выполняется код, а потом проверяется условие. Цикл *do while* показан в следующем примере кода.

```
int counter = 0;
do
{
    Console.WriteLine($"Hello World! The counter is {counter}");
    counter++;
} while (counter < 10);
```

Этот цикл `do` и цикл `while`, приведенный выше, выводят одинаковый результат.

Работа с циклом for

Цикл **for** широко используется в C#. Выполните этот код:

```
for (int index = 0; index < 10; index++)
{
    Console.WriteLine($"Hello World! The index is {index}");
}
```

Этот цикл работает так же, как циклы `while` и `do`, которые вы использовали ранее. Оператор `for` состоит из трех частей, которые отвечают за его работу.

Первая часть — **для инициализатора**: `int index = 0;` объявляет `index` переменной цикла и задает для ее начальное значение `0`.

Средняя часть — **для условия**: `index < 10` объявляет, что этот цикл `for` продолжает выполняться, пока значение счетчика меньше 10.

Последняя часть — **для итератора**: `index++` определяет, как изменится переменная цикла после выполнения блока, следующего после оператора `for`. В нашем случае определяется, что значение `index` должно увеличиваться на 1 каждый раз, когда выполняется блок.

Поэкспериментируйте. Попробуйте использовать следующие варианты.

- Измените инициализатор, чтобы цикл начинался с другого значения.
- Измените условие, чтобы цикл заканчивался другим значением.

По окончании попробуйте самостоятельно написать код, чтобы применить полученные знания.

Есть еще одна циклическая инструкция, которая не рассматривается в этом руководстве: оператор `foreach`. Оператор `foreach` повторяет выполнения для каждого элемента в последовательности элементов. Чаще всего он используется с *коллекциями*, поэтому он рассматривается в следующем руководстве.

Создание вложенных циклов

Цикл `while`, `do` или `for` можно вложить в другой цикл, чтобы создать матрицу, комбинируя каждый элемент во внешнем цикле с каждым элементом во внутреннем цикле. Давайте сделаем это для получения набора из буквенно-цифровых пар, представляющих строки и столбцы.

Один цикл `for` может создавать строки:

```
for (int row = 1; row < 11; row++)
{
    Console.WriteLine($"The row is {row}");
}
```

Другой цикл может создавать столбцы:

```
for (char column = 'a'; column < 'k'; column++)
{
    Console.WriteLine($"The column is {column}");
}
```

Можно вложить один цикл внутрь другого для формирования пар:

```
for (int row = 1; row < 11; row++)
{
    for (char column = 'a'; column < 'k'; column++)
    {
        Console.WriteLine($"The cell is ({row}, {column})");
    }
}
```

Как видите, внешний цикл увеличивается на единицу при каждом полном выполнении внутреннего цикла. Измените порядок вложенности строк и столбцов и просмотрите изменения самостоятельно. По завершении поместите код, приведенный в этом разделе, в метод `ExploreLoops()`.

Объединение ветвей и циклов

Теперь, когда вы ознакомились с оператором `if` и конструкциями цикла на языке C#, попытайтесь написать код C# для поиска суммы всех целых чисел от 1 до 20, которые делятся на 3. Вот несколько подсказок:

- оператор `%` позволяет получить остаток от операции деления;
- оператор `if` предоставляет условие, которое позволяет определить, будет ли число учитываться в сумме;
- цикл `for` позволяет повторить последовательность шагов для всех чисел от 1 до 20.

Попробуйте самостоятельно. Затем проверьте результат. Вы должны получить ответ "63". Один из возможных ответов можно увидеть в полном примере кода в GitHub.

Вы ознакомились с руководством по ветвям и циклам.

Теперь вы можете перейти к ознакомлению с руководством Массивы и коллекции в своей среде разработки.

Дополнительные сведения об этих понятиях см. в следующих статьях:

- Инструкции выбора
- Инструкции итерации

Научитесь управлять коллекциями данных с использованием универсального типа списка

18.12.2021 • 5 minutes to read

Это вводное руководство содержит общие сведения о языке C# и классе List<T>.

Предварительные требования

Для работы с руководством вам потребуется компьютер, настроенный для разработки в локальной среде. В Windows, Linux или macOS для создания, сборки и запуска приложений можно использовать .NET CLI. Для Mac и Windows можно использовать Visual Studio 2019. Инструкции по настройке см. в статье Настройка локальной среды.

Пример простого списка

Создайте каталог с именем *list-tutorial*. Откройте этот каталог и выполните команду `dotnet new console`.

> **IMPORTANT**
> В шаблонах C# для .NET 6 используются *операторы верхнего уровня*. Ваше приложение может не соответствовать коду в этой статье, если вы уже выполнили обновление до предварительных версий .NET 6. Дополнительные сведения см. в статье Новые шаблоны C# для создания инструкций верхнего уровня.
>
> Пакет SDK для .NET 6 также добавляет набор *неявных* директив `global using` для проектов, использующих следующие пакеты SDK:
>
> - Microsoft.NET.Sdk
> - Microsoft.NET.Sdk.Web
> - Microsoft.NET.Sdk.Worker
>
> Эти неявные директивы `global using` включают наиболее распространенные пространства имен для соответствующего типа проектов.

Откройте *Program.cs* в любом редакторе и замените существующий код следующим:

```
using System;
using System.Collections.Generic;

var names = new List<string> { "<name>", "Ana", "Felipe" };
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
```

Замените `<name>` собственным именем. Сохраните *Program.cs*. Введите в окне консоли команду `dotnet run` для тестирования.

Вы создали список строк, добавили три имени в этот список и вывели имена прописными буквами. Для циклического прохода по списку вы примените концепции, которые изучили в предыдущих руководствах.

В коде для отображения имен используется функция интерполяции строк. Если перед `string` добавить

символ `$`, код C# можно внедрять в объявление строки. Фактическая строка заменяет код C# генерируемым значением. В этом примере она заменяет `{name.ToUpper()}` именами, буквы каждого из которых преобразованы в прописные, так как вызван метод ToUpper.

Продолжим изучение.

Изменение содержимого списка

В созданной коллекции используется тип List<T>. При применении такого типа сохраняются последовательности элементов. Тип элементов указывается в угловых скобках.

Важный аспект типа List<T> — возможность увеличения или уменьшения, что позволяет добавлять или удалять элементы. Добавьте этот код в конец программы:

```
Console.WriteLine();
names.Add("Maria");
names.Add("Bill");
names.Remove("Ana");
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
```

В конец списка добавлены еще два имени. При этом одно имя удалено. Сохраните файл и введите `dotnet run` для тестирования.

List<T> позволяет добавлять ссылки на отдельные элементы по **индексу**. Поместите индекс между токенами `[` и `]` после имени списка. Для первого индекса в C# используется значение 0. Добавьте следующий код сразу после добавленного фрагмента и протестируйте его:

```
Console.WriteLine($"My name is {names[0]}");
Console.WriteLine($"I've added {names[2]} and {names[3]} to the list");
```

Доступ к индексу за пределами списка получить невозможно. Помните, что индексы начинаются с 0, поэтому максимальный допустимый индекс меньше, чем число элементов в списке. Вы можете проверить, как долго в списке используется свойство Count. Добавьте следующий код в конец программы:

```
Console.WriteLine($"The list has {names.Count} people in it");
```

Сохраните файл и еще раз введите `dotnet run`, чтобы просмотреть результаты.

Поиск по спискам и их сортировка

В наших примерах используются сравнительно небольшие списки. Но приложения часто создают списки с гораздо большим количеством элементов, иногда они исчисляются в тысячах. Чтобы найти элементы в таких больших коллекциях, необходимо выполнить поиск различных элементов по списку. Метод IndexOf выполняет поиск элемента и возвращает его индекс. Если элемент отсутствует в списке, `IndexOf` возвращает `-1`. Добавьте этот код в конец программы:

```
var index = names.IndexOf("Felipe");
if (index == -1)
{
    Console.WriteLine($"When an item is not found, IndexOf returns {index}");
}
else
{
    Console.WriteLine($"The name {names[index]} is at index {index}");
}

index = names.IndexOf("Not Found");
if (index == -1)
{
    Console.WriteLine($"When an item is not found, IndexOf returns {index}");
}
else
{
    Console.WriteLine($"The name {names[index]} is at index {index}");

}
```

Кроме того, можно сортировать элементы в списке. Метод Sort сортирует все элементы списка в обычном порядке (строки — в алфавитном). Добавьте этот код в конец программы:

```
names.Sort();
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
```

Сохраните файл и введите `dotnet run`, чтобы протестировать последнюю версию.

Прежде чем перейти к следующему разделу, переместим текущий код в отдельный метод. Это упростит начало работы с новым примером. Поместите весь написанный код в новый метод с именем `WorkWithStrings()`. Вызовите этот метод в начале кода программы. В результате ваш код должен выглядеть примерно следующим образом:

```
using System;
using System.Collections.Generic;

WorkWithString();

void WorkWithString()
{
    var names = new List<string> { "<name>", "Ana", "Felipe" };
    foreach (var name in names)
    {
        Console.WriteLine($"Hello {name.ToUpper()}!");
    }

    Console.WriteLine();
    names.Add("Maria");
    names.Add("Bill");
    names.Remove("Ana");
    foreach (var name in names)
    {
        Console.WriteLine($"Hello {name.ToUpper()}!");
    }

    Console.WriteLine($"My name is {names[0]}");
    Console.WriteLine($"I've added {names[2]} and {names[3]} to the list");

    Console.WriteLine($"The list has {names.Count} people in it");

    var index = names.IndexOf("Felipe");
    if (index == -1)
    {
        Console.WriteLine($"When an item is not found, IndexOf returns {index}");
    }
    else
    {
        Console.WriteLine($"The name {names[index]} is at index {index}");
    }

    index = names.IndexOf("Not Found");
    if (index == -1)
    {
        Console.WriteLine($"When an item is not found, IndexOf returns {index}");
    }
    else
    {
        Console.WriteLine($"The name {names[index]} is at index {index}");

    }

    names.Sort();
    foreach (var name in names)
    {
        Console.WriteLine($"Hello {name.ToUpper()}!");
    }
}
```

Списки других типов

Вы уже использовали в списках тип `string`. Создадим List<T> с использованием другого типа. Сначала создадим набор чисел.

Добавьте следующий код в программу после вызова `WorkWithStrings()`:

```
var fibonacciNumbers = new List<int> {1, 1};
```

Будет создан список целых чисел. Для первых двух целых чисел будет задано значение 1. Это два первых значения *последовательности Фибоначчи*. Каждое следующее число Фибоначчи — это сумма двух предыдущих чисел. Добавьте этот код:

```
var previous = fibonacciNumbers[fibonacciNumbers.Count - 1];
var previous2 = fibonacciNumbers[fibonacciNumbers.Count - 2];

fibonacciNumbers.Add(previous + previous2);

foreach (var item in fibonacciNumbers)
    Console.WriteLine(item);
```

Сохраните файл и введите `dotnet run`, чтобы просмотреть результаты.

> **TIP**
> Для задач этого раздела вы можете закомментировать код, который вызывает `WorkingWithStrings();`. Просто поместите два символа `/` перед вызовом. Например: `// WorkingWithStrings();`.

Задача

Попробуйте объединить некоторые идеи из этого и предыдущих занятий. Расширьте код с числами Фибоначчи, который вы создали. Попробуйте написать код для создания первых 20 чисел в последовательности. Подсказка: 20-е число Фибоначчи — 6765.

Выполнение задачи

Пример решения можно просмотреть в готовом примере кода на GitHub.

При каждой итерации цикла суммируются два последних целых числа в списке. Полученное значение добавляется в список. Цикл повторяется, пока в список не будут добавлены 20 элементов.

Поздравляем! Вы выполнили задачи в руководстве по спискам. Можете продолжить изучение дополнительных руководств в своей среде разработки.

Дополнительные сведения о работе с типом `List` см. в статье Коллекции и структуры данных. Также в нем описаны многие другие типы коллекций.

Общая структура программы C#

18.12.2021 • 2 minutes to read

Программа на языке C# состоит из одного или нескольких файлов. Каждый файл может содержать или не содержать пространства имен. Пространство имен может содержать типы, такие как классы, структуры, интерфейсы, перечисления и делегаты или другие пространства имен. Ниже приведен пример структуры программы на C#, содержащей все эти элементы.

```
// A skeleton of a C# program
using System;

// Your program starts here:
Console.WriteLine("Hello world!");

namespace YourNamespace
{
    class YourClass
    {
    }

    struct YourStruct
    {
    }

    interface IYourInterface
    {
    }

    delegate int YourDelegate();

    enum YourEnum
    {
    }

    namespace YourNestedNamespace
    {
        struct YourStruct
        {
        }
    }
}
```

В предыдущем примере для точки входа программы используются *инструкции верхнего уровня*. Эта функция была добавлена в C# 9. До версии C# 9 точка входа была статическим методом с именем `Main`, как показано в следующем примере:

```
// A skeleton of a C# program
using System;
namespace YourNamespace
{
    class YourClass
    {
    }

    struct YourStruct
    {
    }

    interface IYourInterface
    {
    }

    delegate int YourDelegate();

    enum YourEnum
    {
    }

    namespace YourNestedNamespace
    {
        struct YourStruct
        {
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            //Your program starts here...
            Console.WriteLine("Hello world!");
        }
    }
}
```

Связанные разделы

Сведения об этих элементах программы см. в разделе о типах в руководстве по основам:

- Классы
- Структуры
- Пространства имен
- Интерфейсы
- Перечисления
- Делегаты

Спецификация языка C#

Дополнительные сведения см. в разделе Основные понятия в Спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

Система типов C#

18.12.2021 • 12 minutes to read

C# является строго типизированным языком. Каждая переменная и константа имеет тип, как и каждое выражение, результатом вычисления которого является значение. Каждое объявление метода задает имя, тип и вид (значение, ссылка или вывод) для каждого входного параметра и для возвращаемого значения. В библиотеке классов .NET определены встроенные числовые типы и комплексные типы, представляющие разнообразные конструкции. К ним относятся файловая система, сетевые подключения, коллекции и массивы объектов, а также даты. Обычная программа на C# использует типы из этой библиотеки классов и пользовательские типы, которые моделируют уникальные концепции конкретной сферы применения.

В типах может храниться следующая информация:

- место, необходимое для хранения переменной этого типа;
- максимальное и минимальное значения, которые могут быть представлены;
- содержащиеся в типе члены (методы, поля, события и т. д.);
- базовый тип, от которого наследует этот тип;
- реализуемые им интерфейсы;
- разрешенные виды операций.

Компилятор использует сведения о типах, чтобы проверить, все ли операции, выполняемые в коде, являются *типобезопасными*. Например, при объявлении переменной типа `int` компилятор позволяет в дополнение использовать переменную и операции вычитания. При попытке выполнить эти же операции для переменной типа `bool` компилятор выдаст ошибку, как показано в следующем примере:

```
int a = 5;
int b = a + 2; //OK

bool test = true;

// Error. Operator '+' cannot be applied to operands of type 'int' and 'bool'.
int c = a + test;
```

> **NOTE**
> Тем, кто ранее использовал C и C++, нужно обратить внимание на то, что в C# тип `bool` нельзя преобразовать в `int`.

Компилятор внедряет сведения о типе в исполняемый файл в виде метаданных. Среда CLR использует эти метаданные во время выполнения для дальнейшего обеспечения безопасности типа при выделении и освобождении памяти.

Задание типов в объявлениях переменных

Когда вы объявляете в программе переменную или константу, для нее нужно задать тип либо использовать ключевое слово `var`, чтобы компилятор определил тип самостоятельно. В следующем примере показаны некоторые объявления переменных, использующие встроенные числовые типы и сложные пользовательские типы:

```
// Declaration only:
float temperature;
string name;
MyClass myClass;

// Declaration with initializers (four examples):
char firstLetter = 'C';
var limit = 3;
int[] source = { 0, 1, 2, 3, 4, 5 };
var query = from item in source
            where item <= limit
            select item;
```

Типы параметров и возвращаемых значений метода задаются в объявлении метода. Далее представлена сигнатура метода, который требует значение `int` в качестве входного аргумента и возвращает строку:

```
public string GetName(int ID)
{
    if (ID < names.Length)
        return names[ID];
    else
        return String.Empty;
}
private string[] names = { "Spencer", "Sally", "Doug" };
```

После объявления переменной вы не можете повторно объявить ее с новым типом и назначить ей значение, несовместимое с объявленным типом. Например, нельзя объявить переменную типа `int` и затем присвоить ей логическое значение `true`. Но значения можно преобразовать в другие типы, например при сохранении в других переменных или передаче в качестве аргументов метода. Если *преобразование типов* не приводит к потере данных, оно выполняется компилятором автоматически. Для преобразования, которое может привести к потере данных, необходимо выполнить *приведение* в исходном коде.

Дополнительные сведения см. в разделе Приведение и преобразование типов.

Встроенные типы

C# предоставляет стандартный набор встроенных типов. Они используются для представления целых чисел, значений с плавающей запятой, логических выражений, текстовых символов, десятичных значений и других типов данных. Также существуют встроенные типы `string` и `object`. Такие типы доступны для использования в любой программе C#. Полный список встроенных типов см. в разделе Встроенные типы.

Пользовательские типы

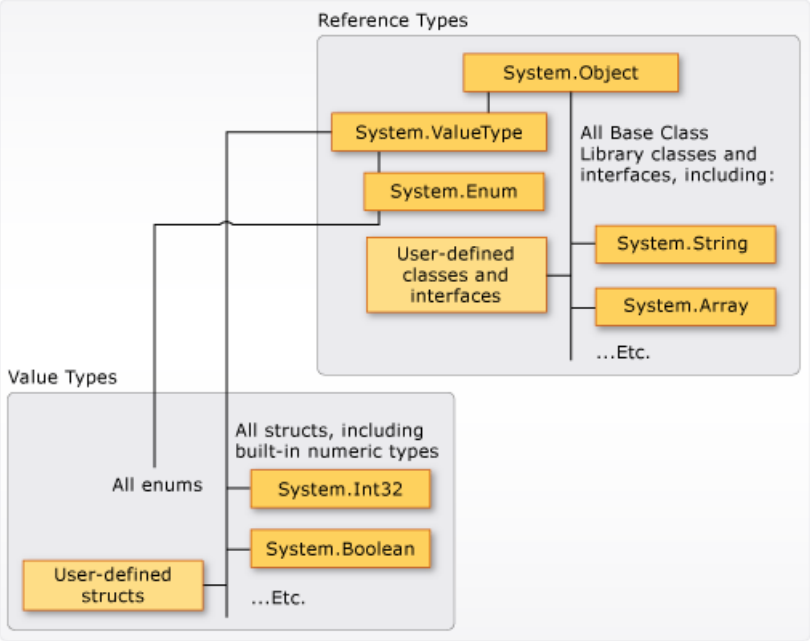
Вы можете создавать собственные типы, используя конструкции `struct`, `class`, `interface`, `enum` и `record`. Сама библиотека классов .NET — это коллекция пользовательских типов, которые вы можете свободно использовать в приложениях. По умолчанию в любой программе C# доступны наиболее часто используемые типы из библиотеки классов. Чтобы сделать доступными другие типы, нужно явным образом добавить в проект ссылку на сборку, которая определяет их. Если компилятору предоставлена ссылка на сборку, то вы можете объявлять в коде переменные (и константы) любых типов, объявленных в этой сборке. См. дополнительные сведения о библиотеке классов .NET.

Система общих типов CTS

Важно понимать две основные вещи, касающиеся системы типов, используемой в .NET:

- Она поддерживает принцип наследования. Типы могут быть производными от других типов, которые называются *базовыми типами*. Производный тип наследует все (с некоторыми ограничениями) методы, свойства и другие члены базового типа. Базовый тип, в свою очередь, может быть производным от какого-то другого типа, при этом производный тип наследует члены обоих базовых типов в иерархии наследования. Все типы, включая встроенные числовые типы, например System.Int32 (ключевое слово C#: `int`), в конечном счете являются производными от одного базового типа System.Object (ключевое слово C#: `object`). Эта унифицированная иерархия типов называется Системой общих типов CTS. Дополнительные сведения о наследовании в C# см. в статье Inheritance (Наследование).
- Каждый тип в CTS определяется как *тип значения* либо *ссылочный тип*. Это справедливо и для всех пользовательских типов, в том числе включенных в библиотеку классов .NET или определенных вами. Если в определении типа используется ключевое слово `struct`, он является типом значения. Например, все встроенные числовые типы определены как `structs`. Если в определении типа используется ключевое слово `class` или `record`, он является ссылочным типом. Для ссылочных типов и типов значений используются разные правила компиляции, и они демонстрируют разное поведение во время выполнения.

Ниже показаны взаимоотношения между типами значения и ссылочными типами в CTS.



> **NOTE**
> Как видно, все наиболее часто используемые типы организованы в пространство имен System. Но само по себе пространство имен, в котором размещен тип, никак не зависит от того, является ли он типом значения или ссылочным типом.

Классы и структуры являются двумя основными конструкциями системы общих типов CTS, используемой на платформе .NET. В C# 9 добавлены записи, которые представляют собой тип класса. Оба они являются структурами данных, которые инкапсулируют набор данных и поведений в одной логической сущности. Данные и поведение являются *членами* класса, структуры или записи. К ним относятся методы, свойства, события и другие элементы, которые описаны далее в этой статье.

Объявление класса, структуры или записи представляет собой своего рода чертеж, на основе которого создаются экземпляры или объекты во время выполнения. Если вы определите класс, структуру или запись с именем `Person`, то `Person` здесь обозначает имя типа. Если вы объявите и инициализируете переменную `p` типа `Person`, принято говорить, что `p` является объектом (или экземпляром) `Person`.

Можно создать несколько экземпляров одного типа `Person`, и каждый экземпляр будет иметь разные значения свойств и полей.

Класс является ссылочным типом. Когда вы создаете объект типа и назначаете его переменной, эта переменная содержит только ссылку на память объекта. Если ссылка на объект сохраняется в новую переменную, эта переменная также ссылается на исходный объект. Изменения, внесенные через одну переменную, отражаются и в другой переменной, поскольку обе они ссылаются на одни и те же данные.

Структура (struct) является типом значения. При создании структуры переменная, которой присвоена структура, содержит фактические данные этой структуры. Если структура присваивается новой переменной, все данные копируются. Таким образом, новая переменная и исходная переменная содержат две отдельные копии одинаковых данных. Изменения, внесенные в одну копию, не влияют на другую.

Типы записей могут быть либо ссылочными типами (`record class`), либо типами значений (`record struct`).

Как правило, классы используются для моделирования более сложного поведения. Классы обычно хранят данные, которые должны быть изменены после создания объекта класса. Структуры лучше всего подходят для небольших структур данных. Структуры обычно хранят данные, которые не должны изменяться после создания структуры. Типы записей — это структуры данных с дополнительными членами, синтезированными компилятором. Записи обычно хранят данные, которые не должны изменяться после создания объекта.

Типы значений

Типы значений являются производными от System.ValueType, который является производным от System.Object. Типы, производные от System.ValueType, имеют особое поведение в среде CLR. Переменные типа значения непосредственно содержат их значения. Память для структуры выделяется встроенным образом в любом контексте, объявленном переменной. Для переменных типа значения не предусмотрены раздельное размещение в куче или накладные расходы при сборке мусора. Можно объявить типы `record struct`, которые являются типами значений, и включить синтезированные члены для записей.

Существует две категории типов значений: `struct` и `enum`.

Встроенные числовые типы являются структурами и имеют поля и методы, к которым можно обращаться.

```
// constant field on type byte.
byte b = byte.MaxValue;
```

Но объявление и присвоение значений вы выполняете для них так, как если бы они были простыми нестатистическими типами:

```
byte num = 0xA;
int i = 5;
char c = 'Z';
```

Типы значений являются *запечатанными*. Тип не может быть производным от любого типа значений, например System.Int32. Вы не можете определить структуру для наследования из любого определенного пользователем класса или структуры, так как структура может наследовать только от System.ValueType. Тем не менее структура может реализовывать один или несколько интерфейсов. Можно выполнить приведение типа структуры к любому типу интерфейса, который он реализует. Это приведет к операции *упаковки-преобразования*, которая создаст программу-оболочку для структуры внутри объекта ссылочного типа в управляемой куче. Операции упаковки-преобразования выполняются при передаче типа значения в метод, принимающий System.Object или любой тип интерфейса в качестве входного

параметра. Дополнительные сведения см. в разделе Упаковка-преобразование и распаковка-преобразование.

Используйте ключевое слово struct, чтобы создать собственные пользовательские типы значений. Как правило, структура используется как контейнер для небольшого набора связанных переменных, как показано в следующем примере:

```
public struct Coords
{
    public int x, y;

    public Coords(int p1, int p2)
    {
        x = p1;
        y = p2;
    }
}
```

См. сведения в описании типов структур. См. дополнительные сведения о типах значений.

Еще одна категория типов значений — это `enum`. Перечисление определяет набор именованных целочисленных констант. Например, перечисление System.IO.FileMode из библиотеки классов .NET содержит набор именованных целочисленных констант, которые определяют правила открытия файла. В следующем примере представлено определение этого типа:

```
public enum FileMode
{
    CreateNew = 1,
    Create = 2,
    Open = 3,
    OpenOrCreate = 4,
    Truncate = 5,
    Append = 6,
}
```

Константа System.IO.FileMode.Create имеет значение 2. Так как имена намного лучше воспринимаются человеком при изучении исходного кода, мы рекомендуем всегда использовать перечисления вместо литеральных числовых констант. Для получения дополнительной информации см. System.IO.FileMode.

Все перечисления наследуют от System.Enum, который наследует от System.ValueType. К перечислениям применимы все те же правила, что к структурам. Дополнительные сведения о перечислениях см. в разделе Типы перечислений.

Ссылочные типы

Тип, который определен как `class`, `record`, `delegate`, массив или `interface`, является `reference type`.

Объявляемая переменная с типом `reference type` будет содержать значение `null` до тех пор, пока вы не назначите ей экземпляр такого типа или не создадите его с помощью оператора `new`. Создание и назначение класса демонстрируется в следующем примере:

```
MyClass myClass = new MyClass();
MyClass myClass2 = myClass;
```

Вы не можете создать экземпляр `interface` напрямую с помощью оператора `new`. Вместо этого создайте и назначьте экземпляр класса, который реализует интерфейс. Рассмотрим следующий пример.

```
MyClass myClass = new MyClass();

// Declare and assign using an existing value.
IMyInterface myInterface = myClass;

// Or create and assign a value in a single statement.
IMyInterface myInterface2 = new MyClass();
```

При создании объекта выделяется память в управляемой куче, и переменная хранит только ссылку на расположение объекта. Хранение типов в управляемой куче требует дополнительных действий как при выделении памяти, так и при удалении, которое выполняется функцией автоматического управления памятью в среде CLR, известной как *сборка мусора*. Сборка мусора является хорошо оптимизированным процессом и в большинстве случаев не ухудшает производительность. Дополнительные сведения о сборке мусора см. в статье Автоматическое управление памятью.

Все массивы являются ссылочными типами, даже если их элементы являются типами значений. Массивы являются неявно производными от класса System.Array, но в C# их можно объявлять и использовать с упрощенным синтаксисом, как показано в следующем примере:

```
// Declare and initialize an array of integers.
int[] nums = { 1, 2, 3, 4, 5 };

// Access an instance property of System.Array.
int len = nums.Length;
```

Ссылочные типы полностью поддерживают наследование. Создаваемый класс может наследовать от любого другого интерфейса или класса, который не определен как запечатанный, а другие классы могут наследовать от этого класса и переопределять его виртуальные методы. Дополнительные сведения о создании собственных классов см. в статье Классы, структуры и записи. Дополнительные сведения о наследовании в C# и виртуальных методах см. в статье Inheritance (Наследование).

Типы литеральных значений

В C# литеральные значения получают тип от компилятора. Вы можете указать способ типизации числового литерала, добавив букву в конце числа. Например, чтобы значение `4.56` рассматривалось как значение `float`, добавьте после этого числа букву "f" или "F": `4.56f`. Если буква отсутствует, компилятор самостоятельно выберет тип для литерала. Дополнительные сведения о том, какие типы могут быть указаны с буквенными суффиксами, см. в разделах Целочисленные числовые типы и Числовые типы с плавающей запятой.

Так как литералы являются типизированными и все типы в конечном счете являются производными от System.Object, можно написать и скомпилировать следующий код:

```
string s = "The answer is " + 5.ToString();
// Outputs: "The answer is 5"
Console.WriteLine(s);

Type type = 12345.GetType();
// Outputs: "System.Int32"
Console.WriteLine(type);
```

Универсальные типы

Тип может быть объявлен с одним или несколькими *параметрами типа* вместо фактического типа (*конкретного типа*), который клиентский код предоставит при создании экземпляра этого типа. Такие

типы называются *универсальными типами*. Например, тип .NET System.Collections.Generic.List<T> имеет один параметр типа, которому в соответствии с соглашением присвоено имя `T`. При создании экземпляра этого типа необходимо указать тип объектов, которые будут содержаться в списке, например `string`:

```
List<string> stringList = new List<string>();
stringList.Add("String example");
// compile time error adding a type other than a string:
stringList.Add(4);
```

Использование параметра типа позволяет повторно использовать один же класс для хранения элементов любого типа, не преобразовывая каждый элемент в объект. Универсальные классы коллекций называются *строго типизированными коллекциями*, так как компилятору известен конкретный тип элементов коллекции и он может выдать ошибку во время компиляции, если, к примеру, вы попытаетесь добавить целое число в объект `stringList`, созданный в предыдущем примере. Дополнительные сведения см. в статье Универсальные шаблоны.

Неявные типы, анонимные типы и типы, допускающие значение NULL

Вы можете неявно типизировать локальную переменную (но не элементы класса) с помощью ключевого слова `var`. Такая переменная получает конкретный тип во время компиляции, но этот тип предоставляется компилятором. Дополнительные сведения см. в статье Implicitly Typed Local Variables (Неявно типизированные локальные переменные).

Иногда нет смысла создавать именованный тип для простых наборов связанных значений, которые не будут сохраняться или передаваться за пределами метода. В таких случаях можно применить *анонимные типы*. Дополнительные сведения см. в статье Анонимные типы.

Обычные типы значений не могут иметь значение `null`. Но вы можете создать специальные *типы, допускающие значения NULL*, добавив символ `?` после имени типа. Например, тип `int?` является типом `int`, который может иметь значение `null`. Типы, допускающие значение NULL, представляют собой экземпляры универсального типа структуры System.Nullable<T>. Типы, допускающие значение NULL, особенно полезны при передаче данных в базы данных, где могут использоваться числовые значения `null`, и из таких баз данных. Дополнительные сведения см. в разделе Типы, допускающие значение NULL.

Тип времени компиляции и тип времени выполнения

Переменная может иметь разные типы времени компиляции и времени выполнения. *Тип времени компиляции* является объявленным или выведенным типом переменной в исходном коде. *Тип времени выполнения* является типом экземпляра, на который ссылается такая переменная. Часто эти два типа совпадают:

```
string message = "This is a string of characters";
```

В других случаях тип времени компиляции отличается, как показано в следующих двух примерах:

```
object anotherMessage = "This is another string of characters";
IEnumerable<char> someCharacters = "abcdefghijklmnopqrstuvwxyz";
```

В обоих предыдущих примерах используется тип времени выполнения `string`. Тип времени компиляции

в первой строке — `object`, а во второй — `IEnumerable<char>`.

Если два типа переменной отличаются, важно понимать, когда применяются типы времени компиляции и времени выполнения. Тип времени компиляции определяет все действия, выполняемые компилятором. Такие действия компилятора включают разрешение вызовов методов, разрешение перегрузки, а также доступные неявные и явные приведения. Тип времени выполнения определяет все действия, разрешаемые при выполнении. Такие действия времени выполнения включают отправку вызовов виртуальных методов, оценку выражений `is` и `switch`, а также другие API тестирования типов. Чтобы лучше понять, как ваш код взаимодействует с типами, определите, какие действия применяются к каждому типу.

Связанные разделы

Дополнительные сведения см. в следующих статьях:

- Встроенные типы
- Типы значений
- Ссылочные типы

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

Объектно-ориентированное программирование

18.12.2021 • 3 minutes to read

Инкапсуляция

Концепцию *инкапсуляции* признают одним из основополагающих принципов объектно-ориентированного программирования. В классе или структуре можно указать уровень доступности для обращения к каждому из его членов из кода, расположенного вне этого класса или структуры. Вы можете скрыть методы и переменные, которые не предназначены для использования вне класса или сборки. Это позволяет снизить риск ошибок в коде и вредоносных действий. Дополнительные сведения см. в разделе Объектно-ориентированное программирование.

Участники

Члены типа включают все методы, поля, константы, свойства и события. В C# не существует глобальных переменных или методов, как в некоторых других языках. Даже точка входа программы, то есть метод `Main`, должна быть объявлена внутри класса или структуры (для инструкций верхнего уровня — неявным образом).

Ниже приведен полный список возможных элементов, которые можно объявить в классе, структуре или записи.

- Поля
- Константы
- Свойства
- Методы
- Конструкторы
- События
- Методы завершения
- Индексаторы
- Операторы
- Вложенные типы

Специальные возможности

Некоторые методы и свойства специально предназначены для того, чтобы их вызов или доступ к ним осуществлялся из *клиентского кода*, то есть из кода за пределами этого класса или структуры. Другие методы и свойства могут использоваться только в самом классе или структуре. Важно ограничить доступность кода так, чтобы только нужные элементы клиентского кода получали к нему доступ. Уровень доступности для типов и их элементов вы можете задать с помощью следующих модификаторов доступа.

- public
- protected
- internal
- protected internal
- private
- private protected.

По умолчанию используется режим доступа `private`.

Наследование

Классы (но не структуры) поддерживают наследование. Класс, производный от другого класса, называемого *базовым классом*, автоматически включает все открытые, защищенные и внутренние члены базового класса за исключением конструкторов и методов завершения. Дополнительные сведения см. в статьях о наследовании и полиморфизме.

Классы могут быть объявлены как абстрактные. Это означает, что один или несколько их членов не имеют реализации. Из абстрактных классов нельзя напрямую создать экземпляры. Они выполняют роль базовых классов для других классов, которые предоставляют реализацию недостающих членов. Также классы можно объявить запечатанными, чтобы запретить наследование от них других классов.

Интерфейсы

Классы, структуры и записи могут реализовывать несколько интерфейсов. Реализация из интерфейса означает, что тип реализует все методы, определенные в интерфейсе. Дополнительные сведения см. в статье Интерфейсы.

Универсальные типы

Для класса, структуры или записи можно определить один параметр типа или несколько. Клиентский код назначает тип при создании экземпляра типа. Например, класс List<T> в пространстве имен System.Collections.Generic определен с помощью одного параметра типа. Клиентский код создает экземпляр `List<string>` или `List<int>`, информируя о том, сущности какого типа будут храниться в этом списке. Дополнительные сведения см. в статье Универсальные шаблоны.

Статические типы

Классы (но не структуры или записи) могут быть объявлены как `static`. Статический класс может содержать только статические элементы. Для него нельзя создать экземпляр с помощью ключевого слова `new`. При запуске программы в память загружается одна копия такого класса. Доступ к его членам осуществляется через имя класса. Классы, структуры и записи могут содержать статические элементы.

Вложенные типы

Класс, структура или запись могут быть вложены в другой класс, структуру или запись.

Разделяемые типы

Вы можете разделить определение класса, структуры или метода на несколько файлов с кодом.

Инициализаторы объектов

Вы можете создавать и инициализировать объекты классов или структур, а также коллекции объектов, назначая значения связанным свойствам.

Анонимные типы

В некоторых ситуациях нет смысла или пользы создавать именованный класс. В таких случаях можно использовать анонимные типы. Анонимные типы определяются их именованными элементами данных.

Методы расширения

Вы можете "расширить" класс, даже не создавая производного класса. Для этого можно создать отдельный тип и вызывать его методы так, как если бы они принадлежали исходному типу.

Неявно типизированные локальные переменные

Внутри метода класса или структуры можно использовать неявное типизирование, чтобы компилятор самостоятельно определял тип переменной во время компиляции.

Записи

В C# 9 появился тип `record`, ссылочный тип, который можно создать вместо класса или структуры. Записи — это классы со встроенным поведением для инкапсуляции данных в неизменяемых типах. В C# 10 представлен новый тип значения `record struct`. Запись (`record class` или `record struct`) предоставляет следующие возможности:

- краткий синтаксис для создания ссылочного типа с неизменяемыми свойствами;
- равенство значений. Две переменные типа записи равны, если у них одинаковые типы и равные значения в каждом поле обеих записей. Классы используют равенство ссылок: две переменные типа класса равны, если они ссылаются на один и тот же объект.
- Краткий синтаксис для обратимого изменения. Выражение `with` позволяет создать новый экземпляр записи, который является копией существующего экземпляра, но в котором изменены значения указанных свойств.
- Встроенное форматирование для отображения. Метод `ToString` выводит имя типа записи, а также имена и значения общих свойств.
- Поддержка иерархий наследования в классах записей. Классы записей поддерживают наследование. Структуры записей не поддерживают наследование.

Дополнительные сведения см. в статье Записи.

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

Исключения и обработка исключений

18.12.2021 • 2 minutes to read

Функции обработки исключений в языке C# помогают вам справиться с непредвиденными или исключительными проблемами, которые возникают при выполнении программы. При обработке исключений используются ключевые слова `try`, `catch` и `finally` для действий, которые могут оказаться неудачными. Это позволяет обрабатывать ошибки так, как кажется разумным, а также правильно высвобождать ресурсы. Исключения могут создаваться средой выполнения (CLR), платформой .NET , библиотеками сторонних поставщиков или кодом самого приложения. Чтобы создать исключение, используйте ключевое слово `throw`.

Во многих случаях исключение может создаваться не тем методом, который вызывается в вашем коде, а одним из последующих методов в стеке вызовов. Если создается такое исключение, среда CLR разворачивает стек, находит метод с блоком `catch` для исключений соответствующего типа и выполняет первый такой обнаруженный блок `catch`. Если подходящий блок `catch` не будет обнаружен во всем стеке вызовов, среда CLR завершает процесс и выводит сообщение для пользователя.

В этом примере метод выполняет проверку деления на нуль и перехватывает ошибку. Если не использовать обработку исключений, такая программа завершит работу с ошибкой **DivideByZeroException was unhandled** (Исключение DivideByZero не обработано).

```
public class ExceptionTest
{
    static double SafeDivision(double x, double y)
    {
        if (y == 0)
            throw new DivideByZeroException();
        return x / y;
    }

    public static void Main()
    {
        // Input for test purposes. Change the values to see
        // exception handling behavior.
        double a = 98, b = 0;
        double result;

        try
        {
            result = SafeDivision(a, b);
            Console.WriteLine("{0} divided by {1} = {2}", a, b, result);
        }
        catch (DivideByZeroException)
        {
            Console.WriteLine("Attempted divide by zero.");
        }
    }
}
```

Общие сведения об исключениях

Исключения имеют следующие свойства.

- Исключения представляют собой типы, производные в конечном счете от `System.Exception`.
- Используйте блок `try` для выполнения таких инструкций, которые могут создавать исключения.

- Когда внутри такого блока `try` возникает исключение, поток управления переходит к первому подходящему обработчику исключений в стеке вызовов. В C# ключевое слово `catch` обозначает обработчик исключений.
- Если для созданного исключения не существует обработчиков, выполнение программы прекращается с сообщением об ошибке.
- Не перехватывайте исключение, если вы не намерены его обрабатывать с сохранением известного состояния приложения. Если вы перехватываете `System.Exception`, создайте его заново в конце блока `catch`, используя ключевое слово `throw`.
- Если блок `catch` определяет переменную исключения, ее можно использовать для получения дополнительных сведений о типе созданного исключения.
- Программа может явным образом создавать исключения с помощью ключевого слова `throw`.
- Объекты исключения содержат подробные сведения об ошибке, например состояние стека вызовов и текстовое описание ошибки.
- Код в блоке `finally` выполняется, даже если создано исключение. Используйте блок `finally`, чтобы высвободить ресурсы, например закрыть потоки и файлы, которые были открыты внутри блока `try`.
- Управляемые исключения реализованы в платформе .NET на основе структурированного механизма обработки исключений Win32. Дополнительные сведения см. в статьях Structured Exception Handling (C/C++) (Структурированная обработка исключений в C и C++) и A Crash Course on the Depths of Win32 Structured Exception Handling (Интенсивное погружение в структурированную обработку исключений на платформе Win32).

Спецификация языка C#

Дополнительные сведения см. в разделе Исключения в Спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- SystemException
- Ключевые слова в C#
- throw
- try-catch
- try-finally
- try-catch-finally
- Исключения

Новые возможности C# 10

18.12.2021 • 6 minutes to read

В C# 10 добавлены следующие функции и улучшения языка C#:

- Структуры записей
- Улучшения типов структуры
- Обработчики интерполированных строк
- Директивы `global using`
- Объявление пространства имен в пределах файла
- Расширенные шаблоны свойств
- Усовершенствования в лямбда-выражениях
- Разрешены интерполированные строки `const`
- Типы записей могут запечатывать `ToString()`
- Улучшенное определенное назначение
- Разрешено присваивание и объявление в одном и том же деконструировании
- Разрешен атрибут `AsyncMethodBuilder` в методах
- Атрибут CallerArgumentExpression
- Улучшенная прагма `#line`

Дополнительные функции доступны в режиме *предварительной версии*. Опробуйте эти функции и отправьте о них отзыв. До окончательного выпуска в них могут быть внесены изменения. Для использования этих функций в проекте необходимо задать `<LangVersion>` значение `Preview`. Дополнительные сведения об универсальных атрибутах см. далее в этой статье.

C# 10 поддерживается в .NET 6. Дополнительные сведения см. в статье Управление версиями языка C#.

Вы можете скачать последний пакет SDK для .NET 6 на странице скачиваемых файлов .NET. Вы также можете скачать версию Visual Studio 2022, которая включает пакет SDK для .NET 6.

Структуры записей

Вы можете объявлять записи типа значения с помощью объявления `record struct``readonly record struct` или . Теперь путем объявления `record class` вы можете уточнить, что `record` имеет ссылочный тип.

Улучшения типов структуры

В C# 10 реализованы следующие улучшения, связанные с типами структур:

- Вы можете объявить конструктор без параметров экземпляра в типе структуры и инициализировать поле экземпляра или свойство в качестве объявления. Дополнительные сведения см. в разделе Конструкторы без параметров и инициализаторы полей статьи Типы структур.
- Левый операнд выражения `with` может иметь любой тип структуры или анонимный (ссылочный) тип.

Обработчик интерполированных строк

Можно создать тип, который создает результирующую строку на основе выражения интерполированной строки. Библиотеки .NET используют эту функцию во многих интерфейсах API. Вы можете создать такой обработчик, следуя инструкциям из этого руководства.

Глобальные директивы using

Можно добавить модификатор `global` в любую директиву using, чтобы указать компилятору, что эта директива применяется ко всем исходным файлам в компиляции. Обычно это все исходные файлы в проекте.

Объявление пространства имен в пределах файла

Можно использовать новую форму объявления `namespace`, чтобы объявить, что все последующие объявления являются членами объявленного пространства имен:

```
namespace MyNamespace;
```

Этот новый синтаксис сохраняет как горизонтальное, так и вертикальное пространство для объявлений `namespace`.

Расширенные шаблоны свойств

Начиная с C# 10 можно ссылаться на вложенные свойства или поля в шаблоне свойства. Например, шаблон формы

```
{ Prop1.Prop2: pattern }
```

допустим в C# 10 и более поздних версиях и эквивалентен

```
{ Prop1: { Prop2: pattern } }
```

допустим в C# 8.0 и более поздних версиях.

Дополнительные сведения см. в примечании к предлагаемой функции Расширенные шаблоны свойств. Дополнительные сведения о шаблоне свойства см. в разделе Шаблон свойства статьи Шаблоны.

Улучшения лямбда-выражений

В C# 10 включено множество улучшений обработки лямбда-выражений:

- Лямбда-выражения могут иметь естественный тип, где компилятор может вывести тип делегата из лямбда-выражения или группы методов.
- Лямбда-выражения могут объявлять тип возвращаемого значения, если компилятор не может его вывести.
- К лямбда-выражениям могут применяться атрибуты.

Эти функции делают лямбда-выражения более похожими на методы и локальные функции. Они упрощают использование лямбда-выражений без объявления переменной типа делегата и более тесно работают с новыми минимальными API ASP.NET Core.

Константные интерполированные строки

В C# 10 строки `const` могут быть инициализированы с помощью интерполяции строк, если все заполнители являются константными строками. Интерполяция строк упрощает чтение константных строк при их создании в приложении. Выражения заполнителей не могут быть числовыми константами, так как эти константы преобразуются в строки во время выполнения. Текущий язык и региональные параметры могут влиять на строковое представление. Дополнительные сведения см. в справочнике по

языку для выражений `const`.

Типы записей могут запечатывать ToString

В C# 10 можно добавить модификатор `sealed` при переопределении `ToString` в типе записи. Запечатывание метода `ToString` предотвращает синтезирование компилятором метода `ToString` для любых производных типов записей. Использование `ToString` с `sealed` гарантирует, что все производные типы записей используют метод `ToString`, определенный в общем базовом типе записи. Дополнительные сведения об этой функции см. в статье о записях.

Присваивание и объявление в одном и том же деконструировании

Это изменение снимает ограничение, существовавшее в предыдущих версиях C#. Ранее деконструирование могло присвоить все значения существующим переменным или инициализировать только что объявленные переменные:

```
// Initialization:
(int x, int y) = point;

// assignment:
int x1 = 0;
int y1 = 0;
(x1, y1) = point;
```

C# 10 устраняет это ограничение:

```
int x = 0;
(x, int y) = point;
```

Улучшенное определенное назначение

До C# 10 существовало много сценариев, в которых при определенном назначении и анализе состояния NULL возникали ложноположительные предупреждения. Обычно они были связаны со сравнениями с логическими константами, доступом к переменной только в операторах `true` или `false` в операторе `if` и выражениях объединения со значением NULL. Эти примеры создавали предупреждения в предыдущих версиях C#, но не в C# 10:

```
string representation = "N/A";
if ((c != null && c.GetDependentValue(out object obj)) == true)
{
   representation = obj.ToString(); // undesired error
}

// Or, using ?.
if (c?.GetDependentValue(out object obj) == true)
{
   representation = obj.ToString(); // undesired error
}

// Or, using ??
if (c?.GetDependentValue(out object obj) ?? false)
{
   representation = obj.ToString(); // undesired error
}
```

Основное преимущество этого улучшения состоит в том, что предупреждения для определенного назначения и анализа состояния NULL теперь являются более точными.

Разрешен атрибут AsyncMethodBuilder в методах

В C# 10 и последующих версиях можно указать другой построитель асинхронных методов для одного метода в дополнение к указанию типа построителя методов для всех методов, возвращающих данный тип, аналогичный задаче. Построитель пользовательских асинхронных методов предоставляет больше возможностей для настройки производительности в сценариях, когда определенный метод может выиграть от настраиваемого построителя.

Дополнительные сведения см. в разделе `AsyncMethodBuilder` статьи об атрибутах, считываемых компилятором.

Диагностика атрибута CallerArgumentExpression

Можно использовать System.Runtime.CompilerServices.CallerArgumentExpressionAttribute, чтобы указать параметр, который компилятор заменяет текстовым представлением другого аргумента. Эта функция позволяет библиотекам создавать более конкретные диагностические сведения. Следующий код проверяет условие. Если условие имеет значение false, сообщение об исключении содержит текстовое представление аргумента, переданного в `condition`:

```
public static void Validate(bool condition, [CallerArgumentExpression("condition")] string? message=null)
{
    if (!condition)
    {
        throw new InvalidOperationException($"Argument failed validation: <{message}>");
    }
}
```

Дополнительные сведения об этой функции см. в статье об атрибутах сведений о вызывающем объекте в разделе справочника по языку.

Улучшенная прагма #line

C# 10 поддерживает новый формат для прагмы `#line`. Скорее всего, вы не будете использовать новый формат, но заметите его влияние. Улучшения обеспечивают более детализированные выходные данные в предметно-ориентированных языках (DSL), например Razor. Подсистема Razor использует эти усовершенствования для улучшения процесса отладки. Отладчики смогут точнее выделять источник Razor. Дополнительные сведения о новом синтаксисе см. в статье о директивах препроцессора в справочнике по языку. Также можно прочитать спецификацию функции, чтобы найти примеры для Razor.

Универсальные атрибуты

> **IMPORTANT**
> *Универсальные атрибуты* являются предварительной версией функций. Для ее включения необходимо задать `<LangVersion>` значение `Preview`. До окончательного выпуска в эту функцию могут быть внесены изменения.

Можно объявить универсальный класс, базовым классом которого является System.Attribute. При этом вы получите более удобный синтаксис для атрибутов, требующих параметра System.Type. Ранее вам пришлось бы создать атрибут, который принимает `Type` качестве параметра конструктора:

```
public class TypeAttribute : Attribute
{
   public TypeAttribute(Type t) => ParamType = t;

   public Type ParamType { get; }
}
```

А для применения атрибута следовало бы использовать оператор `typeof`:

```
[TypeAttribute(typeof(string))]
public string Method() => default;
```

С помощью этой новой функции можно создать универсальный атрибут:

```
public class GenericAttribute<T> : Attribute { }
```

Затем укажите параметр типа для использования атрибута:

```
[GenericAttribute<string>()]
public string Method() => default;
```

И вы можете применить полностью закрытый сконструированный универсальный атрибут. Иными словами, необходимо указать все параметры типа. Например, следующее выражение недопустимо:

```
public class GenericType<T>
{
   [GenericAttribute<T>()] // Not allowed! generic attributes must be fully closed types.
   public string Method() => default;
}
```

Аргументы типа должны соответствовать тем же ограничениям, что и оператор `typeof`. Типы, которым требуются аннотации метаданных, запрещены. Вот несколько примеров.

- `dynamic`
- `nint`, `nuint`
- `string?` (или любой ссылочный тип, допускающий значение NULL)
- `(int X, int Y)` (или любые другие типы кортежей, использующие синтаксис кортежей C#).

Эти типы не представлены непосредственно в метаданных. Они содержат заметки, описывающие тип. Во всех случаях можно использовать базовый тип:

- `object` для `dynamic`.
- IntPtr вместо `nint` или `unint`.
- `string` вместо `string?`.
- `ValueTuple<int, int>` вместо `(int X, int Y)`.

Новые возможности C# 9.0

18.12.2021 • 16 minutes to read

В C# 9.0 добавлены следующие функции и улучшения языка C#.

- Записи
- Методы задания только инициализации
- Инструкции верхнего уровня
- Улучшения сопоставления шаблонов
- Производительность и взаимодействие
  - Целые числа собственного размера
  - Указатели функций
  - Отмена создания флага localsinit
- Функции подбора и завершения
  - Выражения `new` с целевым типом
  - Анонимные функции `static`
  - Условное выражение с целевым типом
  - Ковариантные возвращаемые типы
  - Поддержка расширения `GetEnumerator` для циклов `foreach`
  - Параметры удаления лямбда-выражения
  - Атрибуты локальных функций
- Поддержка генераторов кода
  - Инициализаторы модулей
  - Новые функции для разделяемых методов

C# 9.0 поддерживается в **.NET 5**. Дополнительные сведения см. в статье Управление версиями языка C#.

Вы можете скачать последний пакет SDK для .NET на странице скачиваемых файлов .NET.

Типы записей

В C# 9.0 появились ***типы записей***. Вы можете использовать ключевое слово `record` для определения ссылочного типа, который предоставляет встроенные возможности для инкапсуляции данных. Вы можете создавать типы записей с неизменяемыми свойствами, используя позиционные параметры или стандартный синтаксис свойств:

```
public record Person(string FirstName, string LastName);
```

```
public record Person
{
    public string FirstName { get; init; } = default!;
    public string LastName { get; init; } = default!;
};
```

Кроме того, можно создавать типы записей с изменяемыми свойствами и полями:

```
public record Person
{
    public string FirstName { get; set; } = default!;
    public string LastName { get; set; } = default!;
};
```

Несмотря на поддержку изменения, записи предназначены в первую очередь для неизменяемых моделей данных. Тип записи предоставляет следующие возможности:

- Краткий синтаксис для создания ссылочного типа с неизменяемыми свойствами.
- Поведение, полезное для ссылочного типа, ориентированного на данные:
  - Равенство значений
  - Краткий синтаксис для обратимого изменения.
  - Встроенное форматирование для отображения.
- Поддержка иерархий наследования.

На основе типов структур можно создать типы, ориентированные на данные, которые поддерживают равенство значений и почти не определяют поведение. Но в сравнительно больших моделях данных типы структур имеют определенные недостатки:

- Они не поддерживают наследование.
- Они менее эффективны при определении равенства значений. Для типов значений метод ValueType.Equals использует отражение для поиска всех полей. Для записей компилятор создает метод `Equals`. На практике реализация равенства значений в записях работает заметно быстрее.
- В некоторых сценариях они используют больше памяти, так как каждый экземпляр содержит полную копию всех данных. Типы записей являются ссылочными типами, то есть каждый экземпляр записи содержит только ссылку на данные.

Позиционный синтаксис для определения свойств

Позиционные параметры позволяют объявить свойства записи и инициализировать значения свойств при создании экземпляра:

```
public record Person(string FirstName, string LastName);

public static void Main()
{
    Person person = new("Nancy", "Davolio");
    Console.WriteLine(person);
    // output: Person { FirstName = Nancy, LastName = Davolio }
}
```

При использовании позиционного синтаксиса для определения свойства компилятор создает следующие элементы:

- Открытое автоматически реализуемое свойство "только init" создается для каждого позиционного параметра, предоставленного в объявлении записи. Свойство только init может быть задано только в конструкторе или с помощью инициализатора свойств.
- Основной конструктор, параметры которого соответствуют позиционным параметрам в объявлении записи.
- Метод `Deconstruct` с параметром `out` создается для каждого позиционного параметра, предоставленного в объявлении записи.

Дополнительные сведения см. в разделе Позиционный синтаксис статьи, посвященной записям, в справочнике по языку C#.

Неизменяемость

Тип записи не обязательно является неизменяемым. Вы можете объявить свойства с методами доступа `set` и полями без атрибута `readonly`. Но несмотря на поддержку изменения, записи лучше всего подходят для создания неизменяемых моделей данных. Свойства, создаваемые с использованием позиционного синтаксиса, являются неизменяемыми.

Неизменяемость может быть полезной, если требуется обеспечить потокобезопасность типа, ориентированного на данные, или когда существует необходимость сохранять хэш-код в неизменном виде в хэш-таблице. Это может препятствовать возникновению ошибок, когда, например, вы передаете в метод аргумент по ссылке, и этот метод неожиданно изменяет значение аргумента.

Возможности, уникальные для типов записей, реализуются синтезированными компилятором методами, ни один из которых не нарушает неизменяемость путем изменения состояния объекта.

Равенство значений

Равенство значений означает, что две переменные типа записи считаются равными, если совпадают типы и значения всех свойств и полей. Для других ссылочных типов равенство означает полную идентичность. То есть две переменные ссылочного типа равны только в том случае, если они ссылаются на один и тот же объект.

Следующий пример демонстрирует равенство значений для типов записей:

```
public record Person(string FirstName, string LastName, string[] PhoneNumbers);

public static void Main()
{
    var phoneNumbers = new string[2];
    Person person1 = new("Nancy", "Davolio", phoneNumbers);
    Person person2 = new("Nancy", "Davolio", phoneNumbers);
    Console.WriteLine(person1 == person2); // output: True

    person1.PhoneNumbers[0] = "555-1234";
    Console.WriteLine(person1 == person2); // output: True

    Console.WriteLine(ReferenceEquals(person1, person2)); // output: False
}
```

В типах `class` можно вручную переопределить методы и операторы равенства, чтобы реализовать равенство значений, но разработка и тестирование такого кода занимают много времени и могут стать причиной ошибок. Если такая функциональность встроена изначально, это позволяет избежать ошибок, которые возникнут, когда вы забудете обновить пользовательский код переопределения при добавлении или изменении свойств или полей.

Дополнительные сведения см. в разделе Равенство значений статьи, посвященной записям, в справочнике по языку C#.

Обратимое изменение

Если нужно изменить неизменяемые свойства экземпляра записи, вы можете с помощью выражения `with` выполнить *обратимое изменение*. Выражение `with` создает новый экземпляр записи, который является копией существующего экземпляра записи, и изменяет в этой копии указанные свойства и поля. Для указания требуемых изменений используется синтаксис инициализатора объектов, как показано в следующем примере:

```
public record Person(string FirstName, string LastName)
{
    public string[] PhoneNumbers { get; init; }
}

public static void Main()
{
    Person person1 = new("Nancy", "Davolio") { PhoneNumbers = new string[1] };
    Console.WriteLine(person1);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers = System.String[] }

    Person person2 = person1 with { FirstName = "John" };
    Console.WriteLine(person2);
    // output: Person { FirstName = John, LastName = Davolio, PhoneNumbers = System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { PhoneNumbers = new string[1] };
    Console.WriteLine(person2);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers = System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { };
    Console.WriteLine(person1 == person2); // output: True
}
```

Дополнительные сведения см. в разделе Обратимое изменение статьи, посвященной записям, в справочнике по языку C#.

Встроенное форматирование для отображения

Типы записей имеют создаваемый компилятором метод ToString, который отображает имена и значения открытых свойств и полей. Метод `ToString` возвращает строку в следующем формате:

```
<record type name> { <property name> = <value>, <property name> = <value>, ...}
```

Для ссылочных типов вместо значения свойства отображается имя типа того объекта, на который ссылается это свойство. В следующем примере массив имеет ссылочный тип, поэтому отображается `System.String[]` вместо фактических значений элементов массива:

```
Person { FirstName = Nancy, LastName = Davolio, ChildNames = System.String[] }
```

Дополнительные сведения см. в разделе Встроенное форматирование статьи, посвященной записям, в справочнике по языку C#.

Наследование

Запись может наследовать от другой записи. Но запись не может наследовать от класса, а класс не может наследовать от записи.

Следующий пример демонстрирует наследование с использованием синтаксиса позиционных свойств:

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, Grade = 3 }
}
```

Чтобы две переменные записи считались равными, у них должен совпадать тип времени выполнения. При этом типы содержащихся в них переменных могут отличаться. Это демонстрируется в следующем примере кода:

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Person student = new Student("Nancy", "Davolio", 3);
    Console.WriteLine(teacher == student); // output: False

    Student student2 = new Student("Nancy", "Davolio", 3);
    Console.WriteLine(student2 == student); // output: True
}
```

В этом примере все экземпляры имеют одинаковые свойства и одинаковые значения этих свойств. Но выражение `student == teacher` дает значение `False`, хотя обе переменные имеют тип `Person`. При этом выражение `student == student2` дает значение `True`, хотя одна из переменных имеет тип `Person`, а другая — `Student`.

В выходные данные `ToString` включаются все свойства и поля с атрибутом public, как в производных, так и базовых типах, как показано в следующем примере:

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);

public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, Grade = 3 }
}
```

Дополнительные сведения см. в разделе Наследование статьи, посвященной записям, в справочнике по языку C#.

Методы задания только инициализации

Методы задания только для инициализации обеспечивают единообразный синтаксис для инициализации членов объекта. Инициализаторы свойств позволяют ясно понять, какое значение задает то или иное свойство. Недостаток заключается в том, что эти свойства должны быть устанавливаемыми. Начиная с C# 9.0, для свойств и индексаторов можно создавать методы доступа `init`, а не методы доступа `set`. Вызывающие объекты могут использовать синтаксис инициализатора свойств для установки этих значений в выражениях создания, но после завершения конструирования эти свойства будут доступны только для чтения. Методы задания только для инициализации предоставляют окно для изменения состояния. Это окно закрывается, когда завершается этап конструирования. Этап конструирования фактически завершается после всех инициализаций, включая инициализаторы свойств и выражения with.

Можно объявить методы задания только для инициализации (`init`) в любом написанном вами типе. Ниже приведен пример определения структуры наблюдения за погодой.

```
public struct WeatherObservation
{
    public DateTime RecordedAt { get; init; }
    public decimal TemperatureInCelsius { get; init; }
    public decimal PressureInMillibars { get; init; }

    public override string ToString() =>
        $"At {RecordedAt:h:mm tt} on {RecordedAt:M/d/yyyy}: " +
        $"Temp = {TemperatureInCelsius}, with {PressureInMillibars} pressure";
}
```

Вызывающие объекты могут использовать синтаксис инициализатора свойств для установки значений, сохраняя при этом неизменность.

```
var now = new WeatherObservation
{
    RecordedAt = DateTime.Now,
    TemperatureInCelsius = 20,
    PressureInMillibars = 998.0m
};
```

Попытка изменить наблюдение после инициализации приведет к ошибке компилятора:

```
// Error! CS8852.
now.TemperatureInCelsius = 18;
```

Методы задания только для инициализации могут быть полезны для задания свойств базового класса из производных классов. Они также могут устанавливать производные свойства через вспомогательные методы в базовом классе. В позиционных записях свойства объявляются с помощью методов задания только для инициализации. Эти методы задания используются в выражениях with. Методы задания только для инициализации можно объявить для любых создаваемых вами `class`, `struct` или `record`.

Дополнительные сведения см. в разделе, посвященном ключевому слову init (справочник по C#).

Инструкции верхнего уровня

Инструкции верхнего уровня избавляют от ненужных формальностей во многих приложениях. Рассмотрим каноническую программу Hello World!. .

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Здесь только одна строка кода, которая что-то делает. С помощью инструкций верхнего уровня можно заменить весь этот шаблон директивой `using` и одной строкой, которая делает всю работу:

```
using System;

Console.WriteLine("Hello World!");
```

Если требуется однострочная программа, можно удалить директиву `using` и использовать полное имя типа.

```
System.Console.WriteLine("Hello World!");
```

Только один файл в приложении может использовать инструкции верхнего уровня. Если компилятор обнаруживает операторы верхнего уровня в нескольких исходных файлах, это приводит к ошибке. Ошибка также возникает, если объединить операторы верхнего уровня с объявленным методом точки входа программы (как правило, это метод `Main`). В определенном смысле можно сказать, что один файл содержит инструкции, которые обычно находятся в методе `Main` класса `Program`.

Одним из наиболее распространенных применений этой функции является создание обучающих материалов. Начинающие разработчики на языке C# могут написать каноническую программу Hello World! в одной-двух строках кода. Никакие дополнительные формальности не требуются. Но и опытные разработчики также найдут много применений для этой функции. Инструкции верхнего уровня позволяют экспериментировать в стиле написания сценариев, аналогично записным книжкам Jupyter. Инструкции верхнего уровня отлично подходят для небольших консольных и служебных программ. Функции Azure являются идеальным примером использования операторов верхнего уровня.

Что важнее всего, инструкции верхнего уровня не ограничивают область применения или сложность приложения. Эти инструкции могут обращаться к любому классу .NET и использовать его. Они также не ограничивают использование аргументов командной строки и возвращаемых значений. Инструкции верхнего уровня могут обращаться к массиву строк с именем `args`. Если инструкции верхнего уровня возвращают целочисленное значение, это значение преобразуется в целочисленный код возврата из синтезированного метода `Main`. Инструкции верхнего уровня могут содержать асинхронные выражения. В этом случае синтезированная точка входа возвращает `Task` или `Task<int>`.

Дополнительные сведения см. в статье Инструкции верхнего уровня руководства по программированию на C#.

Улучшения сопоставления шаблонов

C# 9 включает новые улучшения сопоставления шаблонов.

- ***Шаблоны типов*** проверяют соответствие переменной определенному типу.
- ***Шаблоны в круглых скобках*** усиливают или подчеркивают приоритет сочетаний шаблонов.
- ***В шаблонах конъюнкций*** `and` требуется соответствие обоих шаблонов.
- ***В шаблонах дизъюнкций*** `or` требуется соответствие хотя бы одного из шаблонов.
- В ***шаблонах*** `not` ***с отрицанием*** требуется несоответствие данного шаблона.
- ***В шаблонах сравнения*** требуется, чтобы входные данные были меньше, больше, меньше или равны, больше или равны данной константе.

Эти шаблоны обогащают синтаксис шаблонов. Рассмотрим следующие примеры.

```
public static bool IsLetter(this char c) =>
    c is >= 'a' and <= 'z' or >= 'A' and <= 'Z';
```

С помощью дополнительных скобок можно явно указать, что `and` имеет более высокий приоритет, чем

`or`.

```
public static bool IsLetterOrSeparator(this char c) =>
    c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z') or '.' or ',';
```

Одним из наиболее распространенных применений нового синтаксиса является проверка значения на null.

```
if (e is not null)
{
    // ...
}
```

Любой из этих шаблонов можно использовать в любом контексте, где разрешены шаблоны: выражения с шаблоном `is`, выражения `switch`, вложенные шаблоны и шаблоны метки `case` оператора `switch`.

Дополнительные сведения см. в разделе Шаблоны (справочник по C#).

Дополнительные сведения см. в разделах Реляционные шаблоны и Логические шаблоны статьи Шаблоны.

Производительность и взаимодействие

Три новых функции улучшают поддержку собственного взаимодействия и низкоуровневых библиотек, требующих высокой производительности: целые числа собственного размера, указатели функций и пропуск флага `localsinit`.

Целые числа собственного размера `nint` и `nuint` являются целочисленными типами. Они выражаются базовыми типами System.IntPtr и System.UIntPtr. Компилятор предоставляет дополнительные преобразования и операции для этих типов в качестве собственных целых чисел. Целые числа собственного размера определяют свойства для `MaxValue` или `MinValue`. Такие значения не могут быть выражены как константы времени компиляции, так как они зависят от собственного размера целого числа на целевом компьютере. Во время выполнения эти значения доступны только для чтения. Для `nint` можно использовать значения констант в диапазоне [`int.MinValue` .. `int.MaxValue`]. Для `nuint` можно использовать значения констант в диапазоне [`uint.MinValue` .. `uint.MaxValue`]. Компилятор выполняет сворачивание константы для всех унарных и бинарных операторов, используя типы System.Int32 и System.UInt32. Если результат не помещается в 32 бит, операция выполняется во время выполнения и не считается константой. Целые числа собственного размера могут повысить производительность в сценариях с большим количеством целочисленных вычислений, в которых необходимо обеспечить максимально высокую производительность. Дополнительные сведения см. в статье о типах `nint` и `nuint`.

Указатели функций предоставляют простой синтаксис для доступа к кодам операций IL `ldftn` и `calli`. Указатели функций можно объявлять с помощью нового синтаксиса `delegate*`. Тип `delegate*` — это тип указателя. При вызове типа `delegate*` используется `calli`, в отличие от делегата, который использует `callvirt` в методе `Invoke()`. Синтаксически вызовы являются идентичными. При вызове указателя функции используется соглашение о вызовах `managed`. Если требуется объявить о соглашении о вызовах `unmanaged`, добавьте ключевое слово `unmanaged` после синтаксиса `delegate*`. Другие соглашения о вызовах можно указать с помощью атрибутов в объявлении `delegate*`. Дополнительные сведения см. в разделе Небезопасный код и типы указателей.

Наконец, можно добавить атрибут System.Runtime.CompilerServices.SkipLocalsInitAttribute, чтобы компилятор не создавал флаг `localsinit`. Этот флаг указывает среде CLR на нулевую инициализацию всех локальных переменных. Флаг `localsinit` используется в C# по умолчанию, начиная с версии 1.0.

Однако при использовании дополнительной нулевой инициализации в некоторых сценариях может снизиться производительность. В частности, при использовании `stackalloc`. В таких случаях можно добавить атрибут SkipLocalsInitAttribute. Его можно добавить в один метод или свойство, в `class`, `struct`, `interface` или даже в модуль. Этот атрибут не влияет на методы `abstract`. Он влияет на код, созданный для реализации. Дополнительные сведения см. в разделе об атрибуте `SkipLocalsInit`.

Эти функции могут повысить производительность в некоторых сценариях. Их следует использовать только после тщательного тестирования как до, так и после внедрения. Код, включающий целые числа собственного размера, нужно тестировать на нескольких целевых платформах с различными размерами целых чисел. Другие функции требуют небезопасный код.

Функции подбора и завершения

Многие другие функции позволяют более эффективно писать код. В C# 9.0 можно опустить тип в выражении `new`, если тип созданного объекта уже известен. Наиболее часто это используется в объявлениях полей.

```
private List<WeatherObservation> _observations = new();
```

`new` с целевым типом можно также использовать, если необходимо создать объект для передачи его в качестве аргумента в метод. Рассмотрим метод `ForecastFor()` со следующей сигнатурой.

```
public WeatherForecast ForecastFor(DateTime forecastDate, WeatherForecastOptions options)
```

Его можно вызвать следующим образом.

```
var forecast = station.ForecastFor(DateTime.Now.AddDays(2), new());
```

Еще один полезный способ использовать эту функцию — объединить ее со свойствами только для инициализации при инициализации нового объекта.

```
WeatherStation station = new() { Location = "Seattle, WA" };
```

Экземпляр, созданный конструктором по умолчанию, можно вернуть с помощью инструкции `return new();`.

Аналогичная функция улучшает разрешение целевого типа в условных выражениях. Благодаря такому изменению два выражения необязательно должны иметь неявное преобразование из одного в другое, но оба могут иметь неявные преобразования в целевой тип. Скорее всего, вы не заметите этого изменения. Обратите внимание, что некоторые условные выражения, которые ранее требовали приведения или не компилировались, теперь просто начнут работать.

Начиная с C# 9.0, можно добавлять модификатор `static` в лямбда-выражения или анонимные методы. Статические лямбда-выражения аналогичны локальным функциям с модификатором `static`: статические лямбда-выражения и анонимные методы не могут захватывать локальные переменные и состояние экземпляра. Модификатор `static` предотвращает случайное захватывание других переменных.

Ковариантные возвращаемые типы обеспечивают гибкость для типов возвращаемых значений методов переопределения. Метод переопределения может возвращать тип, производный от типа возвращаемых значений переопределенного базового метода. Это может быть полезно для записей и для других типов, которые поддерживают виртуальные методы клонирования или фабричные методы.

Кроме того, цикл `foreach` будет распознавать и использовать метод расширения `GetEnumerator`, который в противном случае удовлетворяет шаблону `foreach`. Это изменение означает, что `foreach` согласуется с другими конструкциями на основе шаблонов, такими как асинхронная модель и деконструирование на основе шаблона. На практике это изменение означает, что можно добавить поддержку `foreach` в любой тип. При перечислении объектов имеет смысл ограничить его использование.

Кроме того, можно использовать пустые переменные в качестве параметров для лямбда-выражений. Это удобное изменение позволяет избежать присвоения имени аргументу, а компилятор может избежать его использования. Для любого аргумента используется символ `_`. Дополнительные сведения см. в разделе Входные параметры лямбда-выражения статьи о лямбда-выражениях.

Наконец, теперь можно применять атрибуты к локальным функциям. Например, к локальным функциям можно применить заметки атрибутов, допускающих значение null.

Поддержка генераторов кода

Две заключительные функции обеспечивают поддержку генераторов кода в C#. Генераторы кода C# — это компонент, который вы можете написать подобно анализатору или исправлению кода Roslyn. Разница заключается в том, что генераторы кода анализируют код и пишут новые файлы исходного кода в рамках процесса компиляции. Типичный генератор кода ищет в коде атрибуты или другие соглашения.

Генератор кода считывает атрибуты или другие элементы кода с помощью анализирующих интерфейсов API Roslyn. На основе этой информации он добавляет новый код в компиляцию. Генераторы исходного кода могут только добавлять код, они не могут изменять существующий код в компиляции.

Этими двумя функциями, добавленными для поддержки генераторов кода, являются расширения для ***синтаксиса разделяемого метода*** *_ и _инициализаторов модулей*_**. Сначала рассмотрим изменения в разделяемые методы. До C# 9.0 разделяемые методы были `private`, но не могли иметь модификаторов доступа, иметь возвращаемое значение `void` и параметры `out`. Эти ограничения подразумевают, что, если реализация метода не предоставлена, компилятор удаляет все вызовы к разделяемому методу. В C# 9.0 эти ограничения снимаются, но требуется, чтобы объявления разделяемых методов имели реализацию. Генераторы кода могут предоставить такую реализацию. Чтобы избежать критических изменений, компилятор рассматривает любой разделяемый метод без модификатора доступа как метод, следующий старым правилам. Если разделяемый метод включает модификатор доступа `private`, этот разделяемый метод обрабатывается в соответствии с новыми правилами. Дополнительные сведения см. в разделе о разделяемом методе (Справочник по C#).

Второй новой функцией для генераторов кода являются ***инициализаторы модулей***. Инициализаторы модулей — это методы, к которым прикреплен атрибут ModuleInitializerAttribute. Эти методы будут вызываться средой выполнения до доступа к полю или вызова метода в целом модуле. Метод инициализатора модуля:

- должен быть статическим;
- должен быть без параметров;
- должен возвращать значение void;
- не должен быть универсальным методом;
- не должен содержаться в универсальном классе;
- должен быть доступен из содержащего модуля.

Последний пункт фактически означает, что метод и содержащий его класс должны быть внутренними или открытыми. Метод не может быть локальной функцией. Дополнительные сведения см. в разделе об атрибуте `ModuleInitializer`.

Новые возможности C# 8.0

18.12.2021 • 15 minutes to read

В C# 8.0 добавлены следующие функции и улучшения языка C#:

- Члены только для чтения
- Методы интерфейса по умолчанию
- Улучшения сопоставления шаблонов:
  - выражения switch;
  - шаблоны свойств;
  - шаблоны кортежей;
  - позиционные шаблоны.
- Объявления using.
- Статические локальные функции.
- Удаляемые ссылочные структуры.
- Ссылочные типы, допускающие значение NULL
- Асинхронные потоки.
- Асинхронные высвобождаемые типы
- Индексы и диапазоны.
- Присваивание объединения со значением NULL
- Неуправляемые сконструированные типы
- Выражение stackalloc во вложенных выражениях
- Улучшение интерполированных строк verbatim

C# 8.0 поддерживается в **.NET Core 3.x** и **.NET Standard 2.1**. Дополнительные сведения см. в статье Управление версиями языка C#.

В остальных разделах этой статьи кратко описываются эти возможности. Здесь приведены ссылки на эти подробные руководства и обзоры (если они доступны). Эти функции можно изучить в своей среде с помощью глобального средства `dotnet try`:

1. Установите глобальное средство dotnet-try.
2. Клонируйте репозиторий dotnet/try-samples.
3. Для репозитория *try-samples* установите в качестве текущего каталога подкаталог *csharp8*.
4. Запустите `dotnet try`.

Члены только для чтения

Модификатор `readonly` можно применить к членам структуры. Он означает, что член не изменяет состояние. Это более детализированный способ, чем применение модификатора `readonly` в объявлении `struct`. Рассмотрим следующую изменяемую структуру:

```
public struct Point
{
    public double X { get; set; }
    public double Y { get; set; }
    public double Distance => Math.Sqrt(X * X + Y * Y);

    public override string ToString() =>
        $"({X}, {Y}) is {Distance} from the origin";
}
```

Как и большинство структур, метод `ToString()` не изменяет состояние. Это можно указать, добавив модификатор `readonly` в объявление `ToString()`:

```
public readonly override string ToString() =>
    $"({X}, {Y}) is {Distance} from the origin";
```

Предыдущее изменение создает предупреждение компилятора, так как `ToString` обращается к свойству `Distance`, которое не помечено как `readonly`:

```
warning CS8656: Call to non-readonly member 'Point.Distance.get' from a 'readonly' member results in an
implicit copy of 'this'
```

Компилятор выдает предупреждение, когда ему требуется создать защитную копию. Свойство `Distance` не изменяет состояние, поэтому вы можете устранить это предупреждение, добавив модификатор `readonly` в объявление:

```
public readonly double Distance => Math.Sqrt(X * X + Y * Y);
```

Обратите внимание, что модификатор `readonly` необходим для свойства только для чтения. Компилятор не предполагает, что методы доступа `get` не изменяют состояние; `readonly` необходимо объявить явно. Автоматические реализуемые свойства являются исключением; компилятор будет рассматривать все автоматические реализуемые методы получения как `readonly`, поэтому не нужно добавлять модификатор `readonly` к свойствам `X` и `Y`.

Компилятор принудительно применяет правило, в соответствии с которым члены `readonly` не изменяют состояние. Следующий метод не будет компилироваться, если не удалить модификатор `readonly`:

```
public readonly void Translate(int xOffset, int yOffset)
{
    X += xOffset;
    Y += yOffset;
}
```

Эта функция позволяет указать намерение вашего проекта, чтобы компилятор мог применить его и выполнить оптимизацию на основе намерения.

Дополнительные сведения см. в разделе о членах экземпляров `readonly` в статье Типы структур.

Методы интерфейса по умолчанию

Теперь вы можете добавлять члены в интерфейсы и предоставлять реализацию для этих членов. Эта возможность языка позволяет разработчикам API добавлять методы в интерфейс в более поздних версиях, не нарушая исходный код или совместимость на уровне двоичного кода с существующими

реализациями этого интерфейса. Существующие реализации *наследуют* реализацию по умолчанию. Эта функция также позволяет C# взаимодействовать с API-интерфейсами, предназначенными для Android или Swift, которые поддерживают аналогичные функциональные возможности. Методы интерфейса по умолчанию также поддерживают сценарии, аналогичные функции языка "признаки".

Методы интерфейса по умолчанию влияют на многие сценарии и элементы языка. В нашем первом учебнике рассматривается изменение интерфейса с помощью реализаций по умолчанию.

Дополнительные шаблоны в нескольких расположениях

Возможность **сопоставления шаблонов** позволяет работать с шаблонами в зависимости от формата в связанных, но различных типах данных. В C# 7.0 появился синтаксис для шаблонов типа и шаблонов константы, использующий выражение `is` и инструкцию `switch`. Эти функции представляют первые пробные шаги на пути к поддержке парадигм программирования, где данные и функции разделены. По мере того как отрасль переходит на использование микрослужб и других облачных архитектур, необходимы также средства других языков.

В C# 8.0 расширены эти возможности, так что вы можете использовать дополнительные выражения шаблонов в нескольких расположениях в коде. Учитывайте эти функции, когда данные и функциональные возможности представлены отдельно. Рассмотрите возможность сопоставления шаблонов, когда алгоритмы зависят от фактов, отличных от типа среды выполнения объекта. Эти методы предоставляют другой способ выражения проектов.

Помимо новых шаблонов в новых расположениях, в C# 8.0 добавлены **рекурсивные шаблоны**. Рекурсивные шаблоны — это шаблоны, которые могут содержать другие шаблоны.

Выражения switch

Часто инструкция `switch` возвращает значение в каждом из блоков `case`. **Выражения switch** позволяет использовать более краткий синтаксис выражения. В нем меньше повторяющихся ключевых слов `case` и `break`, а также меньше фигурных скобок. Например, рассмотрим следующее перечисление, которое выводит список цветов радуги:

```
public enum Rainbow
{
    Red,
    Orange,
    Yellow,
    Green,
    Blue,
    Indigo,
    Violet
}
```

Если для приложения определен тип `RGBColor`, который создается на основе компонентов `R`, `G` и `B`, вы можете преобразовать значение `Rainbow` в RGB-значение, используя следующий метод, содержащий выражение switch:

```
public static RGBColor FromRainbow(Rainbow colorBand) =>
    colorBand switch
    {
        Rainbow.Red    => new RGBColor(0xFF, 0x00, 0x00),
        Rainbow.Orange => new RGBColor(0xFF, 0x7F, 0x00),
        Rainbow.Yellow => new RGBColor(0xFF, 0xFF, 0x00),
        Rainbow.Green  => new RGBColor(0x00, 0xFF, 0x00),
        Rainbow.Blue   => new RGBColor(0x00, 0x00, 0xFF),
        Rainbow.Indigo => new RGBColor(0x4B, 0x00, 0x82),
        Rainbow.Violet => new RGBColor(0x94, 0x00, 0xD3),
        _              => throw new ArgumentException(message: "invalid enum value", paramName:
nameof(colorBand)),
    };
```

Здесь представлено несколько улучшений синтаксиса:

- Переменная расположена перед ключевым словом `switch`. Другой порядок позволяет визуально легко отличить выражение switch от инструкции switch.
- Элементы `case` и `:` заменяются на `=>`. Это более лаконично и интуитивно понятно.
- Случай `default` заменяется пустой переменной `_`.
- Тексты являются выражениями, а не инструкциями.

Сравните это с эквивалентным кодом, где используется классическая инструкция `switch`:

```
public static RGBColor FromRainbowClassic(Rainbow colorBand)
{
    switch (colorBand)
    {
        case Rainbow.Red:
            return new RGBColor(0xFF, 0x00, 0x00);
        case Rainbow.Orange:
            return new RGBColor(0xFF, 0x7F, 0x00);
        case Rainbow.Yellow:
            return new RGBColor(0xFF, 0xFF, 0x00);
        case Rainbow.Green:
            return new RGBColor(0x00, 0xFF, 0x00);
        case Rainbow.Blue:
            return new RGBColor(0x00, 0x00, 0xFF);
        case Rainbow.Indigo:
            return new RGBColor(0x4B, 0x00, 0x82);
        case Rainbow.Violet:
            return new RGBColor(0x94, 0x00, 0xD3);
        default:
            throw new ArgumentException(message: "invalid enum value", paramName: nameof(colorBand));
    };
}
```

Дополнительные сведения см. в разделе о выражении `switch`.

Шаблоны свойств

Шаблон свойств позволяет сопоставлять свойства исследуемого объекта. Рассмотрим сайт электронной коммерции, на котором должен вычисляться налог с продаж по адресу покупателя. Это вычисление не является основной задачей класса `Address`. Оно меняется со временем и, скорее всего, чаще, чем изменения формата адреса. Сумма налога с продаж зависит от свойства `State` адреса. В следующем методе используется шаблон свойства для вычисления налога с продаж по адресу и цене:

```
public static decimal ComputeSalesTax(Address location, decimal salePrice) =>
    location switch
    {
        { State: "WA" } => salePrice * 0.06M,
        { State: "MN" } => salePrice * 0.075M,
        { State: "MI" } => salePrice * 0.05M,
        // other cases removed for brevity...
        _ => 0M
    };
```

При сопоставлении шаблонов создается сокращенный синтаксис для выражения этого алгоритма.

Дополнительные сведения см. в разделе Шаблон свойства статьи Шаблоны.

Шаблоны кортежей

Некоторые алгоритмы зависят от нескольких наборов входных данных. **Шаблоны кортежей** позволяют переключаться между несколькими значениями, выраженными как кортежи. В следующем примере кода показано выражение switch для игры *камень, ножницы, бумага*.

```
public static string RockPaperScissors(string first, string second)
    => (first, second) switch
    {
        ("rock", "paper") => "rock is covered by paper. Paper wins.",
        ("rock", "scissors") => "rock breaks scissors. Rock wins.",
        ("paper", "rock") => "paper covers rock. Paper wins.",
        ("paper", "scissors") => "paper is cut by scissors. Scissors wins.",
        ("scissors", "rock") => "scissors is broken by rock. Rock wins.",
        ("scissors", "paper") => "scissors cuts paper. Scissors wins.",
        (_, _) => "tie"
    };
```

В сообщении указан победитель. В случае отклонения представляются три комбинации, ведущие к ничьей, или другие текстовые входные данные.

Позиционные шаблоны

Некоторые типы включают метод `Deconstruct`, свойства которого деконструируются на дискретные переменные. Если метод `Deconstruct` доступен, можно использовать **позиционные шаблоны** для проверки свойств объекта и использовать эти свойства для шаблона. Рассмотрим следующий класс `Point`, который содержит метод `Deconstruct` для создания дискретных переменных для `X` и `Y`:

```
public class Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y) => (X, Y) = (x, y);

    public void Deconstruct(out int x, out int y) =>
        (x, y) = (X, Y);
}
```

Кроме того, нужно учитывать следующее перечисление, представляющее различные позиции квадранта:

```
public enum Quadrant
{
    Unknown,
    Origin,
    One,
    Two,
    Three,
    Four,
    OnBorder
}
```

В следующем методе используется **позиционный шаблон** для извлечения значений `x` и `y`. Затем используется предложение `when` для определения `Quadrant` точки:

```
static Quadrant GetQuadrant(Point point) => point switch
{
    (0, 0) => Quadrant.Origin,
    var (x, y) when x > 0 && y > 0 => Quadrant.One,
    var (x, y) when x < 0 && y > 0 => Quadrant.Two,
    var (x, y) when x < 0 && y < 0 => Quadrant.Three,
    var (x, y) when x > 0 && y < 0 => Quadrant.Four,
    var (_, _) => Quadrant.OnBorder,
    _ => Quadrant.Unknown
};
```

Шаблон пустой переменной в предыдущем операторе switch совпадает с выражением, если `x` или `y`, но не оба, имеет значение 0. Выражение switch должно создавать значение или исключение. Если ни один из вариантов не совпадает, выражение switch создает исключение. Компилятор создает предупреждение, если в выражении switch не охватываются все возможные случаи.

Ознакомиться с методами сопоставления шаблонов можно в этом подробном учебнике. Дополнительные сведения о позиционном шаблоне см. в разделе Позиционный шаблон статьи Шаблоны.

Объявления using

Объявление using — это объявление переменной, которому предшествует ключевое слово `using`. Оно сообщает компилятору, что объявляемая переменная должна быть удалена в конце области видимости. Для примера рассмотрим следующий код, в котором записывается текстовый файл:

```
static int WriteLinesToFile(IEnumerable<string> lines)
{
    using var file = new System.IO.StreamWriter("WriteLines2.txt");
    int skippedLines = 0;
    foreach (string line in lines)
    {
        if (!line.Contains("Second"))
        {
            file.WriteLine(line);
        }
        else
        {
            skippedLines++;
        }
    }
    // Notice how skippedLines is in scope here.
    return skippedLines;
    // file is disposed here
}
```

В приведенном выше примере файл удаляется при достижении закрывающей фигурной скобки метода. Это конец области, в котором объявляется `file`. Приведенный выше код эквивалентен следующему коду, в котором используется классическая инструкция using:

```
static int WriteLinesToFile(IEnumerable<string> lines)
{
    using (var file = new System.IO.StreamWriter("WriteLines2.txt"))
    {
        int skippedLines = 0;
        foreach (string line in lines)
        {
            if (!line.Contains("Second"))
            {
                file.WriteLine(line);
            }
            else
            {
                skippedLines++;
            }
        }
        return skippedLines;
    } // file is disposed here
}
```

В приведенном выше примере файл удаляется при достижении закрывающей фигурной скобки, связанной с инструкцией `using`.

В обоих случаях компилятор создает вызов метода `Dispose()`. Компилятор создает ошибку, если выражение в инструкции `using` нельзя удалить.

Статические локальные функции

Теперь вы можете добавить модификатор `static` в локальные функции, чтобы убедиться, что локальная функция не сохраняет (не ссылается на) какие-либо переменные из включающей области. Это приводит к возникновению ошибки `CS8421`: "A static local function can't contain a reference to <variable>" (Статическая локальная функция не может содержать ссылку на <variable>).

Рассмотрим следующий код. Локальная функция `LocalFunction` обращается к переменной `y`, объявленной в области видимости (метод `M`). Таким образом `LocalFunction` не может объявляться с помощью модификатора `static`:

```
int M()
{
    int y;
    LocalFunction();
    return y;

    void LocalFunction() => y = 0;
}
```

Следующий код содержит статическую локальную функцию. Она может быть статической, так как она не обращается ко всем переменным в области видимости:

```
int M()
{
    int y = 5;
    int x = 7;
    return Add(x, y);

    static int Add(int left, int right) => left + right;
}
```

Удаляемые ссылочные структуры

Объект `struct`, объявленный с помощью модификатора `ref`, не может реализовывать интерфейсы и поэтому не может реализовать IDisposable. Таким образом, чтобы объект `ref struct` можно было удалить, у него должен быть доступный метод `void Dispose()`. Эта функция также применима к объявлениям `readonly ref struct`.

Ссылочные типы, допускающие значение null

Внутри контекста заметок, допускающих значение null, любая переменная ссылочного типа считается переменной **ссылочного типа**, **не допускающего значения null**. Если вы хотите указать, что переменная может принимать значение null, необходимо добавить к имени типа `?`, чтобы объявить переменную как переменную **ссылочного типа**, **допускающего значения null**.

Для ссылочных типов, не допускающих значение null, компилятор использует анализ потока, чтобы убедиться, что локальные переменные инициализируются в значение, отличное от null, при объявлении. Поля должны быть инициализированы во время построения. Компилятор создает предупреждение, если переменная не задается вызовом ни к одному из доступных конструкторов или инициализатором. Кроме того, ссылочным типам, не допускающим значение null, нельзя задать значение, которое может быть null.

Ссылочные типы, допускающие значение null, не проверяются на предмет того, назначено ли им значение null или получили ли они его после инициализации. Тем не менее, компилятор использует анализ потока, чтобы убедиться, что любая переменная ссылочного типа, допускающего значение null, проверяется на наличие значения null, прежде чем обратиться к ней или назначить ей ссылочный тип, не допускающий значение null.

Дополнительные сведения о функции см. в обзоре ссылочных типов, допускающих значение null. Попробуйте самостоятельно применить эту возможность в новом приложении в этом руководстве по ссылочным типам, допускающим значение null. Дополнительные сведения о шагах переноса существующей базы кода для использования ссылочных типов, допускающих значение NULL, см. в статье о переходе на использование ссылочных типов, допускающих значение NULL.

Асинхронные потоки

Начиная с C# версии 8.0 можно создавать и использовать потоки асинхронно. В методе, который возвращает асинхронный поток, есть три свойства:

1. Он объявлен с помощью модификатора `async`.
2. Он возвращает интерфейс IAsyncEnumerable<T>.
3. Метод содержит инструкции `yield return` для возвращения последовательных элементов в асинхронном потоке.

Для использования асинхронного потока требуется добавить ключевое слово `await` перед ключевым словом `foreach` при перечислении элементов потока. Для добавления ключевого слова `await` требуется, чтобы метод, который перечисляет асинхронный поток, был объявлен с помощью

модификатора `async` и возвращал тип, допустимый для метода `async`. Обычно это означает возвращение структуры Task или Task<TResult>. Это также может быть структура ValueTask или ValueTask<TResult>. Метод может использовать и создавать асинхронный поток. Это означает, что будет возвращен интерфейс IAsyncEnumerable<T>. Следующий код создает последовательность чисел от 0 до 19 с интервалом 100 мс между генерированием каждого числа:

```
public static async System.Collections.Generic.IAsyncEnumerable<int> GenerateSequence()
{
    for (int i = 0; i < 20; i++)
    {
        await Task.Delay(100);
        yield return i;
    }
}
```

Вы бы перечислили последовательность с использованием инструкции `await foreach`:

```
await foreach (var number in GenerateSequence())
{
    Console.WriteLine(number);
}
```

Вы можете попробовать асинхронные потоки самостоятельно в нашем руководстве по созданию и использованию асинхронных потоков. Элементы потока по умолчанию обрабатываются в захваченном контексте. Чтобы отключить захват контекста, используйте метод расширения TaskAsyncEnumerableExtensions.ConfigureAwait. Дополнительные сведения о контекстах синхронизации и захвате текущего контекста см. в статье, посвященной использованию асинхронной модели на основе задач.

Асинхронные высвобождаемые типы

Начиная с C# 8.0 язык поддерживает асинхронные освобождаемые типы, реализующие интерфейс System.IAsyncDisposable. Для работы с асинхронно освобождаемым объектами используется инструкция `await using`. Дополнительные сведения см. в статье Реализация метода DisposeAsync.

Индексы и диапазоны

Диапазоны и индексы обеспечивают лаконичный синтаксис для доступа к отдельным элементам или диапазонам в последовательности.

Поддержка языков опирается на два новых типа и два новых оператора:

- System.Index представляет индекс в последовательности.
- Оператор `^` (индекс с конца), который указывает, что индекс указан относительно конца последовательности.
- System.Range представляет вложенный диапазон последовательности.
- Оператор диапазона `..`, который задает начало и конец диапазона в качестве своих операндов.

Начнем с правил для использования в индексах. Рассмотрим массив `sequence`. Индекс `0` совпадает с `sequence[0]`. Индекс `^0` совпадает с `sequence[sequence.Length]`. Обратите внимание, что `sequence[^0]` создает исключение так же, как и `sequence[sequence.Length]`. Для любого числа `n` индекс `^n` совпадает с `sequence.Length - n`.

Диапазон указывает *начало* и *конец* диапазона. Начало диапазона является включающим, но конец диапазона является исключающим, то есть *начало* включается в диапазон, а *конец* не включается.

Диапазон `[0..^0]` представляет весь диапазон так же, как `[0..sequence.Length]` представляет весь диапазон.

Рассмотрим несколько примеров. Обратите внимание на следующий массив, который помечен индексом от начала и от конца:

```
var words = new string[]
{
                // index from start    index from end
    "The",      // 0                   ^9
    "quick",    // 1                   ^8
    "brown",    // 2                   ^7
    "fox",      // 3                   ^6
    "jumped",   // 4                   ^5
    "over",     // 5                   ^4
    "the",      // 6                   ^3
    "lazy",     // 7                   ^2
    "dog"       // 8                   ^1
};              // 9 (or words.Length) ^0
```

Вы можете получить последнее слово с помощью индекса `^1`:

```
Console.WriteLine($"The last word is {words[^1]}");
// writes "dog"
```

Следующий код создает поддиапазон со словами "quick", "brown" и "fox". Он включает в себя элементы от `words[1]` до `words[3]`. Элемент `words[4]` в диапазон не входит.

```
var quickBrownFox = words[1..4];
```

Следующий код создает поддиапазон со словами "lazy" и "dog". Он включает элементы `words[^2]` и `words[^1]`. Конечный индекс `words[^0]` не включен:

```
var lazyDog = words[^2..^0];
```

В следующих примерах создаются диапазоны, которые должны быть открыты для начала, конца или в обоих случаях:

```
var allWords = words[..]; // contains "The" through "dog".
var firstPhrase = words[..4]; // contains "The" through "fox"
var lastPhrase = words[6..]; // contains "the", "lazy" and "dog"
```

Вы также можете объявить диапазоны как переменные:

```
Range phrase = 1..4;
```

Затем вы можете использовать диапазон внутри символов `[` и `]`:

```
var text = words[phrase];
```

Индексы и диапазоны поддерживаются не только массивами. Можно также использовать индексы и диапазоны со строкой (Span<T> или ReadOnlySpan<T>). Дополнительные сведения см. в разделе Поддержка типа для индексов и диапазонов.

Вы можете изучить сведения об индексах и диапазонах адресов в руководстве Индексы и диапазоны.

Присваивание объединения со значением NULL

В C# 8.0 появился оператор присваивания объединения со значением NULL `??=`. Оператор `??=` можно использовать для присваивания значения правого операнда левому операнду только в том случае, если левый операнд принимает значение `null`.

```
List<int> numbers = null;
int? i = null;

numbers ??= new List<int>();
numbers.Add(i ??= 17);
numbers.Add(i ??= 20);

Console.WriteLine(string.Join(" ", numbers));  // output: 17 17
Console.WriteLine(i);  // output: 17
```

Дополнительные сведения см. в статье Операторы ?? и ??=.

Неуправляемые сконструированные типы

В C# 7.3 и более ранних версиях сконструированный тип (тип, содержащий по крайней мере один аргумент типа) не может быть неуправляемым типом. Начиная с C# 8.0, сконструированный тип значения является неуправляемым, если он содержит поля исключительно неуправляемых типов.

Например, при наличии следующего определения универсального типа `Coords<T>`:

```
public struct Coords<T>
{
    public T X;
    public T Y;
}
```

тип `Coords<int>` является неуправляемым в C# 8.0 и более поздних версиях. Как и для любого неуправляемого типа, вы можете создать указатель на переменную этого типа или выделить блок памяти в стеке для экземпляров этого типа:

```
Span<Coords<int>> coordinates = stackalloc[]
{
    new Coords<int> { X = 0, Y = 0 },
    new Coords<int> { X = 0, Y = 3 },
    new Coords<int> { X = 4, Y = 0 }
};
```

Дополнительные сведения см. в разделе Неуправляемые типы.

Выражение stackalloc во вложенных выражениях

Начиная с C# 8.0, если результат выражения stackalloc имеет тип System.Span<T> или System.ReadOnlySpan<T>, можно использовать выражение `stackalloc` в других выражениях:

```
Span<int> numbers = stackalloc[] { 1, 2, 3, 4, 5, 6 };
var ind = numbers.IndexOfAny(stackalloc[] { 2, 4, 6, 8 });
Console.WriteLine(ind);  // output: 1
```

Улучшение интерполированных строк verbatim

Порядок маркеров `$` и `@` в интерполированных строках verbatim может быть любым: и `$@"..."`, и `@$"..."` являются допустимыми интерполированными строками verbatim. В более ранних версиях C# маркер `$` должен располагаться перед маркером `@`.

Сведения о критических изменениях в компиляторе C#

18.12.2021 • 2 minutes to read

Группа Roslyn поддерживает список критических изменений в компиляторах C# и Visual Basic. Информацию об этих изменениях можно найти по следующим ссылкам в репозитории GitHub:

- Критические изменения в Roslyn в версиях после .NET 6 и до .NET 7
- Критические изменения в Roslyn в C# 10.0/.NET 6
- Критические изменения в Roslyn после .NET 5
- Критические изменения в VS2019 версии 16.8, представленной для .NET 5 и C# 9.0
- Breaking changes in VS2019 Update 1 and beyond compared to VS2019 (Критические изменения в VS2019, обновление 1 и более поздние по сравнению с VS2019)
- Breaking changes since VS2017 (C# 7) (Критические изменения от VS2017 (C# 7))
- Breaking changes in Roslyn 3.0 (VS2019) from Roslyn 2.* (VS2017) (Критические изменения в Roslyn 3.0 (VS2019) из Roslyn 2. (VS2017))
- Breaking changes in Roslyn 2.0 (VS2017) from Roslyn 1.* (VS2015) and native C# compiler (VS2013 and previous). (Критические изменения в Roslyn 2.0 (VS2017) из Roslyn 1. (VS2015) и собственного компилятора C# (VS2013 и более ранней версии)).
- Breaking changes in Roslyn 1.0 (VS2015) from the native C# compiler (VS2013 and previous). (Критические изменения в Roslyn 1.0 (VS2015) из собственного компилятора C# (VS2013 и более ранней версии)).
- Изменение версии Юникода в C# 6

История языка C#

18.12.2021 • 12 minutes to read

В этой статье содержится описание основных выпусков языка C#. Команда разработчиков C# продолжает добавлять новые функции. Сведения о состоянии функции для отдельных языков, включая функции, которые будут добавлены в предстоящие выпуски, см. в репозитории dotnet/roslyn на сайте GitHub.

> **IMPORTANT**
>
> В некоторых возможностях используются типы и методы, которые в спецификации C# определены как *стандартная библиотека*. Платформа .NET предоставляет эти типы и методы в разных пакетах. например, обработка исключений. Каждая инструкция и выражение `throw` проверяется, чтобы убедиться, что вызываемый объект является производным от Exception. Аналогичным образом каждая инструкция `catch` проверяется, чтобы убедиться, что перехваченный тип является производным от Exception. В каждой версии могут добавляться новые требования. Чтобы использовать новейшие возможности языка в старой среде, может потребоваться установить определенные библиотеки. Эти зависимости описаны на странице для каждой конкретной версии. Дополнительные сведения о связи между языком и библиотекой, а также общие сведения о такой зависимости см. здесь.

C# версии 1.0

Если взглянуть на C# версии 1.0, выпущенный вместе с Visual Studio .NET 2002, станет очевидно его сходство с Java. В число поставленных перед ECMA задач проектирования входило создание "простого и современного объектно-ориентированного языка общего назначения". В то время схожесть с Java означала достижение этих ранних целей проектирования.

Однако если взглянуть на C# 1.0 сейчас, вам может стать немного не по себе. В этой версии не было встроенных асинхронных возможностей и некоторых функций для универсальных шаблонов, к которым вы уже привыкли. Фактически, в ней вообще не было универсальных шаблонов. А LINQ? Еще нет. Для добавления этих возможностей потребовалось несколько лет.

По современным меркам C# версии 1.0 не предоставлял практически никаких возможностей. Вам пришлось бы писать довольно подробный код. Однако надо же было с чего-то начинать. C# версии 1.0 был достойной альтернативой Java на платформе Windows.

Основные возможности C# 1.0 включали следующие компоненты:

- Классы
- Структуры
- Интерфейсы
- События
- Свойства
- Делегаты
- Операторы и выражения
- Операторы
- Атрибуты

Версия для C# 1.2

Версия C# 1.2 входит в состав Visual Studio .NET 2003. Она содержит ряд небольших улучшений языка. Наиболее значительным изменением является то, что начиная с этой версии код, создаваемый в цикле `foreach`, вызывает Dispose в IEnumerator, если IEnumerator реализует IDisposable.

C# версии 2.0

Вот теперь все становится гораздо интереснее. Давайте рассмотрим некоторые основные возможности языка C# 2.0, выпущенного в 2005 году вместе с Visual Studio 2005:

- Универсальные шаблоны
- Разделяемые типы
- Анонимные методы
- Типы значений, допускающие значение NULL
- Итераторы
- Ковариантность и контравариантность

Другие возможности C# 2.0 расширяли существующие возможности:

- Отдельный доступ к методу получения и методу задания
- Преобразования групп методов (делегаты)
- Статические классы
- Выведение делегата

Хотя сначала C# и был универсальным объектно-ориентированным языком, в C# версии 2.0 все кардинально изменилось. Нащупав опору под ногами, команда уделила внимание решению серьезных проблем, с которыми сталкивались разработчики. И они подошли к делу с огоньком.

Благодаря универсальным шаблонам вы получаете типы и методы, которые могут работать с произвольным типом, при этом обеспечивая безопасность типа. В результате, например, наличие List<T> позволяет использовать `List<string>` или `List<int>` и совершать типобезопасные операции с этими строками или целыми числами, выполняя итерацию по ним. Использовать универсальные шаблоны удобнее, чем создавать тип `ListInt`, производный от `ArrayList`, или выполнять приведение из `Object` для каждой операции.

В C# версии 2.0 появились итераторы. Если кратко, итераторы позволяют изучать все элементы в `List` (или других перечислимых типов) с помощью цикла `foreach`. Наличие итераторов в качестве первоклассного компонента в языке значительно улучшило удобочитаемость языка и способность воспринимать код.

А еще C# продолжил играть в салки с Java. Для Java уже были выпущены версии, включающие универсальные шаблоны и итераторы. Но вскоре этому было суждено измениться, так как языки продолжили развиваться независимо.

C# версии 3.0

C# версии 3.0 был выпущен в конце 2007 года вместе с Visual Studio 2008, однако существенно набор возможностей языка пополнится только в .NET Framework версии 3.5. Эта версия ознаменовала кардинальное изменение развития C#. Она доказала, что C# действительно является достойным языком программирования. Давайте рассмотрим некоторые основные возможности в этой версии:

- Автоматически реализуемые свойства
- Анонимные типы
- Выражения запросов
- Лямбда-выражения

- Деревья выражений
- Методы расширения
- Неявно типизированные локальные переменные
- Разделяемые методы
- Инициализаторы объектов и коллекций

Если оглянуться назад, становится ясно, что многие из этих возможностей не могли не появиться или появиться отдельно друг от друга. Все они образуют единый и стратегически значимый набор. Принято считать, что уникальным преимуществом C# было выражение запроса, которое также называется LINQ.

Немного углубившись, можно отметить деревья выражений, лямбда-выражения и анонимные типы как основу для создания LINQ. Однако в любом случае в C# 3.0 была представлена революционная концепция. Версия C# 3.0 начала закладывать фундамент для превращения C# в гибридный объектно-ориентированный/функциональный язык.

Среди прочего, теперь вы можете создавать декларативные запросы в стиле SQL для выполнения операций с коллекциями. Вместо создания цикла `for` для вычисления среднего значения из списка целых чисел теперь вам достаточно использовать `list.Average()`. Сочетание выражений запросов и методов расширения позволяет сделать работу со списком целых чисел гораздо интеллектуальнее.

Людям потребовалось некоторое время, чтобы принять эту концепцию и начать применять ее, но это у них получилось. А теперь, спустя несколько лет, код стал гораздо более кратким, простым и функциональным.

C# версии 4.0

Разработчикам версии C# 4.0, выпущенной вместе с Visual Studio 2010, пришлось приложить серьезные усилия, чтобы не отставать от новаторской версии 3.0. С появлением версии 3.0 язык C# смог выйти из тени Java и занять доминирующее положение. Он быстро становился утонченным и элегантным.

Следующая версия представила ряд интересных функций:

- Динамическая привязка
- Именованные/дополнительные аргументы
- Универсальная ковариантность и контравариантность
- Внедренные типы взаимодействия

Внедренные типы взаимодействия облегчили создание сборок COM-взаимодействия для приложения при развертывании. Универсальная ковариантность и контравариантность расширяют возможности применения универсальных шаблонов, однако эти функции несколько академичны и высоко оценены авторами, создающими платформы и библиотеки. Именованные и дополнительные параметры позволяют избавиться от многих перегрузок методов и сделать работу более удобной. Однако ни одна из этих функций не является кардинальным изменением.

Значимым событием стало появление ключевого слова `dynamic`. Благодаря ключевому слову `dynamic` в C# версии 4.0 появилась возможность переопределять компилятор при типизации во время компиляции. Используя это ключевое слово, вы можете создавать конструкции, характерные для языков с динамической типизацией, таких как JavaScript. Вы можете создать `dynamic x = "a string"` и добавить туда шестерку, отложив решение о дальнейших действиях во времени выполнения.

Динамическое связывание подвержено ошибкам, однако открывает великолепные возможности внутри языка.

C# версии 5.0

Версия C# 5.0, выпущенная вместе с Visual Studio 2012, была крайне целенаправленной. Практически все нововведения в этой версии касались другой инновационной концепции: модели `async` и `await` для асинхронного программирования. Ниже приведен список основных функций:

- Асинхронные члены
- Информационные атрибуты вызывающего объекта

См. также

- Code Project. Информационные атрибуты вызывающего объекта в C# 5.0

Информационный атрибут вызывающего объекта позволяет легко получать сведения о контексте, в котором выполняется работа, не прибегая к массивному стандартному коду отражения. Он находит обширное применение в задачах диагностики и ведения журнала.

Однако настоящими звездами этого выпуска являются `async` и `await`. После появления этих функций в 2012 г. C# удалось снова взять быка за рога, первоклассным образом внедрив асинхронность в язык. Если вы когда-либо сталкивались с длительными операциями и реализацией переплетающихся обратных вызовов, эта возможность, скорее всего, вам понравится.

C# версии 6.0

Версии 3.0 и 5.0 добавили в объектно-ориентированный язык C# несколько впечатляющих возможностей. Версия 6.0, выпущенная вместе с Visual Studio 2015, отступает от подхода, ориентированного на реализацию уникальных возможностей. Здесь предоставляются менее масштабные функции, с которыми программирование на C# становится гораздо эффективнее. Вот некоторые из них:

- Статические импорты
- Фильтры исключений
- Инициализаторы автосвойств
- Элементы, воплощающие выражение
- Null-распространитель
- Интерполяция строк
- Оператор nameof

Другие новые возможности:

- Инициализаторы индекса
- Выражение Await в блоках Catch и Finally
- Значения по умолчанию для свойств метода получения

Каждая из этих функций интересна сама по себе. Но если взглянуть на них вместе, заметна интересная особенность. В этой версии C# избавился от стандартного текста, чтобы сделать код более сжатым и удобочитаемым. Поэтому эта версия, несомненно, привлечет сторонников простого и понятного кода.

Еще одно новшество, связанное с этой версией, не является функцией языка в обычном понимании. Состоялся выпуск компилятора Roslyn в виде службы. Теперь компилятор C# написан на C#, и вы можете использовать его при программировании.

C# версии 7.0

C# версии 7.0 был выпущен вместе с Visual Studio 2017. В этой версии получили развитие некоторые функции из версии C# 6.0, однако компилятор в виде службы отсутствует. Ниже приведены некоторые из новых функций:

- Переменные Out

- Кортежи и деконструкция
- Сопоставление шаблонов
- Локальные функции
- Расширенные элементы, воплощающие выражение
- Локальные переменные и возвращаемые значения Ref

Другие возможности:

- Операции удаления
- Двоичные литералы и цифровые разделители
- Выражения throw

Все это предоставляет разработчиком множество новых возможностей и позволяет писать более понятный код. Важным аспектом является доработка объявления переменных путем использования ключевого слова `out` и поддержки нескольких возвращаемых значений благодаря кортежу.

Но при этом сфера применения C# продолжает расширяться. Теперь .NET Core подходит для любой операционной системы, а также уверенно движется в направлении облачных технологий и портативности. Очевидно, что разработчики активно занимаются этими новыми возможностями наравне с добавлением новых функций.

C# версии 7.1

C# начали выпускать *доработанные выпуски* с C# 7.1. В этой версии добавлен элемент конфигурации выбора версии языка, три новых возможности языка и новое поведение компилятора.

Новые языковые функции в этом выпуске

- Метод `async` `Main`
  - Точка входа для приложения может иметь модификатор `async`.
- Литеральные выражения `default`
  - Литеральные выражения по умолчанию можно использовать в выражениях значения по умолчанию, если можно вывести тип целевого объекта.
- Выводимые имена элементов кортежа
  - Имена элементов кортежа часто можно вывести из инициализации кортежа.
- Сопоставление шаблонов в параметрах универсального типа
  - Выражения сопоставления шаблонов можно использовать с переменными, тип которых является параметром универсального типа.

Наконец, у компилятора есть два параметра `-refout` и `-refonly`, которые управляют созданием базовой сборки.

C# версии 7.2

В C# 7.2 добавлено несколько небольших языковых функций:

- возможность использовать инициализаторы для массивов `stackalloc`;
- возможность использовать инструкции `fixed` с любым типом, который поддерживает шаблон;
- доступ к полям фиксированной ширины без закрепления;
- возможность переназначать локальные переменные `ref`;
- объявление типов `readonly struct`, указывающее, что структура является неизменяемой и должна передаваться в методы члена как параметр `in`;
- добавление модификатора `in` для параметров, указывающего, что аргумент передается по ссылке, но

не изменяется вызываемым методом;

- использование модификатора `ref readonly` для возвращаемого значения метода, указывающего, что метод возвращает значение по ссылке, но не допускает операции записи в соответствующий объект;
- объявление типов `ref struct`, указывающее, что тип структуры обращается напрямую к управляемой памяти и всегда должен обрабатываться с выделением стека;
- возможность использовать дополнительные универсальные ограничения.
- Неконечные именованные аргументы
  - После именованных аргументов могут следовать позиционные аргументы.
- Начальные символы подчеркивания в числовых литералах
  - Перед любыми печатными знаками в числовых литералах теперь могут использоваться начальные знаки подчеркивания.
- Модификатор доступа `private protected`
  - Модификатор доступа `private protected` разрешает доступ для производных классов в одной сборке.
- Условные выражения `ref`
  - Результат условного выражения `?:` теперь может быть ссылкой.

C# версии 7.3

Новые возможности в выпуске C# 7.3 можно разделить на две основные группы. Одна из них — набор функций для повышения эффективности безопасного кода до уровня небезопасного кода. Вторая — постепенные улучшения существующих функций. В этом выпуске добавлены новые параметры компилятора.

В ту группу, которая отвечает за повышение производительности безопасного кода, входят следующие новые возможности:

- доступ к полям фиксированной ширины без закрепления;
- возможность переназначать локальные переменные `ref`;
- возможность использовать инициализаторы для массивов `stackalloc`;
- возможность использовать инструкции `fixed` с любым типом, который поддерживает шаблон;
- Можно использовать более универсальные ограничения.

Для существующих функций предоставлены следующие улучшения:

- возможность проверить `==` и `!=` с типами кортежа;
- больше мест для использование переменных выражений;
- возможность подключить атрибуты к резервному полю автоматически реализуемых свойств;
- улучшенное разрешение методов, аргументы которых отличаются модификатором `in`;
- стало меньше неоднозначных вариантов при разрешении перегрузок.

Новые параметры компилятора:

- `-publicsign` позволяет включить подписывание сборок как программного обеспечения с открытым кодом;
- `-pathmap` позволяет предоставить сопоставление для исходных каталогов.

C# версии 8.0

C# 8.0 — это первый основной выпуск C#, специально предназначенный для .NET Core. Некоторые функции основаны на новых возможностях среды CLR, другие — на типах библиотек, добавленных только в .NET Core. В C# 8.0 добавлены следующие функции и улучшения языка C#:

- Члены только для чтения
- Методы интерфейса по умолчанию
- Улучшения сопоставления шаблонов:
  - выражения switch;
  - шаблоны свойств;
  - шаблоны кортежей;
  - позиционные шаблоны.
- Объявления using.
- Статические локальные функции.
- Удаляемые ссылочные структуры.
- Ссылочные типы, допускающие значение NULL
- Асинхронные потоки.
- Индексы и диапазоны.
- Присваивание объединения со значением NULL
- Неуправляемые сконструированные типы
- Выражение stackalloc во вложенных выражениях
- Улучшение интерполированных строк verbatim

Для членов интерфейса по умолчанию требуются улучшения среды CLR. Эти функции были добавлены в среду CLR для .NET Core 3.0. Для диапазонов и индексов, а также для асинхронных потоков требуются новые типы в библиотеках .NET Core 3.0. Ссылочные типы, допускающие значение NULL, реализованные в компиляторе, гораздо удобнее использовать, когда библиотеки снабжаются заметками для предоставления семантической информации о состоянии NULL аргументов и возвращаемых значений. Эти заметки сейчас добавляются в библиотеки .NET Core.

C# версии 9

C# версии 9 был выпущен с .NET 5. Это версия языка используется по умолчанию для любой сборки, предназначенной для выпуска .NET 5. Она содержит следующие новые и улучшенные функции:

- Записи
- Методы задания только инициализации
- Инструкции верхнего уровня
- Улучшения сопоставления шаблонов
- Производительность и взаимодействие
  - Целые числа собственного размера
  - Указатели функций
  - Отмена создания флага localsinit
- Функции подбора и завершения
  - Выражения `new` с целевым типом
  - Анонимные функции `static`
  - Условное выражение с целевым типом
  - Ковариантные возвращаемые типы
  - Поддержка расширения `GetEnumerator` для циклов `foreach`
  - Параметры удаления лямбда-выражения
  - Атрибуты локальных функций
- Поддержка генераторов кода
  - Инициализаторы модулей
  - Новые функции для разделяемых методов

В C# 9 получили развитие три направления предыдущих выпусков: устранение формальностей, отделение данных от алгоритмов и доступность большего количества шаблонов в большем количестве мест.

Операторы верхнего уровня упрощают чтение основной программы. Сократилась потребность в формальностях: больше не нужны ни пространства имен, `Program`, ни классы, ни `static void Main()`.

Внедрение `records` обеспечивает краткость синтаксиса для ссылочных типов, которые следуют семантике значений для равенства. Эти типы используются для определения контейнеров данных, которые обычно определяют минимальное поведение. Методы задания только инициализации предоставляют возможность для обратимого изменения (выражений `with`) в записях. В C# 9 также добавлены ковариантные возвращаемые типы, благодаря чему производные записи могут переопределять виртуальные методы и возвращать тип, производный от возвращаемого типа базового метода.

Возможности сопоставления шаблонов расширены в нескольких направлениях. Числовые типы теперь поддерживают *шаблоны диапазонов*. Шаблоны можно комбинировать с помощью шаблонов `and`, `or` и `not`. Для уточнения более сложных шаблонов можно добавить круглые скобки.

Высокопроизводительные вычисления на C# поддерживает еще один набор функций.

- Типы `nint` и `nuint` моделируют целочисленные типы собственного размера в целевом ЦП.
- Указатели функций предоставляют функциональные возможности, подобные возможностям делегатов, но при этом не требуют выделения, необходимого для создания объекта-делегата.
- Инструкцию `localsinit` можно опустить для сохранения инструкций.

Еще один набор улучшений поддерживает сценарии, в которых *генераторы кода* добавляют функциональные возможности.

- Инициализаторы модулей — это методы, которые среда выполнения вызывает при загрузке сборки.
- Разделяемые методы поддерживают новые модификаторы доступа и возвращаемые типы, отличные от void. В таких случаях необходимо предоставить реализацию.

В C# 9 добавлено множество других небольших функций, улучшающих производительность разработчиков, а также написание и чтение кода:

- Выражения `new` целевого типа
- Анонимные функции `static`
- Условные выражения целевого типа
- Поддержка расширения `GetEnumerator()` для циклов `foreach`
- Лямбда-выражения могут объявлять параметры отмены
- Атрибуты могут применяться к локальным функциям

Выпуск C# 9 — это следующий этап развития, благодаря которому C# остается современным языком программирования общего назначения. Его возможности обеспечат дальнейшую поддержку современных рабочих нагрузок и типов приложений.

Статья изначально опубликована в блоге NDepend, с разрешения Эрика Дитриха (Erik Dietrich) и Патрика Смачиа (Patrick Smacchia).

Связи между языковыми компонентами и типами библиотек

18.12.2021 • 2 minutes to read

Определение языка C# требует, чтобы стандартная библиотека имела определенные типы, а также определенные доступные члены этих типов. Компилятор создает код, использующий эти необходимые типы и члены для многих разных языковых компонентов. При необходимости можно использовать пакеты NuGet, которые содержат типы, необходимые для более новых версий языка при написании кода для сред, где эти типы или члены еще не развернуты.

Эта зависимость от функциональности стандартной библиотеки входила в состав языка C# с самой первой версии. В эту версию были включены следующие примеры:

- Exception — используется для всех исключений, создаваемых компилятором.
- String — тип C# `string` является синонимом String.
- Int32 — синоним `int`.

Эта первая версия была простой: компилятор и стандартная библиотека поставлялись вместе, и для каждого из них была только одна версия.

В последующих версиях C# в число зависимостей иногда добавлялись новые типы или члены. Примеры: INotifyCompletion, CallerFilePathAttribute и CallerMemberNameAttribute. В C# 7.0 этот процесс продолжается путем добавления зависимости от ValueTuple для реализации кортежей.

Группа разработчиков языка стремится минимизировать контактную зону типов и членов, необходимых в совместимой стандартной библиотеке. Эта задача уравновешивается потребностью в простой структуре, когда новые функции библиотек легко внедряются в язык. В будущих версиях C# появятся функции, требующие новых типов и членов в стандартной библиотеке. Важно понять, как управлять этими зависимостями в своей работе.

Управление зависимостями

Средства компилятора C# теперь отделены от цикла выпуска библиотек .NET на поддерживаемых платформах. В действительности различные библиотеки .NET имеют разные циклы выпуска: .NET Framework в Windows выпускается в Центре обновления Windows, .NET Core поставляется по отдельному расписанию, а версии библиотеки для Xamarin поставляются со средствами Xamarin для каждой целевой платформы.

Большую часть времени вы не будете замечать эти изменения. Однако при работе с более новой версией языка, которой нужны функции, пока отсутствующие в библиотеках .NET на данной платформе, вам потребуется сослаться на пакеты NuGet, чтобы предоставить эти новые типы. По мере обновления поддерживаемых вашим приложением платформ вы сможете удалить лишние ссылки.

Такое разделение означает, что даже ориентируясь на компьютеры, где может не быть соответствующей платформы, вы можете использовать новые возможности языка.

Соображения относительно версии и обновления для разработчиков на C#

18.12.2021 • 2 minutes to read

Совместимость очень важна при добавлении новых функций языка C#. В большинстве случаев существующий код сможет компилироваться в новой версии компилятора безо всяких проблем.

Особое внимание требуется в тех случаях, когда в библиотеку добавляются новые возможности языка. Например, если вы создаете новую библиотеку с функциями из последней версии, но ее должны использовать приложения, созданные в предыдущих версиях компилятора. Или же вы обновляете существующую библиотеку, но многие ваши пользователи пока используют старые версии. Принимая решения о внедрении новых функций, необходимо учитывать два аспекта совместимости: на уровне исходного кода и на уровне двоичных файлов.

Изменения с совместимостью на уровне двоичных файлов

Изменения библиотеки считаются **совместимыми на уровне двоичных файлов**, если обновленную библиотеку можно использовать без повторной сборки приложений и других использующих ее библиотек. В этом сценарии не нужно перестраивать зависимые сборки и (или) вносить изменения в исходный код.

Изменения с совместимостью на уровне исходного кода

Изменения библиотеки считаются **совместимыми на уровне исходного кода**, если для использующих ее приложений и библиотек никаких изменений исходного кода не требуется, однако для корректной работы необходима его перекомпиляция с новой версией.

Несовместимые изменения

Если изменение не является совместимым ни **на уровне исходного кода**, ни **на уровне двоичных файлов**, тогда для зависимых приложений и библиотек необходимо и изменить исходный код и выполнить перекомпиляцию.

Оценка существующих библиотек

Эти концепции совместимости влияют на компоненты библиотеки с атрибутами public и protected, но не на ее внутреннюю реализацию. Любые новые возможности внутри библиотеки всегда **совместимы на уровне двоичных файлов**.

Изменения, **совместимые на уровне двоичных файлов**, включают новый синтаксис, который при публичном объявлении будет выдавать такой же скомпилированный код, что и старый. Например, к изменению, имеющему **совместимость на уровне двоичных файлов**, относится замена метода на элемент, воплощающий выражение:

Исходный код:

```
public double CalculateSquare(double value)
{
    return value * value;
}
```

Новый код:

```
public double CalculateSquare(double value) => value * value;
```

Изменения, **совместимые на уровне исходного кода**, включают новый синтаксис, который выдает для публично объявляемых элементов новый скомпилированный код, но при этом сохраняет совместимость с ранее существовавшими вызывающими объектами. Например, если сигнатура метода ранее передавала параметр по значению, а теперь передает параметр по ссылке с помощью `in`, то такое изменение будет совместимым на уровне исходного кода, но не на уровне двоичных файлов.

Исходный код:

```
public double CalculateSquare(double value) => value * value;
```

Новый код:

```
public double CalculateSquare(in double value) => value * value;
```

В статьях о новых возможностях указывается, является ли новая функция, затрагивающая публичные объявления, совместимой на уровне исходного кода или на уровне двоичных файлов.

Создание типов записей

18.12.2021 • 10 minutes to read

В C# 9 появились *записи* — новый ссылочный тип, который можно создать вместо классов или структур. В C# 10 добавляются *структуры записей*, чтобы можно было определять записи в качестве типов значений. Записи отличаются от классов тем, что типы записей используют *равенство на основе значений*. Две переменные типа записи равны, если определения типов записей идентичны и если для каждого поля значения в обеих записях равны. Две переменные типа класса равны, если объекты, на которые они ссылаются, относятся к одному и тому же типу класса, а переменные ссылаются на один и тот же объект. Равенство на основе значений подразумевает другие полезные возможности. Компилятор создает многие из этих элементов при объявлении `record` вместо `class`. Компилятор создает те же методы для типов `record struct`.

Из этого руководства вы узнаете, как выполнять следующие задачи:

- Решите, следует ли объявлять `class` или `record`.
- Объявите типы записей и типы позиционных записей.
- Замените методы на созданные компилятором методы в записях.

Предварительные условия

Вам нужно настроить компьютер для выполнения .NET 6 или более поздней версии, включая компилятор C# 10 или более поздней версии. Компилятор C# 10 доступен начиная с Visual Studio 2022 или пакета SDK для .NET Core 6.

Характеристики записей

Вы определяете *запись*, объявляя тип с помощью ключевого слова `record` вместо ключевого слова `class` или `struct`. При необходимости можно объявить `record class`, чтобы уточнить, что это ссылочный тип. Запись является ссылочным типом и соответствует семантике равенства на основе значений. Можно определить `record struct`, чтобы создать запись, которая является типом значения. Чтобы обеспечить такую семантику, компилятор создает несколько методов для типа записи (для типов `record class` и `record struct`):

- Переопределение Object.Equals(Object).
- Виртуальный метод `Equals`, параметр которого является типом записи.
- Переопределение Object.GetHashCode().
- Методы для `operator ==` и `operator !=`.
- Типы записей реализуют System.IEquatable<T>.

Записи также предоставляют переопределение Object.ToString(). Компилятор синтезирует методы для отображения записей с помощью Object.ToString(). Эти элементы будут рассмотрены при написании кода для этого учебника. Записи поддерживают выражения `with`, позволяющие выполнять обратимое изменение записей.

Вы также можете объявить *позиционные записи*, используя более краткий синтаксис. При объявлении позиционных записей компилятор синтезирует дополнительные методы:

- Основной конструктор, параметры которого соответствуют позиционным параметрам в объявлении записи.

- Открытые свойства для каждого параметра основного конструктора. Эти свойства используются *только для инициализации* для типов `record class` и `readonly record struct`. Для типов `record struct` они доступны для *чтения и записи*.
- Метод `Deconstruct` для извлечения свойств из записи.

Создание данных о температуре

Данные и статистика представляют собой сценарии, в которых необходимо использовать записи. В этом учебнике вы создадите приложение, которое вычисляет *градусо-дни* для разных целей. *Градусо-дни* представляют собой меру тепла (или недостатка тепла) в течение нескольких дней, недель или месяцев. Градусо-дни позволяют отслеживать и прогнозировать использование энергии. Чем больше жарких дней, тем больше будет использоваться кондиционер, а чем больше холодных дней, тем больше будут использоваться обогреватели. Градусо-дни помогают управлять популяциями растений и учитывать рост растений по мере смены времен года. Градусо-дни помогают отслеживать миграцию животных в зависимости от климата.

Формула основана на средней температуре в определенный день и базовой температуре. Чтобы вычислить градусо-дни за период времени, вам нужно знать высокую и низкую температуру каждого дня этого периода. Давайте начнем с создания нового приложения. Создание нового консольного приложения. Создать новый тип записи в новом файле с именем DailyTemperature.cs:

```
public readonly record struct DailyTemperature(double HighTemp, double LowTemp);
```

В предыдущем коде определяется *позиционная запись*. Запись `DailyTemperature` является `readonly record struct`, так как вы не собираетесь наследовать от нее и она должна быть неизменной. Свойства `HighTemp` и `LowTemp` являются *свойствами только для инициализации*, то есть их можно задать в конструкторе или с помощью инициализатора свойств. Если нужно, чтобы позиционированные параметры были доступны для чтения и записи, объявите `record struct` вместо `readonly record struct`. Тип `DailyTemperature` также имеет *основной конструктор* с двумя параметрами, соответствующими двум свойствам. Основной конструктор используется для инициализации записи `DailyTemperature`:

```
private static DailyTemperature[] data = new DailyTemperature[]
{
    new DailyTemperature(HighTemp: 57, LowTemp: 30),
    new DailyTemperature(60, 35),
    new DailyTemperature(63, 33),
    new DailyTemperature(68, 29),
    new DailyTemperature(72, 47),
    new DailyTemperature(75, 55),
    new DailyTemperature(77, 55),
    new DailyTemperature(72, 58),
    new DailyTemperature(70, 47),
    new DailyTemperature(77, 59),
    new DailyTemperature(85, 65),
    new DailyTemperature(87, 65),
    new DailyTemperature(85, 72),
    new DailyTemperature(83, 68),
    new DailyTemperature(77, 65),
    new DailyTemperature(72, 58),
    new DailyTemperature(77, 55),
    new DailyTemperature(76, 53),
    new DailyTemperature(80, 60),
    new DailyTemperature(85, 66)
};
```

В записи можно добавлять собственные свойства или методы, включая позиционные записи. Необходимо

вычислить среднюю температуру каждого дня. Это свойство можно добавить в запись `DailyTemperature`:

```
public readonly record struct DailyTemperature(double HighTemp, double LowTemp)
{
    public double Mean => (HighTemp + LowTemp) / 2.0;
}
```

Давайте убедимся, что вы можете использовать эти данные. Добавьте приведенный ниже код в метод `Main`:

```
foreach (var item in data)
    Console.WriteLine(item);
```

Запустите приложение, и вы увидите результат, похожий на следующий (несколько строк удалено для экономии места):

```
DailyTemperature { HighTemp = 57, LowTemp = 30, Mean = 43.5 }
DailyTemperature { HighTemp = 60, LowTemp = 35, Mean = 47.5 }


DailyTemperature { HighTemp = 80, LowTemp = 60, Mean = 70 }
DailyTemperature { HighTemp = 85, LowTemp = 66, Mean = 75.5 }
```

В приведенном выше коде показаны выходные данные переопределения `ToString`, синтезированные компилятором. Если вы предпочитаете другой текст, то можете написать собственную версию `ToString`. При этом компилятор не будет синтезировать версию автоматически.

Вычисление градусо-дней

Чтобы вычислить градусо-дни, нужно взять разность между базовой температурой и средней температурой за определенный день. Чтобы измерить теплые дни за период времени, не учитывайте дни, когда средняя температура была ниже базовой. Чтобы измерить холодные дни за период времени, не учитывайте дни, когда средняя температура была выше базовой. Например, в США за основу берется 65 градусов по Фаренгейту. В эту температуру не требуется включать нагрев или охлаждение. Если средняя температура дня составляет 70 градусов по Фаренгейту, это 5 градусо-дней в плане охлаждения и 0 градусо-дней в плане обогрева. Если средняя температура дня составляет 55 градусов по Фаренгейту, это 0 градусо-дней в плане охлаждения и 10 градусо-дней в плане обогрева.

Эти формулы можно выразить как небольшую иерархию типов записей: абстрактный тип градусо-дней и два конкретных типа для градусо-дней обогрева и градусо-дней охлаждения. Эти типы также могут быть позиционными записями. Они принимают базовую температуру и последовательность ежедневных записей температуры в качестве аргументов для основного конструктора:

```
public abstract record DegreeDays(double BaseTemperature, IEnumerable<DailyTemperature> TempRecords);

public sealed record HeatingDegreeDays(double BaseTemperature, IEnumerable<DailyTemperature> TempRecords)
    : DegreeDays(BaseTemperature, TempRecords)
{
    public double DegreeDays => TempRecords.Where(s => s.Mean < BaseTemperature).Sum(s => BaseTemperature -
s.Mean);
}

public sealed record CoolingDegreeDays(double BaseTemperature, IEnumerable<DailyTemperature> TempRecords)
    : DegreeDays(BaseTemperature, TempRecords)
{
    public double DegreeDays => TempRecords.Where(s => s.Mean > BaseTemperature).Sum(s => s.Mean -
BaseTemperature);
}
```

Абстрактная запись `DegreeDays` является общим базовым классом для записей `HeatingDegreeDays` и `CoolingDegreeDays`. Объявления основного конструктора в производных записях показывают, как управлять инициализацией базовой записи. Ваша производная запись объявляет параметры для всех параметров в основном конструкторе базовой записи. Базовая запись объявляет и инициализирует эти свойства. Производная запись не скрывает их, но создает и инициализирует только свойства для параметров, которые не объявлены в базовой записи. В этом примере производные записи не добавляют новые параметры основного конструктора. Протестируйте код, добавив следующий код в метод `Main`:

```
var heatingDegreeDays = new HeatingDegreeDays(65, data);
Console.WriteLine(heatingDegreeDays);

var coolingDegreeDays = new CoolingDegreeDays(65, data);
Console.WriteLine(coolingDegreeDays);
```

Вы получите выходные данные, подобные следующим:

```
HeatingDegreeDays { BaseTemperature = 65, TempRecords = record_types.DailyTemperature[], DegreeDays = 85 }
CoolingDegreeDays { BaseTemperature = 65, TempRecords = record_types.DailyTemperature[], DegreeDays = 71.5 }
```

Определение методов, синтезируемых компилятором

В коде вычисляется правильное число градусо-дней обогрева и охлаждения за этот период времени. Но в этом примере показано, почему может потребоваться заменить некоторые синтезированные методы на записи. Вы можете объявить собственную версию любых синтезированных компилятором методов в типе записи, за исключением метода клонирования. Метод клонирования имеет имя, созданное компилятором, и вы не можете предоставить другую реализацию. Эти синтезированные методы включают конструктор копий, элементы интерфейса System.IEquatable<T>, проверки равенства и неравенства, а также GetHashCode(). Для этой цели вы создадите `PrintMembers`. Можно также объявить собственный `ToString`, но `PrintMembers` предоставляет лучший вариант для сценариев наследования. Чтобы предоставить собственную версию синтезированного метода, сигнатура должна соответствовать синтезированному методу.

Элемент `TempRecords` в выходных данных консоли не имеет смысла. Он отображает тип, но больше ничего. Это поведение можно изменить, предоставив собственную реализацию синтезированного метода `PrintMembers`. Сигнатура зависит от модификаторов, применяемых к объявлению `record`:

- Если тип записи — `sealed` или `record struct`, то сигнатура — `private bool PrintMembers(StringBuilder builder);`
- Если тип записи не `sealed` и является производным от `object` (то есть базовая запись не

объявляется), то сигнатура — `protected virtual bool PrintMembers(StringBuilder builder);`
- Если тип записи не `sealed` и является производным от другой записи, то сигнатура — `protected override bool PrintMembers(StringBuilder builder);`

Эти правила проще всего осмыслить, поняв цель `PrintMembers`. `PrintMembers` добавляет сведения о каждом свойстве в типе записи в строку. По контракту базовые записи должны добавлять свои элементы в отображение. Предполагается, что производные элементы будут добавлять свои элементы. Каждый тип записи синтезирует переопределение `ToString`, которое выглядит следующим образом для `HeatingDegreeDays`:

```
public override string ToString()
{
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.Append("HeatingDegreeDays");
    stringBuilder.Append(" { ");
    if (PrintMembers(stringBuilder))
    {
        stringBuilder.Append(" ");
    }
    stringBuilder.Append("}");
    return stringBuilder.ToString();
}
```

Вы объявляете метод `PrintMembers` в записи `DegreeDays`, которая не выводит тип коллекции:

```
protected virtual bool PrintMembers(StringBuilder stringBuilder)
{
    stringBuilder.Append($"BaseTemperature = {BaseTemperature}");
    return true;
}
```

Сигнатура объявляет метод `virtual protected`, чтобы обеспечить соответствие версии компилятора. Не беспокойтесь, если получаете неправильные методы доступа; язык применит правильную сигнатуру. Если вы забыли правильные модификаторы для синтезированного метода, компилятор выдает предупреждения или ошибки, которые помогут получить правильную сигнатуру.

В C# 10 и более поздних версий метод `ToString` можно объявить как `sealed` в типе записи. Это предотвращает предоставление новой реализации в производных записях. Производные записи по-прежнему будут содержать переопределение `PrintMembers`. Вы можете запечатать метод `ToString`, если вы не хотите, чтобы он отображал тип среды выполнения записи. В предыдущем примере теряются сведения о том, где выполнялось измерение градусо-дней нагрева или охлаждения.

Обратимое изменение

Синтезированные элементы в классе позиционной записи не изменяют состояние записи. Главная цель — упростить создание неизменяемых записей. Помните, что вы объявили `readonly record struct`, чтобы создать неизменяемую структуру записи. Просмотрите предыдущие объявления для `HeatingDegreeDays` и `CoolingDegreeDays`. Добавленные элементы выполняют вычисления со значениями для записи, но не изменяют состояние. Позиционные записи упрощают создание неизменяемых ссылочных типов.

Создание неизменяемых ссылочных типов означает, что необходимо использовать обратимое изменение. Вы создаете новые экземпляры записей, аналогичные существующим экземплярам записей, используя выражения `with`. Эти выражения являются конструкцией копии с дополнительными назначениями, которые изменяют копию. Результатом является новый экземпляр записи, где каждое свойство было скопировано из существующей записи и при необходимости изменено. Исходная запись не изменится.

Давайте добавим в программу несколько функций, демонстрирующих выражения `with`. Во-первых, создадим новую запись для вычислений рост градусо-дней, используя те же данные. *Рост градусо-дней* обычно использует 41 градус по Фаренгейту в качестве базового показателя и измеряет температуру выше базовой. Чтобы использовать те же данные, можно создать новую запись, похожую на `coolingDegreeDays`, но с другой базовой температурой:

```
// Growing degree days measure warming to determine plant growing rates
var growingDegreeDays = coolingDegreeDays with { BaseTemperature = 41 };
Console.WriteLine(growingDegreeDays);
```

Вы можете сравнить число рассчитанных градусов с числом, вычисленным при более высокой базовой температуре. Помните, что записи являются *ссылочными типами*, и эти копии являются поверхностными копиями. Массив данных не копируется, но обе записи ссылаются на одни и те же данные. Этот факт является преимуществом в еще одном сценарии. Для растущих градусо-дней полезно отслеживать общее число за предыдущие пять дней. С помощью выражений `with` можно создавать новые записи с разными исходными данными. Следующий код создает коллекцию этих сумм, а затем отображает значения:

```
// showing moving accumulation of 5 days using range syntax
List<CoolingDegreeDays> movingAccumulation = new();
int rangeSize = (data.Length > 5) ? 5 : data.Length;
for (int start = 0; start < data.Length - rangeSize; start++)
{
    var fiveDayTotal = growingDegreeDays with { TempRecords = data[start..(start + rangeSize)] };
    movingAccumulation.Add(fiveDayTotal);
}
Console.WriteLine();
Console.WriteLine("Total degree days in the last five days");
foreach(var item in movingAccumulation)
{
    Console.WriteLine(item);
}
```

Можно также использовать выражения `with` для создания копий записей. Не указывайте свойства между фигурными скобками в выражении `with`. Это означает, что нужно создать копию и не менять свойства:

```
var growingDegreeDaysCopy = growingDegreeDays with { };
```

Запустите готовое приложение, чтобы увидеть результаты.

Сводка

В этом учебнике мы рассмотрели несколько аспектов записей. Записи предоставляют краткий синтаксис для типов, где главная цель — хранение данных. Для объектно-ориентированных классов основным назначением является определение обязанностей. В этом учебнике вы узнали о *позиционных записях*, где можно использовать краткий синтаксис для объявления свойств записи. Компилятор синтезирует несколько элементов записи для копирования и сравнения записей. Вы можете добавить любые другие элементы, необходимые для ваших типов записей. Вы можете создавать неизменяемые типы записей, зная, что ни один из созданных компилятором элементов не изменит состояние. Выражения `with` упрощают поддержку обратимого изменения.

Записи добавляют еще один способ определения типов. Определения `class` используются для создания объектно-ориентированных иерархий, в которых основное внимание уделяется обязанностям и поведению объектов. Вы создаете типы `struct` для структур данных, которые хранят данные и

достаточно малы для эффективного копирования. Типы `record` создаются, когда требуется определить равенство и сравнение на основе значений, но не нужно копировать значения, а вы хотите использовать ссылочные переменные. Вы создаете типы `record struct`, если требуется, чтобы функции записей для типа были достаточно малы для эффективного копирования.

Полное описание записей можно узнать, прочитав справочную статью по языку C# для типа записи, предлагаемую спецификацию типа записи и спецификации структуры записи.

Учебник. Анализ идей с использованием операторов верхнего уровня для создания кода по мере обучения

18.12.2021 • 7 minutes to read

Из этого руководства вы узнаете, как выполнять следующие задачи:

- Ознакомитесь с правилами, касающимися использования операторов верхнего уровня.
- Используете операторы верхнего уровня для изучения алгоритмов.
- Выполните рефакторинг результатов исследования в многократно используемые компоненты.

Предварительные требования

Вам нужно настроить свой компьютер для выполнения .NET 6, включая компилятор C# 10. Компилятор C# 10 доступен начиная с Visual Studio 2022 или пакета SDK для .NET 6.

В этом руководстве предполагается, что вы знакомы с C# и .NET, включая Visual Studio или .NET CLI.

Начало знакомства

Операторы верхнего уровня позволяют избежать лишних формальностей благодаря помещению точки входа программы в статический метод в классе. Новое консольное приложение обычно начинается со следующего кода.

```
using System;

namespace Application
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Приведенный выше код является результатом выполнения команды `dotnet new console` для создания консольного приложения. Из этих одиннадцати строк кода только одна является исполняемой. Эту программу можно упростить с помощью новой функции, которая называется операторами верхнего уровня. Она позволяет удалить из этой программы все строки, кроме двух.

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

IMPORTANT

В шаблонах C# для .NET 6 используются *операторы верхнего уровня*. Ваше приложение может не соответствовать коду в этой статье, если вы уже выполнили обновление до предварительных версий .NET 6. Дополнительные сведения см. в статье Новые шаблоны C# для создания инструкций верхнего уровня.

Пакет SDK для .NET 6 также добавляет набор *неявных* директив `global using` для проектов, использующих следующие пакеты SDK:

- Microsoft.NET.Sdk
- Microsoft.NET.Sdk.Web
- Microsoft.NET.Sdk.Worker

Эти неявные директивы `global using` включают наиболее распространенные пространства имен для соответствующего типа проектов.

Эта функция упрощает изучение новых идей. Операторы верхнего уровня можно использовать для скриптов или исследования возможностей. Получив работающую базовую структуру, можно перейти к рефакторингу кода и создать методы, классы и другие сборки для многократно используемых компонентов. Операторы верхнего уровня позволяют быстро проводить эксперименты и реализовывать упражнения для начинающих. Они также позволяют легко переходить от экспериментов к полнофункциональным программам.

Операторы верхнего уровня выполняются в том порядке, в котором они следуют в файле. Их можно использовать только в одном исходном файле приложения. Если они используются более чем в одном файле, компилятор выдает ошибку.

Создание программы для ответа на вопросы на .NET

В этом учебнике мы создадим консольное приложение, которое отвечает на вопрос, требующий ответа "да" или "нет", случайным образом. Его функциональность будет реализована постепенно. Вы можете сосредоточиться на задаче, а не на формальностях, необходимых для построения структуры обычной программы. Когда нужная функциональность будет получена, вы можете выполнить рефакторинг приложения по своему усмотрению.

Начать можно с записи вопроса обратно в консоль. Для этого можно использовать следующий код.

```
Console.WriteLine(args);
```

Переменная `args` не объявляется. В случае с одним исходным файлом, содержащим операторы верхнего уровня, компилятор распознает `args` как аргументы командной строки. args имеет тип `string[]`, как во всех программах на C#.

Вы можете протестировать код, выполнив следующую команду `dotnet run`.

```
dotnet run -- Should I use top level statements in all my programs?
```

Аргументы после `--` в командной строке передаются в программу. Вы можете увидеть тип переменной `args`, так как в консоли выводится следующее.

```
System.String[]
```

Чтобы записать вопрос в консоль, необходимо перечислить аргументы через пробел. Замените вызов

`WriteLine` следующим кодом.

```
Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();
```

Теперь при запуске программы она будет правильно выводить вопрос в виде строки аргументов.

Вывод случайного ответа

После вывода вопроса можно добавить код для выбора случайного ответа. Сначала добавьте массив возможных ответов.

```
string[] answers =
{
    "It is certain.",       "Reply hazy, try again.",     "Don’t count on it.",
    "It is decidedly so.",  "Ask again later.",           "My reply is no.",
    "Without a doubt.",     "Better not tell you now.",   "My sources say no.",
    "Yes – definitely.",    "Cannot predict now.",        "Outlook not so good.",
    "You may rely on it.",  "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
};
```

Этот массив содержит десять утвердительных ответов, пять неопределенных и пять отрицательных. Далее добавьте следующий код для выбора случайного ответа из массива и его вывода.

```
var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

Вы можете снова запустить приложение, чтобы увидеть результаты. Выходные данные должны быть примерно следующими.

```
dotnet run -- Should I use top level statements in all my programs?

Should I use top level statements in all my programs?
Better not tell you now.
```

Этот код отвечает на вопросы, но давайте добавим еще одну функцию. Вы хотите, чтобы ваше приложение делало вид, будто думает над ответом. Для этого можно добавить ASCII-анимацию и сделать небольшую паузу. Добавьте следующий код после строки, которая выводит на экран вопрос.

```
for (int i = 0; i < 20; i++)
{
    Console.Write("| -");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("/ \\");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("- |");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("\\ /");
    await Task.Delay(50);
    Console.Write("\b\b\b");
}
Console.WriteLine();
```

Необходимо также добавить оператор `using` в начало исходного файла.

```
using System.Threading.Tasks;
```

Операторы `using` должны находиться перед всеми остальными операторами в файле. В противном случае возникает ошибка компилятора. Вы можете снова запустить программу и посмотреть анимацию. Так уже лучше. Поэкспериментируйте с длительностью задержки и выберите понравившуюся.

Приведенный выше код создает несколько вращающихся линий, разделенных пробелами. Добавление ключевого слова `await` предписывает компилятору создать точку входа программы в виде метода с модификатором `async` и возвращает System.Threading.Tasks.Task. Эта программа не возвращает значения, поэтому ее точка входа возвращает `Task`. Если программа возвращала бы целочисленное значение, необходимо было бы добавить оператор return в конце блока операторов верхнего уровня. Он указывал бы возвращаемое целочисленное значение. Если операторы верхнего уровня включают в себя выражение `await`, возвращается тип System.Threading.Tasks.Task<TResult>.

Рефакторинг кода для дальнейшего использования

Код программы должен выглядеть так.

```
Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();

for (int i = 0; i < 20; i++)
{
    Console.Write("| -");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("/ \\");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("- |");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("\\ /");
    await Task.Delay(50);
    Console.Write("\b\b\b");
}
Console.WriteLine();

string[] answers =
{
    "It is certain.",       "Reply hazy, try again.",     "Don't count on it.",
    "It is decidedly so.",  "Ask again later.",           "My reply is no.",
    "Without a doubt.",     "Better not tell you now.",   "My sources say no.",
    "Yes – definitely.",    "Cannot predict now.",        "Outlook not so good.",
    "You may rely on it.",  "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
};

var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

Приведенный выше код имеет смысл. Он работает. Но его нельзя использовать многократно. Теперь, когда вы получили работающее приложение, пора извлечь из него многократно используемые части.

Одним из кандидатов является код, отображающий анимацию во время ожидания. Этот фрагмент кода может стать методом.

Можно начать с создания локальной функции в файле. Замените текущую анимацию следующим кодом.

```
await ShowConsoleAnimation();

static async Task ShowConsoleAnimation()
{
    for (int i = 0; i < 20; i++)
    {
        Console.Write("| -");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("/ \\");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("- |");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("\\ /");
        await Task.Delay(50);
        Console.Write("\b\b\b");
    }
    Console.WriteLine();
}
```

Приведенный выше код создает локальную функцию внутри метода main. Его также нельзя использовать повторно. Поэтому давайте извлечем этот код в класс. Создайте файл с именем *utilities.cs* и добавьте следующий код.

```
namespace MyNamespace
{
    public static class Utilities
    {
        public static async Task ShowConsoleAnimation()
        {
            for (int i = 0; i < 20; i++)
            {
                Console.Write("| -");
                await Task.Delay(50);
                Console.Write("\b\b\b");
                Console.Write("/ \\");
                await Task.Delay(50);
                Console.Write("\b\b\b");
                Console.Write("- |");
                await Task.Delay(50);
                Console.Write("\b\b\b");
                Console.Write("\\ /");
                await Task.Delay(50);
                Console.Write("\b\b\b");
            }
            Console.WriteLine();
        }
    }
}
```

Файл, содержащий инструкции верхнего уровня, также может содержать пространства имен и типы в конце файла (после инструкций верхнего уровня). Но при работе с этим учебником метод анимации нужно поместить в отдельный файл, чтобы упростить его повторное использование.

Наконец, можно очистить код анимации, удалив повторяющиеся элементы.

```
foreach (string s in new[] { "| -", "/ \\", "- |", "\\ /", })
{
    Console.Write(s);
    await Task.Delay(50);
    Console.Write("\b\b\b");
}
```

Теперь у вас есть законченное приложение и вы выполнили рефакторинг частей, пригодных для многократного использования. Вы можете вызвать новый служебный метод с помощью инструкций верхнего уровня, как показано ниже в готовой версии основной программы:

```
using MyNamespace;

Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();

await Utilities.ShowConsoleAnimation();

string[] answers =
{
    "It is certain.",       "Reply hazy, try again.",     "Don't count on it.",
    "It is decidedly so.",  "Ask again later.",           "My reply is no.",
    "Without a doubt.",     "Better not tell you now.",   "My sources say no.",
    "Yes – definitely.",    "Cannot predict now.",        "Outlook not so good.",
    "You may rely on it.",  "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
};

var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

В приведенном выше примере добавляется вызов `Utilities.ShowConsoleAnimation` и дополнительная инструкция `using`.

Сводка

Операторы верхнего уровня упрощают создание простых программ для изучения новых алгоритмов. Вы можете экспериментировать с алгоритмами, используя разные фрагменты кода. Определив наиболее эффективный подход, можно выполнить рефакторинг кода, чтобы его было удобнее обслуживать.

Операторы верхнего уровня упрощают программы, основанные на консоли. К ним относятся функции Azure, действия GitHub и другие небольшие служебные программы. Дополнительные сведения см. в статье Инструкции верхнего уровня (руководство по программированию на C#).

Использование сопоставления шаблонов для создания поведения класса и улучшения кода

18.12.2021 • 9 minutes to read

Функции сопоставления шаблонов в C# предоставляют синтаксис для выражения алгоритмов. Эти методы можно использовать для реализации поведения в классах. Объектно-ориентированную архитектуру класса можно объединить с реализацией, ориентированной на данные, чтобы обеспечить лаконичный код при моделировании реальных объектов.

Из этого руководства вы узнаете, как выполнять следующие задачи:

- Выражать объектно-ориентированные классы с помощью шаблонов данных.
- Внедрять эти шаблоны, используя функции сопоставления шаблонов на C#.
- Использовать диагностику компилятора для проверки реализации.

Предварительные требования

Необходимо настроить компьютер для выполнения .NET 5, включая компилятор C# 9. Компилятор C# 9 доступен начиная с версии 16.8 Visual Studio 2019 или в пакете SDK для .NET 5.

Создание симуляции шлюза канала

В этом руководстве вы создадите класс C#, который симулирует шлюз канала. Шлюз — это устройство, которое поднимает и опускает судна, которые перемещаются между двумя водными пространствами на разных уровнях. Шлюз включает двое ворот и механизм для изменения уровня воды.

При эксплуатации в обычных условиях судно входит в одни из ворот, при этом уровень воды в шлюзе соответствует уровню воды на стороне входа судна. Когда судно находится в шлюзовой камере, уровень воды изменяется, чтобы соответствовать уровню воды камеры, из которой судно будет выходить. Как только уровень воды соответствует уровню воды камеры выхода, в камере выхода открываются ворота. Меры безопасности настроены таким образом, чтобы оператор не мог создать опасную ситуацию в шлюзе. Уровень воды изменяется только при закрытых воротах. Можно открыть только одни ворота. Чтобы открыть ворота, внутренний уровень воды в шлюзе должен соответствовать уровню воды на выходе.

Для моделирования такого поведения можно создать класс C#. Класс `CanalLock` будет поддерживать команды для открытия или закрытия обоих ворот. А также иметь разные команды для увеличения или уменьшения уровня воды. Класс также должен поддерживать свойства для чтения текущего состояния ворот и уровня воды. Ваши методы будут реализовать меры безопасности.

Определение класса

Вам нужно будет создать консольное приложение для тестирования класса `CanalLock`. Создайте новый консольный проект для .NET 5 с помощью Visual Studio или интерфейса командной строки .NET. Затем добавьте новый класс и назовите его `CanalLock`. Теперь создайте общедоступный API, однако не реализуйте методы:

```
public enum WaterLevel
{
    Low,
    High
}
public class CanalLock
{
    // Query canal lock state:
    public WaterLevel CanalLockWaterLevel { get; private set; } = WaterLevel.Low;
    public bool HighWaterGateOpen { get; private set; } = false;
    public bool LowWaterGateOpen { get; private set; } = false;

    // Change the upper gate.
    public void SetHighGate(bool open)
    {
        throw new NotImplementedException();
    }

    // Change the lower gate.
    public void SetLowGate(bool open)
    {
        throw new NotImplementedException();
    }

    // Change water level.
    public void SetWaterLevel(WaterLevel newLevel)
    {
        throw new NotImplementedException();
    }

    public override string ToString() =>
        $"The lower gate is {(LowWaterGateOpen ? "Open" : "Closed")}. " +
        $"The upper gate is {(HighWaterGateOpen ? "Open" : "Closed")}. " +
        $"The water level is {CanalLockWaterLevel}.";
}
```

Приведенный выше код инициализирует объект таким образом, чтобы и те и другие ворота были закрыты, а уровень воды оставался низким. Далее напишите следующий код проверки в методе `Main`, который поможет вам создать первую реализацию класса:

```
// Create a new canal lock:
var canalGate = new CanalLock();

// State should be doors closed, water level low:
Console.WriteLine(canalGate);

canalGate.SetLowGate(open: true);
Console.WriteLine($"Open the lower gate:  {canalGate}");

Console.WriteLine("Boat enters lock from lower gate");

canalGate.SetLowGate(open: false);
Console.WriteLine($"Close the lower gate:  {canalGate}");

canalGate.SetWaterLevel(WaterLevel.High);
Console.WriteLine($"Raise the water level: {canalGate}");
Console.WriteLine(canalGate);

canalGate.SetHighGate(open: true);
Console.WriteLine($"Open the higher gate:  {canalGate}");

Console.WriteLine("Boat exits lock at upper gate");
Console.WriteLine("Boat enters lock from upper gate");

canalGate.SetHighGate(open: false);
Console.WriteLine($"Close the higher gate: {canalGate}");

canalGate.SetWaterLevel(WaterLevel.Low);
Console.WriteLine($"Lower the water level: {canalGate}");

canalGate.SetLowGate(open: true);
Console.WriteLine($"Open the lower gate:  {canalGate}");

Console.WriteLine("Boat exits lock at upper gate");

canalGate.SetLowGate(open: false);
Console.WriteLine($"Close the lower gate:  {canalGate}");
```

Затем добавьте первую реализацию каждого метода в класс `CanalLock`. Следующий код реализует методы класса, не влияя на правила безопасности. Испытания на безопасность будут добавлены позже:

```
// Change the upper gate.
public void SetHighGate(bool open)
{
    HighWaterGateOpen = open;
}

// Change the lower gate.
public void SetLowGate(bool open)
{
    LowWaterGateOpen = open;
}

// Change water level.
public void SetWaterLevel(WaterLevel newLevel)
{
    CanalLockWaterLevel = newLevel;
}
```

Тесты, которые вы написали, прошли успешно. Вы реализовали основы. Теперь напишите тест для условия независимого отказа. По окончании предыдущих тестов одни и другие ворота закрыты, а уровень воды — низкий. Добавьте тест для открытия верхних ворот:

```
Console.WriteLine("=============================================");
Console.WriteLine("     Test invalid commands");
// Open "wrong" gate (2 tests)
try
{
    canalGate = new CanalLock();
    canalGate.SetHighGate(open: true);
}
catch (InvalidOperationException)
{
    Console.WriteLine("Invalid operation: Can't open the high gate. Water is low.");
}
Console.WriteLine($"Try to open upper gate: {canalGate}");
```

Тест завершается сбоем, поскольку ворота открываются. Для первой реализации внесите исправления, используя следующий код:

```
// Change the upper gate.
public void SetHighGate(bool open)
{
    if (open && (CanalLockWaterLevel == WaterLevel.High))
        HighWaterGateOpen = true;
    else if (open && (CanalLockWaterLevel == WaterLevel.Low))
        throw new InvalidOperationException("Cannot open high gate when the water is low");
}
```

Тесты успешно пройдены. При добавлении дополнительных тестов вы добавляете дополнительные `if` предложений и проверяете различные свойства. При добавлении дополнительных условий со временем методы становятся слишком сложными.

Реализация команд с помощью шаблонов

Лучшим решением станет использование *шаблонов*, чтобы определить, находится ли объект в допустимом состоянии для выполнения команды. Вы можете выразить разрешение на выполнение команды в качестве функции с тремя переменными: состояние ворот, уровень воды и новый параметр.

НОВЫЙ ПАРАМЕТР	СОСТОЯНИЕ ВОРОТ	УРОВЕНЬ ВОДЫ	РЕЗУЛЬТАТ
Закрыто	Закрыто	Высокий	Закрыто
Закрыто	Закрыто	Низкий	Закрыто
Закрыто	Открыть	Высокий	Открыть
~~Закрыто~~	~~Открыть~~	~~Низкая~~	~~Закрыто~~
Открыть	Закрыто	Высокий	Открыть
Открыть	Закрыто	Низкий	Закрыто (ошибка)
Открыть	Открыть	Высокий	Открыть
~~Открыть~~	~~Открыть~~	~~Низкая~~	~~Закрыто (ошибка)~~

Текст четвертой и последней строки в таблице зачеркнут, поскольку они недопустимы. Добавленный код должен гарантировать, что верхние ворота никогда не откроются при низком уровне воды. Эти

состояния можно запрограммировать как одиночное выражение switch (помните, что `false` означает "Закрыто"):

```
HighWaterGateOpen = (open, HighWaterGateOpen, CanalLockWaterLevel) switch
{
    (false, false, WaterLevel.High) => false,
    (false, false, WaterLevel.Low) => false,
    (false, true, WaterLevel.High) => false,
    (false, true, WaterLevel.Low) => false, // should never happen
    (true, false, WaterLevel.High) => true,
    (true, false, WaterLevel.Low) => throw new InvalidOperationException("Cannot open high gate when the
water is low"),
    (true, true, WaterLevel.High) => true,
    (true, true, WaterLevel.Low) => false, // should never happen
};
```

Попробуйте эту версию. Тесты пройдены, проверка кода. В полной таблице показаны возможные комбинации входных и выходных значений. Это означает, что вы и другие разработчики можете взглянуть на таблицу и проверить охват всех возможных входных данных. И даже проще — вам в этом может помочь компилятор. После добавления предыдущего кода можно увидеть, что компилятор выдает предупреждение: *CS8524* указывает, что выражение switch не охватывает все возможные входные данные. Причиной предупреждения является то, что некоторые входные данные имеют тип `enum`. Компилятор интерпретирует "все возможные входные данные" как все входные данные из базового типа, обычно это — `int`. Это выражение `switch` проверяет только значения, указанные в `enum`. Для устранения предупреждения можно добавить шаблон пустой переменной catch-all для последней ветви выражения. Это условие создает исключение, поскольку указывает на недопустимые входные данные:

```
_  => throw new InvalidOperationException("Invalid internal state"),
```

Предыдущая ветвь switch должна быть последней в выражении `switch`, так как она соответствует всем входным данным. Поэкспериментируйте, изменяя порядок. Это вызывает ошибку компилятора *CS8510* — недопустимый код в шаблоне. Естественная структура выражений switch позволяет компилятору создавать ошибки и предупреждения для потенциальных ошибок. Компилятор "безопасная сеть" упрощает создание чистого кода с меньшим количеством итераций, а также позволяет объединять ветви переключения с подстановочными знаками. Компилятор выдаст ошибки, если комбинация приведет к недопустимым ветвям, а также к предупреждениям при удалении необходимой ветви.

Первое изменение состоит в том, чтобы объединить все ветви, в которых командой является закрытие ворот; оно всегда разрешено. Добавьте следующий код в виде первой ветви в выражении switch.

```
(false, _, _) => false,
```

После добавления предыдущей ветви вы получите четыре ошибки компилятора, по одной на каждую из ветвей в команде `false`. Эти ветви уже охвачены недавно добавленной ветвью. И вы можете безопасно удалить эти четыре строки. Вы пытались заменить эти условия новой ветвью переключения.

Теперь можно упростить четыре ветви для команды открытия ворот. Ворота можно открыть в обоих случаях, если уровень воды высок. (В первом случае они уже открыты.) Для первого случая, при котором уровень воды низкий, должно появиться исключение, а для другого — нет. Если шлюз уже находится в недопустимом состоянии, для появления такого же исключения необходимы безопасные условия. Эти ветви можно упростить таким образом:

```
(true, _, WaterLevel.High) => true,
(true, false, WaterLevel.Low) => throw new InvalidOperationException("Cannot open high gate when the water
is low"),
_ => throw new InvalidOperationException("Invalid internal state"),
```

Запустите тесты еще раз, они должны пройти успешно. Финальная версия метода `SetHighGate` выглядит вот так:

```
// Change the upper gate.
public void SetHighGate(bool open)
{
    HighWaterGateOpen = (open, HighWaterGateOpen, CanalLockWaterLevel) switch
    {
        (false, _,    _)               => false,
        (true, _,     WaterLevel.High) => true,
        (true, false, WaterLevel.Low)  => throw new InvalidOperationException("Cannot open high gate when
the water is low"),
        _                              => throw new InvalidOperationException("Invalid internal state"),
    };
}
```

Самостоятельное внедрение шаблонов

Теперь, когда вы знакомы с методикой, заполните методы `SetLowGate` и `SetWaterLevel` самостоятельно. Начните с добавления следующего кода для проверки недопустимых операций для этих методов:

```
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetWaterLevel(WaterLevel.High);
    canalGate.SetLowGate(open: true);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't open the lower gate. Water is high.");
}
Console.WriteLine($"Try to open lower gate: {canalGate}");
// change water level with gate open (2 tests)
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetLowGate(open: true);
    canalGate.SetWaterLevel(WaterLevel.High);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't raise water when the lower gate is open.");
}
Console.WriteLine($"Try to raise water with lower gate open: {canalGate}");
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetWaterLevel(WaterLevel.High);
    canalGate.SetHighGate(open: true);
    canalGate.SetWaterLevel(WaterLevel.Low);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't lower water when the high gate is open.");
}
Console.WriteLine($"Try to lower water with high gate open: {canalGate}");
```

Запустите приложение еще раз. Видно, что новые тесты завершаются ошибкой, а шлюз переходит в недопустимое состояние. Попробуйте реализовать оставшиеся методы самостоятельно. Метод для нижних ворот должен быть идентичен методу для верхних ворот. Метод для изменения уровня воды имеет другие проверки, однако должен иметь ту же структуру. Вы можете использовать тот же процесс для метода, который задает уровень воды. Начните со всех четырех входных данных: состояние обоих ворот, текущий уровень воды и необходимый новый уровень воды. Выражение switch должно начинаться с:

```
CanalLockWaterLevel = (newLevel, CanalLockWaterLevel, LowWaterGateOpen, HighWaterGateOpen) switch
{
    // elided
};
```

Вы получите 16 ветвей переключения, которые нужно заполнить данными. Проверьте и упростите их.

Получилось ли у вас что-то наподобие этого?

```
// Change the lower gate.
public void SetLowGate(bool open)
{
    LowWaterGateOpen = (open, LowWaterGateOpen, CanalLockWaterLevel) switch
    {
        (false, _, _) => false,
        (true, _, WaterLevel.Low) => true,
        (true, false, WaterLevel.High) => throw new InvalidOperationException("Cannot open high gate when
the water is low"),
        _ => throw new InvalidOperationException("Invalid internal state"),
    };
}

// Change water level.
public void SetWaterLevel(WaterLevel newLevel)
{
    CanalLockWaterLevel = (newLevel, CanalLockWaterLevel, LowWaterGateOpen, HighWaterGateOpen) switch
    {
        (WaterLevel.Low, WaterLevel.Low, true, false) => WaterLevel.Low,
        (WaterLevel.High, WaterLevel.High, false, true) => WaterLevel.High,
        (WaterLevel.Low, _, false, false) => WaterLevel.Low,
        (WaterLevel.High, _, false, false) => WaterLevel.High,
        (WaterLevel.Low, WaterLevel.High, false, true) => throw new InvalidOperationException("Cannot lower
water when the high gate is open"),
        (WaterLevel.High, WaterLevel.Low, true, false) => throw new InvalidOperationException("Cannot raise
water when the low gate is open"),
        _ => throw new InvalidOperationException("Invalid internal state"),
    };
}
```

Тесты должны успешно завершиться, а шлюз — работать безопасно.

Итоги

Из этого руководства вы узнали, как использовать сопоставление шаблонов для проверки внутреннего состояния объекта, прежде чем изменять состояние. Вы можете проверять сочетания свойств. После создания таблиц для любого из этих переходов необходимо протестировать код, а затем упростить его для более удобного чтения и сопровождения. Эти изначальные рефакторинги могут рекомендовать другие рефакторинги, которые проверяют внутреннее состояние или управляют другими изменениями API. В этом учебнике были объединены классы и объекты с более ориентированным на данные подходом, в котором для реализации этих классов используются шаблоны.

Учебник. Обновление интерфейсов с помощью методов интерфейса по умолчанию в C# 8.0

18.12.2021 • 5 minutes to read

Начиная с C# 8.0 в .NET Core 3.0 можно определить реализацию при объявлении члена интерфейса. Наиболее распространенным сценарием является безопасное добавление членов в интерфейс, который уже выпущен и используется многочисленными клиентами.

В этом руководстве вы узнаете, как:

- Безопасно расширять интерфейсы, добавляя методы с реализациями.
- Создавать параметризованные реализации, которые обеспечивают большую гибкость.
- Позволить разработчику реализовать более конкретную реализацию в виде переопределения.

Предварительные требования

Вам нужно настроить свой компьютер для выполнения .NET Core, включая компилятор C# 8.0. Компилятор C# 8.0 доступен, начиная с версии 16.3 Visual Studio 2019, или в пакете SDK .NET Core 3.0.

Обзор сценария

Этот учебник начинается с первой версии библиотеки связи с клиентом. Начальное приложение можно получить в нашем репозитории примеров на сайте GitHub. Компания, которая создала эту библиотеку, рассчитывала, что клиенты с существующими приложениями будут ее внедрять. Она определила минимальный интерфейс для реализации библиотеки пользователями. Вот определение интерфейса для клиента:

```
public interface ICustomer
{
    IEnumerable<IOrder> PreviousOrders { get; }

    DateTime DateJoined { get; }
    DateTime? LastOrder { get; }
    string Name { get; }
    IDictionary<DateTime, string> Reminders { get; }
}
```

Компания определила второй интерфейс, представляющий заказ:

```
public interface IOrder
{
    DateTime Purchased { get; }
    decimal Cost { get; }
}
```

На основе этих интерфейсов команда может собрать библиотеку для удобства работы клиентов своих пользователей. Их целью было более полно взаимодействовать с существующими клиентами и повысить уровень связи с новыми.

Пришло время перейти к библиотеке для следующего выпуска. Одна из востребованных возможностей — добавление скидки за лояльность для клиентов, размещающих большое количество заказов. Это

новая скидка лояльности применяется каждый раз, когда клиент делает заказ. Специальная скидка является свойством каждого клиента. Каждая реализация `ICustomer` может задавать разные правила для скидки лояльности.

Наиболее удобный способ добавления этой функции — расширить `ICustomer` методом для применения любых скидок лояльности. Это предложение по разработке вызвало проблемы среди опытных разработчиков. "Интерфейсы являются неизменяемыми после выпуска! Это критическое изменение!" В C# 8.0 добавлены *реализации интерфейсов по умолчанию* для обновления интерфейсов. Авторы библиотеки могут добавлять новые элементы интерфейса и реализации по умолчанию для этих элементов.

Реализации интерфейса по умолчанию позволяют разработчикам обновить интерфейс, по-прежнему позволяя другим разработчикам переопределять эту реализацию. Пользователи библиотеки могут принимать реализацию по умолчанию в качестве некритического изменения. Если их бизнес-правила не совпадают, их можно переопределить.

Обновление с методами интерфейса по умолчанию

Команда пришла к выводу относительно реализации по умолчанию: скидки за лояльность для клиентов.

Обновление должно давать возможность задать два свойства: количество заказов, необходимое, чтобы иметь право на скидку, а также процент скидки. В результате мы получаем идеальный сценарий для применения методов интерфейса по умолчанию. Можно добавить метод в интерфейс `ICustomer` и предоставить наиболее вероятную реализацию. Все существующие и любые новые реализации могут использовать реализацию по умолчанию или предоставить свои собственные.

Сначала добавьте в интерфейс новый метод, включая тело метода:

```
// Version 1:
public decimal ComputeLoyaltyDiscount()
{
    DateTime TwoYearsAgo = DateTime.Now.AddYears(-2);
    if ((DateJoined < TwoYearsAgo) && (PreviousOrders.Count() > 10))
    {
        return 0.10m;
    }
    return 0;
}
```

Автор библиотеки написал первый тест для проверки реализации:

```
SampleCustomer c = new SampleCustomer("customer one", new DateTime(2010, 5, 31))
{
    Reminders =
    {
        { new DateTime(2010, 08, 12), "childs's birthday" },
        { new DateTime(1012, 11, 15), "anniversary" }
    }
};

SampleOrder o = new SampleOrder(new DateTime(2012, 6, 1), 5m);
c.AddOrder(o);

o = new SampleOrder(new DateTime(2103, 7, 4), 25m);
c.AddOrder(o);

// Check the discount:
ICustomer theCustomer = c;
Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}");
```

Обратите внимание на следующую часть теста:

```
// Check the discount:
ICustomer theCustomer = c;
Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}");
```

Приведение `SampleCustomer` к `ICustomer` необходимо. Классу `SampleCustomer` не требуется предоставлять реализацию для `ComputeLoyaltyDiscount`; она предоставляется интерфейсом `ICustomer`. Тем не менее класс `SampleCustomer` не наследует члены от своих интерфейсов. Это правило не изменилось. Чтобы вызвать любой метод, который был объявлен и реализован в интерфейсе, переменная должна иметь тип интерфейса (`ICustomer` в этом примере).

Задайте параметризацию

Это хорошее начало. Однако реализация по умолчанию слишком строга. Другие пользователи этой системы могут выбрать различные пороговые значения для числа покупок, разную длительность членства или другой процент скидки. Удобство работы обновления можно повысить для нескольких клиентов сразу, предоставляя возможность установить эти параметры. Давайте добавим статический метод, который задает эти три параметра, контролируя реализацию по умолчанию:

```
// Version 2:
public static void SetLoyaltyThresholds(
    TimeSpan ago,
    int minimumOrders = 10,
    decimal percentageDiscount = 0.10m)
{
    length = ago;
    orderCount = minimumOrders;
    discountPercent = percentageDiscount;
}
private static TimeSpan length = new TimeSpan(365 * 2, 0,0,0); // two years
private static int orderCount = 10;
private static decimal discountPercent = 0.10m;

public decimal ComputeLoyaltyDiscount()
{
    DateTime start = DateTime.Now - length;

    if ((DateJoined < start) && (PreviousOrders.Count() > orderCount))
    {
        return discountPercent;
    }
    return 0;
}
```

В этом небольшом фрагменте кода показано много новых языковых возможностей. Интерфейсы теперь могут содержать статические члены, включая поля и методы. Также включены разные модификаторы доступа. Дополнительные поля являются закрытыми, новый метод является открытым. Все эти модификаторы разрешены для членов интерфейса.

Приложениям, использующим общую формулу для вычисления скидок лояльности с разными параметрами, не нужно создавать собственные реализации; достаточно задать аргументы с помощью статического метода. Например, следующий код задает "повышение уровня клиента", в рамках которого награждается любой клиент более чем с месячным сроком членства:

```
ICustomer.SetLoyaltyThresholds(new TimeSpan(30, 0, 0, 0), 1, 0.25m);
Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}");
```

Расширение реализации по умолчанию

Код, который вы добавили на данный момент, предоставляет удобную реализацию для сценариев, где нужны что-то наподобие реализации по умолчанию или несвязанный набор правил. Чтобы дополнить код, давайте проведем рефакторинг, чтобы учесть сценарии, где пользователям нужно расширить реализацию по умолчанию.

Предположим, стартап хочет привлечь новых клиентов. Они предоставляют скидку в 50 % на первый заказ нового клиента. В противном случае существующие клиенты получают стандартные скидки. Автору библиотеки необходимо перенести реализацию по умолчанию в метод `protected static`, чтобы любой класс, реализующий этот интерфейс, мог повторно использовать код в своей реализации. По умолчанию реализация члена интерфейса также вызывает этот общий метод:

```
public decimal ComputeLoyaltyDiscount() => DefaultLoyaltyDiscount(this);
protected static decimal DefaultLoyaltyDiscount(ICustomer c)
{
    DateTime start = DateTime.Now - length;

    if ((c.DateJoined < start) && (c.PreviousOrders.Count() > orderCount))
    {
        return discountPercent;
    }
    return 0;
}
```

В реализации класса, реализующего этот интерфейс, переопределение может вызывать статический вспомогательный метод и расширить эту логику для предоставления скидки новым клиентам:

```
public decimal ComputeLoyaltyDiscount()
{
   if (PreviousOrders.Any() == false)
        return 0.50m;
    else
        return ICustomer.DefaultLoyaltyDiscount(this);
}
```

Весь готовый код доступен в репозитории примеров на GitHub. Начальное приложение можно получить в нашем репозитории примеров на сайте GitHub.

Эти новые функции означают, что интерфейсы могут обновляться безопасно при наличии разумной реализации по умолчанию для этих новых элементов. Тщательно проектируйте интерфейсы для выражения единой функциональной идеи, которую могут реализовывать несколько классов. Это упрощает обновление этих определений интерфейса при обнаружении новых требований для этой же функциональности.

Учебник. Функциональные возможности смешения при создании классов с помощью методов интерфейса по умолчанию

18.12.2021 • 8 minutes to read

Начиная с C# 8.0 в .NET Core 3.0 можно определить реализацию при объявлении члена интерфейса. Эта функция предоставляет новые возможности, позволяющие определить реализации по умолчанию для компонентов, объявленных в интерфейсах. Классы могут выбирать, когда следует переопределять функциональность, когда следует использовать функциональную возможность по умолчанию и когда не следует объявлять поддержку отдельных функций.

В этом руководстве вы узнаете, как:

- Создать интерфейсы с реализациями, описывающими отдельные функции.
- Создать классы, которые используют реализации по умолчанию.
- Создать классы, которые переопределяют некоторые или все реализации по умолчанию.

Предварительные требования

Вам нужно настроить свой компьютер для выполнения .NET Core, включая компилятор C# 8.0. Компилятор C# 8.0 доступен в Visual Studio 2019 версии 16.3 и выше или пакете SDK .NET Core 3.0 и выше.

Ограничения методов расширения

Одним из способов реализации поведения, проявляемого в рамках интерфейса, является определение методов расширения, которые обеспечивают поведение по умолчанию. Интерфейсы объявляют минимальный набор элементов, предоставляя большую контактную зону для любого класса, реализующего этот интерфейс. Например, методы расширения в Enumerable обеспечивают реализацию любой последовательности в качестве источника запроса LINQ.

Методы расширения разрешаются с использованием объявленного типа переменной во время компиляции. Реализующие интерфейс классы могут обеспечить лучшую реализацию для любого метода расширения. Объявления переменных должны соответствовать реализующему типу, чтобы позволить компилятору выбрать эту реализацию. Если тип во время компиляции соответствует интерфейсу, метод вызывает разрешение для метода расширения. Другая проблема с методами расширения заключается в том, что эти методы доступны везде, где доступен класс, содержащий методы расширения. Классы не могут объявлять, должны они или не должны предоставлять функции, объявленные в методах расширения.

Начиная с C# 8.0, можно объявить реализации по умолчанию как методы интерфейса. Так каждый класс автоматически использует реализацию по умолчанию. Классы, обеспечивающие лучшую реализацию, могут переопределить определение метода интерфейса с помощью более эффективного алгоритма. В каком-то смысле этот прием напоминает то, как можно использовать метод расширения.

В этой статье вы узнаете, как реализации интерфейса по умолчанию позволяют поддерживать новые сценарии.

Разработка приложения

Рассмотрим приложение для системы домашней автоматики. Возможно, у вас есть много типов разного освещения и индикаторов, которые можно использовать во всем доме. Каждый источник освещения должен поддерживать интерфейсы API, позволяющие включать и выключать эти источники, а также передавать данные о текущем состоянии. Некоторые источники освещения и индикаторы могут поддерживать другие функции, такие как:

- включение освещения, а затем его отключение по таймеру;
- мигание освещения в течение определенного периода времени.

Некоторые из этих расширенных возможностей можно эмулировать на устройствах, поддерживающих минимальный набор. Это соответствует реализации по умолчанию. Для устройств со встроенными дополнительными возможностями программное обеспечение устройства будет использовать собственные возможности. Для других источников освещения они могут реализовать интерфейс и использовать реализацию по умолчанию.

Элементы интерфейса по умолчанию являются лучшим решением для этого сценария, чем методы расширения. Авторы классов могут управлять выбором интерфейсов для реализации. Выбранные ими интерфейсы доступны как методы. Кроме того, так как методы стандартного интерфейса являются виртуальными по умолчанию, диспетчеризация методов всегда выбирает реализацию в классе.

Давайте создадим код для демонстрации этих различий.

Создание интерфейсов

Сначала можно создать интерфейс, который определяет поведение для всех источников освещения.

```
public interface ILight
{
    void SwitchOn();
    void SwitchOff();
    bool IsOn();
}
```

Основное средство тестирования верхнего освещения может реализовать этот интерфейс, как показано в следующем коде.

```
public class OverheadLight : ILight
{
    private bool isOn;
    public bool IsOn() => isOn;
    public void SwitchOff() => isOn = false;
    public void SwitchOn() => isOn = true;

    public override string ToString() => $"The light is {(isOn ? "on" : "off")}";
}
```

В этом учебнике код не поддерживает устройства Интернета вещей, но эмулирует такие действия, записывая сообщения в консоль. Вы можете исследовать код без автоматизации дома.

Теперь определим интерфейс, автоматически отключающий освещение по истечении времени ожидания.

```
public interface ITimerLight : ILight
{
    Task TurnOnFor(int duration);
}
```

Можно добавить базовую реализацию к верхнему освещению, но лучшим решением является изменение этого определения интерфейса для предоставления реализации по умолчанию типа `virtual`.

```
public interface ITimerLight : ILight
{
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Using the default interface method for the ITimerLight.TurnOnFor.");
        SwitchOn();
        await Task.Delay(duration);
        SwitchOff();
        Console.WriteLine("Completed ITimerLight.TurnOnFor sequence.");
    }
}
```

После добавления этого изменения класс `OverheadLight` может реализовать функцию таймера, объявляя поддержку интерфейса.

```
public class OverheadLight : ITimerLight { }
```

Другой тип освещения может поддерживать более сложный протокол. Он может предоставить собственную реализацию для `TurnOnFor`, как показано в следующем коде.

```
public class HalogenLight : ITimerLight
{
    private enum HalogenLightState
    {
        Off,
        On,
        TimerModeOn
    }

    private HalogenLightState state;
    public void SwitchOn() => state = HalogenLightState.On;
    public void SwitchOff() => state = HalogenLightState.Off;
    public bool IsOn() => state != HalogenLightState.Off;
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Halogen light starting timer function.");
        state = HalogenLightState.TimerModeOn;
        await Task.Delay(duration);
        state = HalogenLightState.Off;
        Console.WriteLine("Halogen light finished custom timer function");
    }

    public override string ToString() => $"The light is {state}";
}
```

В отличие от переопределения виртуальных методов классов, объявление `TurnOnFor` в классе `HalogenLight` не использует ключевое слово `override`.

Смешение и сопоставление возможностей

Преимущества методов интерфейса по умолчанию становятся понятнее, когда вы добавляете расширенные возможности. Использование интерфейса позволяет смешивать и сопоставлять возможности. Это также позволяет каждому автору класса выбирать между реализацией по умолчанию и пользовательской реализацией. Давайте добавим интерфейс с реализацией по умолчанию для мигающего освещения.

```
public interface IBlinkingLight : ILight
{
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("Using the default interface method for IBlinkingLight.Blink.");
        for (int count = 0; count < repeatCount; count++)
        {
            SwitchOn();
            await Task.Delay(duration);
            SwitchOff();
            await Task.Delay(duration);
        }
        Console.WriteLine("Done with the default interface method for IBlinkingLight.Blink.");
    }
}
```

Реализация по умолчанию позволяет освещению мигать. С помощью реализации по умолчанию к верхнему освещению можно добавить возможности таймера и мигания.

```
public class OverheadLight : ILight, ITimerLight, IBlinkingLight
{
    private bool isOn;
    public bool IsOn() => isOn;
    public void SwitchOff() => isOn = false;
    public void SwitchOn() => isOn = true;

    public override string ToString() => $"The light is {(isOn ? "on" : "off")}";
}
```

Новый тип освещения `LEDLight` поддерживает функцию таймера и функцию мигания напрямую. Такой стиль освещения реализует интерфейсы `ITimerLight` и `IBlinkingLight`, а также переопределяет метод `Blink`.

```
public class LEDLight : IBlinkingLight, ITimerLight, ILight
{
    private bool isOn;
    public void SwitchOn() => isOn = true;
    public void SwitchOff() => isOn = false;
    public bool IsOn() => isOn;
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("LED Light starting the Blink function.");
        await Task.Delay(duration * repeatCount);
        Console.WriteLine("LED Light has finished the Blink funtion.");
    }

    public override string ToString() => $"The light is {(isOn ? "on" : "off")}";
}
```

`ExtraFancyLight` может напрямую поддерживать функции мигания и таймера.

```
public class ExtraFancyLight : IBlinkingLight, ITimerLight, ILight
{
    private bool isOn;
    public void SwitchOn() => isOn = true;
    public void SwitchOff() => isOn = false;
    public bool IsOn() => isOn;
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("Extra Fancy Light starting the Blink function.");
        await Task.Delay(duration * repeatCount);
        Console.WriteLine("Extra Fancy Light has finished the Blink function.");
    }
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Extra Fancy light starting timer function.");
        await Task.Delay(duration);
        Console.WriteLine("Extra Fancy light finished custom timer function");
    }

    public override string ToString() => $"The light is {(isOn ? "on" : "off")}";
}
```

Ранее созданный класс `HalogenLight` не поддерживает мигание. Поэтому не добавляйте `IBlinkingLight` в список его поддерживаемых интерфейсов.

Определение типов освещения с помощью сопоставления шаблонов

Теперь давайте напишем тестовый код. С помощью функции сопоставления шаблонов в C# можно определить возможности освещения путем проверки поддерживаемых им интерфейсов. Следующий метод выполняет поддерживаемые возможности для каждого источника освещения.

```
private static async Task TestLightCapabilities(ILight light)
{
    // Perform basic tests:
    light.SwitchOn();
    Console.WriteLine($"\tAfter switching on, the light is {(light.IsOn() ? "on" : "off")}");
    light.SwitchOff();
    Console.WriteLine($"\tAfter switching off, the light is {(light.IsOn() ? "on" : "off")}");

    if (light is ITimerLight timer)
    {
        Console.WriteLine("\tTesting timer function");
        await timer.TurnOnFor(1000);
        Console.WriteLine("\tTimer function completed");
    }
    else
    {
        Console.WriteLine("\tTimer function not supported.");
    }

    if (light is IBlinkingLight blinker)
    {
        Console.WriteLine("\tTesting blinking function");
        await blinker.Blink(500, 5);
        Console.WriteLine("\tBlink function completed");
    }
    else
    {
        Console.WriteLine("\tBlink function not supported.");
    }
}
```

Следующий код в методе `Main` последовательно создает каждый тип освещения и тестирует его.

```
static async Task Main(string[] args)
{
    Console.WriteLine("Testing the overhead light");
    var overhead = new OverheadLight();
    await TestLightCapabilities(overhead);
    Console.WriteLine();

    Console.WriteLine("Testing the halogen light");
    var halogen = new HalogenLight();
    await TestLightCapabilities(halogen);
    Console.WriteLine();

    Console.WriteLine("Testing the LED light");
    var led = new LEDLight();
    await TestLightCapabilities(led);
    Console.WriteLine();

    Console.WriteLine("Testing the fancy light");
    var fancy = new ExtraFancyLight();
    await TestLightCapabilities(fancy);
    Console.WriteLine();
}
```

Как компилятор определяет наилучшую реализацию

В этом сценарии показан базовый интерфейс без каких-либо реализаций. Добавление метода в интерфейс `ILight` вводит новые сложности. Языковые правила, управляющие методами интерфейса по умолчанию, снижают влияние на конкретные классы, которые реализуют несколько производных интерфейсов. Попробуем улучшить исходный интерфейс с помощью нового метода, чтобы продемонстрировать, как это изменит использование интерфейса. Каждый световой индикатор может сообщать о своем состоянии питания в виде перечисленного значения.

```
public enum PowerStatus
{
    NoPower,
    ACPower,
    FullBattery,
    MidBattery,
    LowBattery
}
```

Реализация по умолчанию предполагает отключение питания от сети.

```
public interface ILight
{
    void SwitchOn();
    void SwitchOff();
    bool IsOn();
    public PowerStatus Power() => PowerStatus.NoPower;
}
```

Эти изменения компилируются правильно, несмотря на то что `ExtraFancyLight` объявляет поддержку интерфейса `ILight` и производных интерфейсов `ITimerLight` и `IBlinkingLight`. В интерфейсе `ILight` объявлена только одна "ближайшая" реализация. Любой класс, который объявляет переопределение, стает "ближайшей" реализацией. Вы видели примеры с предыдущими классами, которые переопределили элементы других производных интерфейсов.

Старайтесь не переопределять один и тот же метод в нескольких производных интерфейсах. При этом создается неоднозначный вызов метода каждый раз, когда класс реализует оба производных интерфейса. Компилятор не может выбрать один лучший метод, поэтому выдает ошибку. Например, если и `IBlinkingLight`, и `ITimerLight` реализовали переопределение `PowerStatus`, то для `OverheadLight` потребуется предоставить более конкретное переопределение. В противном случае компилятор не сможет выбрать между реализациями двух производных интерфейсов. Обычно эту ситуацию можно избежать, сохраняя определения интерфейсов компактными и направленными на одну функцию. В этом сценарии каждая возможность освещения является собственным интерфейсом. Несколько интерфейсов наследуются только классами.

В этом примере показан один из сценариев, в котором можно определить отдельные функции, которые можно комбинировать в классы. Вы объявили любой набор поддерживаемых функциональных возможностей, объявляя интерфейсы, поддерживаемые классом. Виртуальные методы интерфейса по умолчанию позволяют классам использовать или определять различные реализации для любого или всех методов интерфейса. Такая возможность языка предоставляет новые способы для моделирования реальных систем, которые вы создаете. Методы интерфейса по умолчанию предоставляют более понятный способ для выражения связанных классов, которые могут смешивать и сопоставлять различные возможности с помощью их виртуальных реализаций.

Индексы и диапазоны

18.12.2021 • 5 minutes to read

Диапазоны и индексы обеспечивают лаконичный синтаксис для доступа к отдельным элементам или диапазонам в последовательности.

В этом руководстве вы узнаете, как:

- Использовать этот синтаксис для диапазонов в последовательности.
- Проектировать начало и конец каждой последовательности.
- Составлять сценарии для типов Index и Range.

Поддержка языков для индексов и диапазонов

Поддержка языков опирается на два новых типа и два новых оператора:

- System.Index представляет индекс в последовательности.
- Оператор `^` (индекс с конца), который указывает, что индекс указан относительно конца последовательности.
- System.Range представляет вложенный диапазон последовательности.
- Оператор диапазона `..`, который задает начало и конец диапазона в качестве своих операндов.

Начнем с правил для использования в индексах. Рассмотрим массив `sequence`. Индекс `0` совпадает с `sequence[0]`. Индекс `^0` совпадает с `sequence[sequence.Length]`. Выражение `sequence[^0]` создает исключение так же, как и `sequence[sequence.Length]`. Для любого числа `n` индекс `^n` совпадает с `sequence[sequence.Length - n]`.

```
string[] words = new string[]
{
                // index from start    index from end
    "The",      // 0                   ^9
    "quick",    // 1                   ^8
    "brown",    // 2                   ^7
    "fox",      // 3                   ^6
    "jumped",   // 4                   ^5
    "over",     // 5                   ^4
    "the",      // 6                   ^3
    "lazy",     // 7                   ^2
    "dog"       // 8                   ^1
};              // 9 (or words.Length) ^0
```

Последнее слово можно получить с помощью индекса `^1`. Добавьте следующий код после инициализации:

```
Console.WriteLine($"The last word is {words[^1]}");
```

Диапазон указывает *начало* и *конец* диапазона. Диапазоны являются открытыми, то есть *конечное значение* не включается в диапазон. Диапазон `[0..^0]` представляет весь диапазон так же, как `[0..sequence.Length]` представляет весь диапазон.

Следующий код создает поддиапазон со словами "quick", "brown" и "fox". Он включает в себя элементы от `words[1]` до `words[3]`. Элемент `words[4]` в диапазон не входит. Добавьте в тот же метод указанный

ниже код. Скопируйте и вставьте этот код в нижней части интерактивного окна.

```
string[] quickBrownFox = words[1..4];
foreach (var word in quickBrownFox)
    Console.Write($"< {word} >");
Console.WriteLine();
```

Следующий код возвращает диапазон со словами "lazy" и "dog". Он включает элементы `words[^2]` и `words[^1]`. Конечный индекс `words[^0]` не включен. Добавьте также следующий код:

```
string[] lazyDog = words[^2..^0];
foreach (var word in lazyDog)
    Console.Write($"< {word} >");
Console.WriteLine();
```

В следующих примерах создаются диапазоны, которые должны быть открыты для начала, конца или в обоих случаях:

```
string[] allWords = words[..]; // contains "The" through "dog".
string[] firstPhrase = words[..4]; // contains "The" through "fox"
string[] lastPhrase = words[6..]; // contains "the, "lazy" and "dog"
foreach (var word in allWords)
    Console.Write($"< {word} >");
Console.WriteLine();
foreach (var word in firstPhrase)
    Console.Write($"< {word} >");
Console.WriteLine();
foreach (var word in lastPhrase)
    Console.Write($"< {word} >");
Console.WriteLine();
```

Диапазоны или индексы можно также объявлять как переменные. Впоследствии такую переменную можно использовать внутри символов `[` и `]`:

```
Index the = ^3;
Console.WriteLine(words[the]);
Range phrase = 1..4;
string[] text = words[phrase];
foreach (var word in text)
    Console.Write($"< {word} >");
Console.WriteLine();
```

Следующий пример демонстрирует многие возможные причины для таких решений. Измените `x`, `y` и `z` и опробуйте различные комбинации. Поэкспериментируйте со значениями, у которых `x` меньше `y`, а `y` меньше, чем `z` для допустимых сочетаний. Добавьте в новый метод приведенный ниже код. Опробуйте различные комбинации:

```
int[] numbers = Enumerable.Range(0, 100).ToArray();
int x = 12;
int y = 25;
int z = 36;

Console.WriteLine($"{numbers[^x]} is the same as {numbers[numbers.Length - x]}");
Console.WriteLine($"{numbers[x..y].Length} is the same as {y - x}");

Console.WriteLine("numbers[x..y] and numbers[y..z] are consecutive and disjoint:");
Span<int> x_y = numbers[x..y];
Span<int> y_z = numbers[y..z];
Console.WriteLine($"\tnumbers[x..y] is {x_y[0]} through {x_y[^1]}, numbers[y..z] is {y_z[0]} through
{y_z[^1]}");

Console.WriteLine("numbers[x..^x] removes x elements at each end:");
Span<int> x_x = numbers[x..^x];
Console.WriteLine($"\tnumbers[x..^x] starts with {x_x[0]} and ends with {x_x[^1]}");

Console.WriteLine("numbers[..x] means numbers[0..x] and numbers[x..] means numbers[x..^0]");
Span<int> start_x = numbers[..x];
Span<int> zero_x = numbers[0..x];
Console.WriteLine($"\t{start_x[0]}..{start_x[^1]} is the same as {zero_x[0]}..{zero_x[^1]}");
Span<int> z_end = numbers[z..];
Span<int> z_zero = numbers[z..^0];
Console.WriteLine($"\t{z_end[0]}..{z_end[^1]} is the same as {z_zero[0]}..{z_zero[^1]}");
```

Поддержка типов для индексов и диапазонов

Индексы и диапазоны предоставляют четкий и краткий синтаксис для доступа к одному элементу или диапазону элементов последовательности. Выражение индекса обычно возвращает тип элементов последовательности. Выражение диапазона обычно возвращает тот же тип последовательности, что и исходная последовательность.

Любой тип, который предоставляет индексатор с параметром Index или Range, явно поддерживает индексы или диапазоны соответственно. Индексатор, который принимает один параметр Range, может возвращать другой тип последовательности, например System.Span<T>.

> **IMPORTANT**
>
> Производительность кода, использующего оператор range, зависит от типа операнда последовательности.
>
> Временная сложность оператора range зависит от типа последовательности. Например, если последовательность является `string` или массивом, то результатом будет копия указанного раздела входных данных, поэтому временная сложность имеет значение *O(N)* (где N — это длина диапазона). С другой стороны, если это System.Span<T> или System.Memory<T>, результат ссылается на то же резервное хранилище, что означает отсутствие копии и использование операции *O(1)* .
>
> Помимо временной сложности, создаются дополнительные выделения и копии, влияющие на производительность. В коде, чувствительном к производительности, рекомендуется использовать `Span<T>` или `Memory<T>` в качестве типа последовательности, так как оператор range не предоставляет им выделений.

Тип является **счетным**, если у него есть свойство с именем `Length` или `Count` с доступным методом получения и типом возвращаемого значения `int`. Счетный тип, который не поддерживает индексы или диапазоны явным образом, может предоставить неявную поддержку для них. Дополнительные сведения см. в разделах о поддержке неявного индекса и неявного диапазона в примечании к предлагаемой функции. Диапазоны с поддержкой неявных диапазонов возвращают тот же тип последовательности, что и исходная.

Например, следующие типы .NET поддерживают как индексы, так и диапазоны: String, Span<T> и

ReadOnlySpan<T>. List<T> поддерживает индексы, но не поддерживает диапазоны.

Array реализует более сложное поведение. Одномерные массивы поддерживают как индексы, так и диапазоны. Многомерные массивы не поддерживают индексаторы или диапазоны. Индексатор для многомерного массива имеет не один, а несколько параметров. Массивы массивов также поддерживают и диапазоны и индексаторы. В следующем примере показано выполнение итерации для прямоугольного подраздела массива массивов. В этом примере выполняется итерация раздела в центре с исключением трех первых и последних строк, а также двух первых и последних столбцов из каждой выбранной строки:

```
var jagged = new int[10][]
{
   new int[10] { 0,  1, 2, 3, 4, 5, 6, 7, 8, 9},
   new int[10] { 10,11,12,13,14,15,16,17,18,19},
   new int[10] { 20,21,22,23,24,25,26,27,28,29},
   new int[10] { 30,31,32,33,34,35,36,37,38,39},
   new int[10] { 40,41,42,43,44,45,46,47,48,49},
   new int[10] { 50,51,52,53,54,55,56,57,58,59},
   new int[10] { 60,61,62,63,64,65,66,67,68,69},
   new int[10] { 70,71,72,73,74,75,76,77,78,79},
   new int[10] { 80,81,82,83,84,85,86,87,88,89},
   new int[10] { 90,91,92,93,94,95,96,97,98,99},
};

var selectedRows = jagged[3..^3];

foreach (var row in selectedRows)
{
    var selectedColumns = row[2..^2];
    foreach (var cell in selectedColumns)
    {
        Console.Write($"{cell}, ");
    }
    Console.WriteLine();
}
```

Во всех случаях оператор range для Array выделяет массив для хранения возвращаемых элементов.

Сценарии для индексов и диапазонов

Часто диапазоны и индексы используются для анализа отдельной части последовательности большего размера. В новом синтаксисе легче понять, о какой части последовательности идет речь. Локальная функция `MovingAverage` принимает Range в качестве аргумента. После этого метод перечисляет только это диапазон и вычисляет минимальное, максимальное и среднее значение. Попробуйте добавить в свой проект следующий код:

```
int[] sequence = Sequence(1000);

for(int start = 0; start < sequence.Length; start += 100)
{
    Range r = start..(start+10);
    var (min, max, average) = MovingAverage(sequence, r);
    Console.WriteLine($"From {r.Start} to {r.End}:    \tMin: {min},\tMax: {max},\tAverage: {average}");
}

for (int start = 0; start < sequence.Length; start += 100)
{
    Range r = ^(start + 10)..^start;
    var (min, max, average) = MovingAverage(sequence, r);
    Console.WriteLine($"From {r.Start} to {r.End}:  \tMin: {min},\tMax: {max},\tAverage: {average}");
}

(int min, int max, double average) MovingAverage(int[] subSequence, Range range) =>
    (
        subSequence[range].Min(),
        subSequence[range].Max(),
        subSequence[range].Average()
    );

int[] Sequence(int count) =>
    Enumerable.Range(0, count).Select(x => (int)(Math.Sqrt(x) * 100)).ToArray();
```

Учебник. Четкое выражение проектного замысла с помощью ссылочных типов, допускающих и не допускающих значение null

18.12.2021 • 9 minutes to read

В C# 8.0 вводятся ссылочные типы, допускающие значение NULL, которые дополняют ссылочные типы таким же образом, как типы значений, допускающие значение NULL, дополняют другие типы значений. Чтобы объявить переменную как имеющую **ссылочный тип, допускающий значение null**, добавьте `?` к типу. Например, `string?` соответствует типу `string`, допускающему значение null. Эти новые типы можно использовать для более четкого выражения проектного замысла: некоторые переменные *всегда должны содержать значение*, а в других *они могут отсутствовать*.

В этом руководстве вы узнаете, как:

- Внедрять в проекты ссылочные типы, допускающие и не допускающие значение null.
- Включать в коде проверку ссылочных типов, допускающих значение null.
- Писать код, в котором компилятор принудительно применяет эти проектные решения.
- Использовать ссылочный тип, допускающий значение null, в собственных проектах.

Предварительные требования

Вам нужно настроить свой компьютер для выполнения .NET Core, включая компилятор C# 8.0. Компилятор C# 8.0 доступен в Visual Studio 2019 или .NET Core 3.0.

В этом руководстве предполагается, что вы знакомы с C# и .NET, включая Visual Studio или .NET CLI.

Внедрение в проекты ссылочных типов, допускающих значение null

В этом руководстве вы создадите библиотеку, моделирующую выполнение опроса. Код использует оба ссылочных типа (допускающий и не допускающий значение null) для представления реальных понятий. Вопросы в опросе никогда не могут принимать значение null. Респондент может предпочесть не отвечать на вопрос. В этом случае ответы могут принимать значение `null`.

Код, который вы напишете для этого примера, выражает это намерение, а компилятор применяет его.

Создание приложения и включение ссылочных типов, допускающих значение null

Создайте новое консольное приложение в Visual Studio или из командной строки с помощью `dotnet new console`. Присвойте приложению имя `NullableIntroduction`. Создав приложение, вам нужно указать, что весь проект компилируется с включенным **контекстом заметок, допускающим значение NULL**. Откройте файл *.csproj* и добавьте элемент `Nullable` в элемент `PropertyGroup`. Задайте для него значение `enable`. Необходимо выбрать функцию **ссылочных типов, допускающих значение NULL**, даже в проектах C# 8.0. Это связано с тем, что когда эта функция будет включена, существующие объявления ссылочных переменных становятся **ссылочными типами, допускающими значение null**. Хотя это решение поможет найти проблемы, где существующий код может не иметь правильных проверок нулевых значений, оно может не точно отражать исходное намерение проекта.

```
<Nullable>enable</Nullable>
```

В версиях, предшествующих .NET 6, новые проекты не включают элемент `Nullable`. Начиная с .NET 6 все файлы новых проектов содержат элемент `<Nullable>enable</Nullable>`.

Проектирование типов для приложения

Для этого приложения опроса требуется создать ряд классов:

- Класс, моделирующий список вопросов.
- Класс, моделирующий список людей, с которыми связались для опроса.
- Класс, моделирующий ответы человека, принявшего участие в опросе.

Эти типы будут использовать ссылочные типы, допускающие значения null и не допускающие значения null, чтобы выразить, какие элементы являются обязательными, а какие — необязательными. Ссылочные типы, допускающие значение null, четко указывают на это намерение проекта:

- Вопросы, входящие в опрос, не могут иметь значение null: Нет смысла задавать пустой вопрос.
- Респонденты никогда не могут принимать значение null. Следует отслеживать людей, с которыми вы связались, включая респондентов, которые отказались участвовать.
- Любой ответ на вопрос может принимать значение null. Респонденты могут отказаться отвечать на некоторые или все вопросы.

Если вы программировали на C#, возможно, вы так привыкли ссылаться на типы, допускающие значение `null`, что упустили другие возможности для объявления экземпляров, не допускающих значение null:

- Набор вопросов не должен допускать значение null.
- Набор ответов не должен допускать значение null.

При написании кода вы увидите, что если ссылочный тип, не допускающий значение null, является типом по умолчанию для ссылок, это позволяет избежать распространенных ошибок, которые могут привести к исключениям из-за пустых ссылок (NullReferenceException). В этом руководстве описана важность принятия решений о том, какие переменные могут или не могут принимать значение `null`. Язык не предоставлял синтаксис для выражения этих решений, а теперь предоставляет.

Приложение, которое вы создадите, будет выполнять следующие действия:

1. Создание опроса и добавление в него вопросов.
2. Создание псевдослучайного набора респондентов для опроса.
3. Связь с респондентами, пока размер опроса не достигает целевого значения.
4. Запись важных статистических данных на основе ответов на опрос.

Создание опроса с типами, допускающими и не допускающими значение NULL

Первый код, который вы напишете, создает опрос. Вы напишете классы для моделирования вопроса и выполнения опроса. Опрос состоит из вопросов трех типов, различающихся форматом ответов: да или нет, числовые ответы и текстовые ответы. Создайте класс `public SurveyQuestion`:

```
namespace NullableIntroduction
{
    public class SurveyQuestion
    {
    }
}
```

Компилятор интерпретирует каждое объявление переменной ссылочного типа как ссылочный тип, **не допускающий значение** NULL, для кода во включенном контексте заметок, допускающих значение NULL. Первое предупреждение вы увидите после добавления свойств текста и типа вопроса, как показано в следующем коде:

```
namespace NullableIntroduction
{
    public enum QuestionType
    {
        YesNo,
        Number,
        Text
    }

    public class SurveyQuestion
    {
        public string QuestionText { get; }
        public QuestionType TypeOfQuestion { get; }
    }
}
```

Так как вы еще не инициализировали `QuestionText`, компилятор выдает предупреждение о том, что свойство, не допускающее значение null, не инициализировано. Для проекта требуется, чтобы текст вопроса не принимал значение null, поэтому необходимо добавить конструктор, чтобы инициализировать свойство и значение `QuestionType`. Законченное определение класса выглядит следующим образом:

```
namespace NullableIntroduction
{
    public enum QuestionType
    {
        YesNo,
        Number,
        Text
    }

    public class SurveyQuestion
    {
        public string QuestionText { get; }
        public QuestionType TypeOfQuestion { get; }

        public SurveyQuestion(QuestionType typeOfQuestion, string text) =>
            (TypeOfQuestion, QuestionText) = (typeOfQuestion, text);
    }
}
```

После добавления конструктора предупреждение удаляется. Аргумент конструктора также является ссылочным типом, не допускающим значение null, поэтому компилятор не выдает никаких предупреждений.

Затем создайте класс `public` с именем `SurveyRun`. Этот класс содержит список объектов `SurveyQuestion` и методов для добавления вопросов в опрос, как показано в следующем коде:

```
using System.Collections.Generic;

namespace NullableIntroduction
{
    public class SurveyRun
    {
        private List<SurveyQuestion> surveyQuestions = new List<SurveyQuestion>();

        public void AddQuestion(QuestionType type, string question) =>
            AddQuestion(new SurveyQuestion(type, question));
        public void AddQuestion(SurveyQuestion surveyQuestion) => surveyQuestions.Add(surveyQuestion);
    }
}
```

Как и раньше, необходимо инициализировать объект списка с помощью значения, отличного от null, или компилятор выдаст предупреждение. При второй перегрузке `AddQuestion` не будет проверок значений null, так как они не требуются: вы объявили, что переменная не допускает значение null. Это значение не может быть равно `null`.

Переключитесь на *Program.cs* в редакторе и замените содержимое метода `Main` следующими строками кода:

```
var surveyRun = new SurveyRun();
surveyRun.AddQuestion(QuestionType.YesNo, "Has your code ever thrown a NullReferenceException?");
surveyRun.AddQuestion(new SurveyQuestion(QuestionType.Number, "How many times (to the nearest 100) has that
happened?"));
surveyRun.AddQuestion(QuestionType.Text, "What is your favorite color?");
```

Так как весь проект находится во включенном контексте заметок, допускающих значение NULL, при каждой передаче значения `null` в любой метод, который ожидает ссылочный тип, не допускающий значение NULL, будет отображаться предупреждение. Попробуйте добавить следующую строку в `Main`:

```
surveyRun.AddQuestion(QuestionType.Text, default);
```

Создание респондентов и получение ответов на опрос

Теперь напишите код, который генерирует ответы на опрос. Этот процесс включает в себя несколько мелких задач:

1. Создать метод, генерирующий объекты-респонденты. Они представляют людей, которых просят пройти опрос.
2. Создать логику, чтобы имитировать процесс задания вопросов респонденту и собирать ответы или отмечать, что респондент не ответил.
3. Повторять опрос до тех пор, пока нужное количество респондентов не ответит на вопросы.

Вам понадобится класс для представления ответа на опрос, поэтому добавьте его сейчас. Включите поддержку значений null. Добавьте свойство `Id` и конструктор, который его инициализирует, как показано в следующем коде:

```
namespace NullableIntroduction
{
    public class SurveyResponse
    {
        public int Id { get; }

        public SurveyResponse(int id) => Id = id;
    }
}
```

Затем добавьте метод `static` для создания новых участников путем генерации случайного идентификатора:

```
private static readonly Random randomGenerator = new Random();
public static SurveyResponse GetRandomId() => new SurveyResponse(randomGenerator.Next());
```

Основная задача этого класса состоит в том, чтобы генерировать ответы участников на вопросы в опросе. Эта задача состоит из нескольких шагов:

1. Запросите участие в опросе. Если пользователь не согласен, верните отсутствующий (или null) ответ.
2. Задайте каждый вопрос и запишите ответы. Любой ответ может отсутствовать (или иметь значение null).

Добавьте приведенный ниже код к классу `SurveyResponse`:

```
private Dictionary<int, string>? surveyResponses;
public bool AnswerSurvey(IEnumerable<SurveyQuestion> questions)
{
    if (ConsentToSurvey())
    {
        surveyResponses = new Dictionary<int, string>();
        int index = 0;
        foreach (var question in questions)
        {
            var answer = GenerateAnswer(question);
            if (answer != null)
            {
                surveyResponses.Add(index, answer);
            }
            index++;
        }
    }
    return surveyResponses != null;
}

private bool ConsentToSurvey() => randomGenerator.Next(0, 2) == 1;

private string? GenerateAnswer(SurveyQuestion question)
{
    switch (question.TypeOfQuestion)
    {
        case QuestionType.YesNo:
            int n = randomGenerator.Next(-1, 2);
            return (n == -1) ? default : (n == 0) ? "No" : "Yes";
        case QuestionType.Number:
            n = randomGenerator.Next(-30, 101);
            return (n < 0) ? default : n.ToString();
        case QuestionType.Text:
        default:
            switch (randomGenerator.Next(0, 5))
            {
                case 0:
                    return default;
                case 1:
                    return "Red";
                case 2:
                    return "Green";
                case 3:
                    return "Blue";
            }
            return "Red. No, Green. Wait.. Blue... AAARGGGGGHHH!";
    }
}
```

Хранилищем для ответов на опрос является `Dictionary<int, string>?`, из которого видно, что оно может принимать значение null. Используйте новую языковую функцию, чтобы объявить намерение проекта как компилятору, так и всем, кто будет работать с кодом позже. Если вы когда-нибудь разыменуете `surveyResponses`, не проверив значения `null`, компилятор отобразит предупреждение. В методе `AnswerSurvey` предупреждение не отображается, так как компилятор может определить, что переменной `surveyResponses` ранее было присвоено значение, отличное от null.

Использование значения `null` для отсутствующих ответов указывает на важную особенность при работе с допускающими значение NULL ссылочными типами: перед вами не стоит цель удалить все значения `null` из программы. Скорее, вам нужно привести написанный вами код в соответствие со своим замыслом. Отсутствующие значения — это неотъемлемая часть кода. И значение `null` хорошо подходит для их выражения. Если вы попытаетесь удалить все значения `null`, вам придется определить другой способ, чтобы выразить такие отсутствующие значения без использования `null`.

Далее необходимо написать метод `PerformSurvey` в классе `SurveyRun`. Добавьте в класс `SurveyRun` следующий код:

```
private List<SurveyResponse>? respondents;
public void PerformSurvey(int numberOfRespondents)
{
    int respondentsConsenting = 0;
    respondents = new List<SurveyResponse>();
    while (respondentsConsenting < numberOfRespondents)
    {
        var respondent = SurveyResponse.GetRandomId();
        if (respondent.AnswerSurvey(surveyQuestions))
            respondentsConsenting++;
        respondents.Add(respondent);
    }
}
```

В этом случае выбор `List<SurveyResponse>?` с возможностью принимать значение null указывает на то, что ответ может отсутствовать. Это значит, что респонденты еще не участвовали в опросе. Обратите внимание, что респонденты добавляются до тех пор, пока не будет достигнуто целевое значение.

Для запуска опроса осталось добавить вызов в конце метода `Main`:

```
surveyRun.PerformSurvey(50);
```

Анализ ответов на опросы

Далее следует отобразить ответы на опрос. Добавьте код во многие из написанных вами классов. Этот код демонстрирует различие ссылочных типов, допускающих и не допускающих значение null. Начните с добавления элементов, воплощающих выражение, в класс `SurveyResponse`:

```
public bool AnsweredSurvey => surveyResponses != null;
public string Answer(int index) => surveyResponses?.GetValueOrDefault(index) ?? "No answer";
```

Так как `surveyResponses` является ссылочным типом, допускающим значение null, прежде чем отменять ссылку на него, укажите, что проверка требуется. Метод `Answer` возвращает строку, не допускающую значения null, поэтому мы должны охватить вариант отсутствующего ответа, используя оператор объединения со значением null.

Затем добавьте эти три элемента, воплощающие выражение, в класс `SurveyRun`:

```
public IEnumerable<SurveyResponse> AllParticipants => (respondents ?? Enumerable.Empty<SurveyResponse>());
public ICollection<SurveyQuestion> Questions => surveyQuestions;
public SurveyQuestion GetQuestion(int index) => surveyQuestions[index];
```

Элемент `AllParticipants` должен принимать во внимание, что переменная `respondents` может иметь значение null, но возвращаемое значение не может быть равно null. Если изменить это выражение, удалив `??` и следующую пустую последовательность, компилятор выдаст предупреждение о том, что метод может возвращать `null`, а его сигнатура возвращает тип, не допускающий значение null.

Наконец, добавьте следующий цикл в нижней части метода `Main`:

```
foreach (var participant in surveyRun.AllParticipants)
{
    Console.WriteLine($"Participant: {participant.Id}:");
    if (participant.AnsweredSurvey)
    {
        for (int i = 0; i < surveyRun.Questions.Count; i++)
        {
            var answer = participant.Answer(i);
            Console.WriteLine($"\t{surveyRun.GetQuestion(i).QuestionText} : {answer}");
        }
    }
    else
    {
        Console.WriteLine("\tNo responses");
    }
}
```

В этом коде не требуется никаких проверок `null`, так как вы разработали базовые интерфейсы так, чтобы они возвращали ссылочные типы, не допускающие значения null.

Получите код

Вы можете получить код этого руководства из нашего репозитория samples в папке csharp/NullableIntroduction.

Экспериментируйте, изменяя объявления типов между ссылочными типами, допускающими и не допускающими значение null. Посмотрите, как создаются различные предупреждения, чтобы избежать случайного разыменования `null`.

Следующие шаги

Узнайте, как использовать ссылочный тип, допускающий значение NULL, при использовании Entity Framework:

Основы Entity Framework Core. Работа со ссылочными типами, допускающими значение NULL

Учебник. Создание и использование асинхронных потоков с использованием C# 8.0 и .NET Core 3.0

18.12.2021 • 8 minutes to read

В C# 8.0 представлены **асинхронные потоки**, которые моделируют источник данных потоковой передачи. Потоки данных часто извлекают или создают элементы асинхронно. Асинхронные потоки полагаются на новые интерфейсы, появившиеся в .NET Standard 2.1. Эти интерфейсы поддерживаются в .NET Core 3.0 и более поздних версиях. Они предоставляют естественную модель программирования для асинхронных потоковых источников данных.

В этом руководстве вы узнаете, как:

- Создать источник данных, который формирует последовательность элементов данных асинхронно.
- Использовать этот источник данных асинхронно.
- Поддержка отмены и перехваченных контекстов для асинхронных потоков.
- Распознавать, когда новый интерфейс и источник данных предпочтительнее для более ранних синхронных последовательностей данных.

Предварительные требования

Вам нужно настроить свой компьютер для выполнения .NET Core, включая компилятор C# 8.0. Компилятор C# 8 доступен, начиная с версии 16.3 Visual Studio 2019 или в пакете SDK .NET Core 3.0.

Чтобы вы могли получить доступ к конечной точке GraphQL GitHub, необходимо создать маркер доступа GitHub. Выберите следующие разрешения для маркеров доступа GitHub.

- repo:status
- public_repo

Храните маркер доступа в надежном месте, чтобы вы могли использовать его для получения доступа к конечной точке API GitHub.

> **WARNING**
> Храните свой личный маркер доступа в безопасном месте. Любое программное обеспечение с вашим личным маркером доступа может выполнять вызовы API GitHub с помощью ваших прав доступа.

В этом руководстве предполагается, что вы знакомы с C# и .NET, включая Visual Studio или .NET CLI.

Запуск начального приложения

Вы можете получить код для начального приложения, используемый в этом руководстве, в репозитории dotnet/docs в папке csharp/whats-new/tutorials.

Начальное приложение представляет собой консольное приложение, которое использует интерфейс GraphQL GitHub для получения последних проблем, написанных в репозитории dotnet/docs. Начнем с просмотра следующего кода для метода `Main` начального приложения.

```
static async Task Main(string[] args)
{
    //Follow these steps to create a GitHub Access Token
    // https://help.github.com/articles/creating-a-personal-access-token-for-the-command-line/#creating-a-
token
    //Select the following permissions for your GitHub Access Token:
    // - repo:status
    // - public_repo
    // Replace the 3rd parameter to the following code with your GitHub access token.
    var key = GetEnvVariable("GitHubKey",
    "You must store your GitHub key in the 'GitHubKey' environment variable",
    "");

    var client = new GitHubClient(new Octokit.ProductHeaderValue("IssueQueryDemo"))
    {
        Credentials = new Octokit.Credentials(key)
    };

    var progressReporter = new progressStatus((num) =>
    {
        Console.WriteLine($"Received {num} issues in total");
    });
    CancellationTokenSource cancellationSource = new CancellationTokenSource();

    try
    {
        var results = await RunPagedQueryAsync(client, PagedIssueQuery, "docs",
            cancellationSource.Token, progressReporter);
        foreach(var issue in results)
            Console.WriteLine(issue);
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine("Work has been cancelled");
    }
}
```

Вы можете задать переменную среды `GitHubKey` личному маркеру доступа или заменить последний аргумент в вызове на `GetEnvVariable` с помощью личного маркера доступа. Не размещайте код доступа в исходном коде, если будете предоставлять общий доступ к источнику другим пользователям. Никогда не отправляйте коды доступа в репозиторий с общим исходным кодом.

После создания клиента GitHub код в `Main` создает объект отчета о ходе выполнения и маркер отмены. После создания этих объектов `Main` вызывает `RunPagedQueryAsync`, чтобы получить более 250 недавно созданных проблем. Результаты отобразятся после выполнения этой задачи.

При запуске начального приложения вы можете обнаружить некоторые важные замечания о том, как будет выполняться приложение. Вы увидите ход выполнения, передаваемый каждой странице, возвращенной с GitHub. Прежде чем GitHub вернет каждую новую страницу проблем, возникает заметная пауза. Наконец, проблемы отображаются только после того, как получены все 10 страниц с GitHub.

Изучение реализации

Реализация показывает, почему возникло поведение, обсуждавшееся в предыдущем разделе. Изучите код для `RunPagedQueryAsync`.

```
private static async Task<JArray> RunPagedQueryAsync(GitHubClient client, string queryText, string repoName,
CancellationToken cancel, IProgress<int> progress)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    JArray finalResults = new JArray();
    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results = JObject.Parse(response.HttpResponse.Body.ToString());

        int totalCount = (int)issues(results)["totalCount"];
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"];
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)["startCursor"].ToString();
        issuesReturned += issues(results)["nodes"].Count();
        finalResults.Merge(issues(results)["nodes"]);
        progress?.Report(issuesReturned);
        cancel.ThrowIfCancellationRequested();
    }
    return finalResults;

    JObject issues(JObject result) => (JObject)result["data"]["repository"]["issues"];
    JObject pageInfo(JObject result) => (JObject)issues(result)["pageInfo"];
}
```

Давайте сконцентрируемся на алгоритме разбивки по страницам и асинхронной структуре предыдущего кода. (Дополнительные сведения об API GraphQL GitHub см. в этой документации.) Метод `RunPagedQueryAsync` перечисляет проблемы от самых последних до самых старых. Чтобы продолжить с предыдущей страницы, он запрашивает по 25 выпусков на страницу и проверяет структуру ответа `pageInfo`. Это следует за стандартной поддержкой страниц GraphQL для многостраничных ответов. Ответ включает в себя объект `pageInfo`, который содержит значение `hasPreviousPages` и `startCursor`, используемые для запроса предыдущей страницы. Проблемы в массиве `nodes`. Метод `RunPagedQueryAsync` добавляет эти узлы в массив, который содержит результаты со всех страниц.

После получения и восстановления страницы результатов `RunPagedQueryAsync` сообщает о ходе выполнения и проверяет наличие отмены. Если есть запрос на отмену, `RunPagedQueryAsync` выдает OperationCanceledException.

Существует несколько элементов в этом коде, которые можно улучшить. Самое главное, `RunPagedQueryAsync` должен выделить хранилище для всех возвращенных проблем. Этот пример останавливается после нахождения 250 проблем, так как для извлечения всех открытых проблем потребуется гораздо больше памяти на их хранение. Протоколы для поддержки отчетов о ходе выполнения и отмены делают алгоритм более сложным для понимания при первом чтении. Задействовано больше типов и API. Вы должны отслеживать передачу данных с помощью CancellationTokenSource и связанного с ним CancellationToken, чтобы понять, где запрашивается отмена и где она предоставляется.

Предоставление лучшего способа асинхронных потоков

Асинхронные потоки и связанная языковая поддержка обращаются ко всем этим вопросам. Код, который формирует последовательность, теперь может использовать `yield return` для возврата элементов в методе, который был объявлен с помощью модификатора `async`. Вы можете применить асинхронный поток, используя цикл `await foreach`, аналогично любой последовательности с помощью цикла `foreach`.

Эти новые языковые функции зависят от трех новых интерфейсов, добавленных в .NET Standard 2.1 и реализованных в .NET Core 3.0.

- System.Collections.Generic.IAsyncEnumerable<T>
- System.Collections.Generic.IAsyncEnumerator<T>
- System.IAsyncDisposable

Большинство разработчиков C# должны знать об этих трех интерфейсах. Они ведут себя подобно своим синхронным аналогам.

- System.Collections.Generic.IEnumerable<T>
- System.Collections.Generic.IEnumerator<T>
- System.IDisposable

Один тип, который может быть незнаком, — System.Threading.Tasks.ValueTask. Структура `ValueTask` предоставляет API, аналогичный классу System.Threading.Tasks.Task. `ValueTask` используется в этих интерфейсах по причинам производительности.

Преобразование в асинхронные потоки

Затем для создания асинхронного потока преобразуйте метод `RunPagedQueryAsync`. Сначала измените подпись `RunPagedQueryAsync`, чтобы вернуть `IAsyncEnumerable<JToken>`, затем удалите маркер отмены и объекты хода выполнения из списка параметров, как показано в следующем коде.

```
private static async IAsyncEnumerable<JToken> RunPagedQueryAsync(GitHubClient client,
    string queryText, string repoName)
```

В следующем коде показано, как начальный код обрабатывает каждую страницу для извлечения.

```
finalResults.Merge(issues(results)["nodes"]);
progress?.Report(issuesReturned);
cancel.ThrowIfCancellationRequested();
```

Замените эти три строки следующим кодом.

```
foreach (JObject issue in issues(results)["nodes"])
    yield return issue;
```

Вы также можете удалить объявление `finalResults` ранее в этом методе и оператор `return`, следующий за измененным циклом.

Вы завершили изменения для создания асинхронного потока. Готовый метод должен иметь вид, аналогичный приведенному ниже коду:

```
private static async IAsyncEnumerable<JToken> RunPagedQueryAsync(GitHubClient client,
    string queryText, string repoName)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results = JObject.Parse(response.HttpResponse.Body.ToString());

        int totalCount = (int)issues(results)["totalCount"];
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"];
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)["startCursor"].ToString();
        issuesReturned += issues(results)["nodes"].Count();

        foreach (JObject issue in issues(results)["nodes"])
            yield return issue;
    }

    JObject issues(JObject result) => (JObject)result["data"]["repository"]["issues"];
    JObject pageInfo(JObject result) => (JObject)issues(result)["pageInfo"];
}
```

Затем измените код, который использует коллекцию, для асинхронного потока. Найдите следующий код в `Main`, который обрабатывает коллекцию проблем.

```
var progressReporter = new progressStatus((num) =>
{
    Console.WriteLine($"Received {num} issues in total");
});
CancellationTokenSource cancellationSource = new CancellationTokenSource();

try
{
    var results = await RunPagedQueryAsync(client, PagedIssueQuery, "docs",
        cancellationSource.Token, progressReporter);
    foreach(var issue in results)
        Console.WriteLine(issue);
}
catch (OperationCanceledException)
{
    Console.WriteLine("Work has been cancelled");
}
```

Замените код следующим циклом `await foreach`.

```
int num = 0;
await foreach (var issue in RunPagedQueryAsync(client, PagedIssueQuery, "docs"))
{
    Console.WriteLine(issue);
    Console.WriteLine($"Received {++num} issues in total");
}
```

Новый интерфейс IAsyncEnumerator<T> является производным от IAsyncDisposable. Это означает, что предыдущий цикл будет асинхронно удалять поток по завершении цикла. Цикл похож на следующий код:

```
int num = 0;
var enumerator = RunPagedQueryAsync(client, PagedIssueQuery, "docs").GetEnumeratorAsync();
try
{
    while (await enumerator.MoveNextAsync())
    {
        var issue = enumerator.Current;
        Console.WriteLine(issue);
        Console.WriteLine($"Received {++num} issues in total");
    }
} finally
{
    if (enumerator != null)
        await enumerator.DisposeAsync();
}
```

Элементы потока по умолчанию обрабатываются в захваченном контексте. Чтобы отключить захват контекста, используйте метод расширения TaskAsyncEnumerableExtensions.ConfigureAwait. Дополнительные сведения о контекстах синхронизации и захвате текущего контекста см. в статье, посвященной использованию асинхронной модели на основе задач.

Асинхронные потоки поддерживают отмену, используя тот же протокол, что и другие методы `async`. Для поддержки отмены можно изменить сигнатуру для метода асинхронного итератора следующим образом:

```
private static async IAsyncEnumerable<JToken> RunPagedQueryAsync(GitHubClient client,
    string queryText, string repoName, [EnumeratorCancellation] CancellationToken cancellationToken =
default)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results = JObject.Parse(response.HttpResponse.Body.ToString());

        int totalCount = (int)issues(results)["totalCount"];
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"];
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)["startCursor"].ToString();
        issuesReturned += issues(results)["nodes"].Count();

        foreach (JObject issue in issues(results)["nodes"])
            yield return issue;
    }

    JObject issues(JObject result) => (JObject)result["data"]["repository"]["issues"];
    JObject pageInfo(JObject result) => (JObject)issues(result)["pageInfo"];
}
```

Атрибут EnumeratorCancellationAttribute заставляет компилятор создать код для IAsyncEnumerator<T>, который делает токен, передаваемый `GetAsyncEnumerator`, видимым в тексте асинхронного итератора в виде аргумента. Внутри `runQueryAsync` можно проверить состояние маркера и отменить дальнейшую работу при необходимости.

Используйте другой метод расширения, WithCancellation, чтобы передать токен отмены асинхронному потоку. Измените цикл, перечисляя проблемы следующим образом:

```
private static async Task EnumerateWithCancellation(GitHubClient client)
{
    int num = 0;
    var cancellation = new CancellationTokenSource();
    await foreach (var issue in RunPagedQueryAsync(client, PagedIssueQuery, "docs")
        .WithCancellation(cancellation.Token))
    {
        Console.WriteLine(issue);
        Console.WriteLine($"Received {++num} issues in total");
    }
}
```

Вы можете получить код для готового руководства из репозитория dotnet/docs в папке csharp/whats-new/tutorials.

Запуск готового приложения

Снова запустите приложение. Сравните его поведение с поведением начального приложения. Первая

страница результатов перечисляется, как только она становится доступной. Поскольку каждую новую страницу запрашивают и извлекают, результаты следующей страницы быстро перечисляются, возникает пауза. Блок `try` / `catch` не требует обработки отмены. Вызывающий может прекратить перечисление коллекции. Отчет о ходе выполнения четко сформирован, так как асинхронный поток формирует результаты скачивания каждой страницы. Состояние каждой возвращенной проблемы включается в цикл `await foreach`. Для отслеживания хода выполнения объект обратного вызова не требуется.

Изучив код, вы увидите улучшения в использовании памяти. Вам больше не нужно выделять коллекцию для хранения всех результатов до их перечисления. Вызывающий может определить, как использовать результаты и нужен ли набор хранилищ.

Запустите начальное и готовое приложение, и вы увидите различия между реализациями самостоятельно. Вы можете удалить маркер доступа GitHub, созданный при начале работы с этим руководством, после завершения изучения. Если злоумышленник получил доступ к этому маркеру, ему удастся получить доступ к API GitHub с помощью ваших учетных данных.

Учебник. Написание пользовательского обработчика интерполяции строк

18.12.2021 • 10 minutes to read

Из этого руководства вы узнаете, как выполнять следующие задачи:

- Реализация шаблона обработчика интерполяции строк.
- Взаимодействие с получателем в операции интерполяции строк.
- Добавление аргументов в обработчик интерполяции строк.
- Общие сведения о новых возможностях библиотеки для интерполяции строк.

Предварительные требования

Необходимо настроить компьютер для выполнения .NET 6, включая компилятор C# 10. Компилятор C# 10 доступен начиная с Visual Studio 2022 или пакета SDK для .NET 6.

В этом руководстве предполагается, что вы знакомы с C# и .NET, включая Visual Studio или .NET CLI.

Новая структура

В C# 10 добавлена поддержка пользовательского *обработчика интерполяции строк*. Обработчик интерполяции строк — это тип, который обрабатывает выражение заполнителя в интерполированной строке. Без пользовательского обработчика заполнители обрабатываются аналогично String.Format. Каждый заполнитель форматируется как текст, а затем компоненты объединяются для формирования результирующей строки.

Можно написать обработчик для любого сценария, в котором используются сведения о результирующей строке. Будет ли он использоваться? Какие ограничения в формате? Некоторые примеры:

- Может потребоваться, чтобы ни одна из результирующих строк не превышала определенные ограничения, например 80 символов. Вы можете обрабатывать интерполированные строки для заполнения буфера фиксированной длины и прекращать обработку после достижения этой длины буфера.
- Возможно, у вас табличный формат и каждый заполнитель должен иметь фиксированную длину. Пользовательский обработчик может обеспечить это вместо того, чтобы приводить в соответствие весь клиентский код.

В этом учебнике описано, как создать обработчик интерполяции строк для одного из основных сценариев производительности: библиотек ведения журналов. В зависимости от настроенного уровня ведения журнала работа по созданию сообщения журнала не требуется. Если ведение журнала отключено, работа по созданию строки из выражения интерполированной строки не требуется. Сообщение не выводится, поэтому любое объединение строк можно пропустить. Кроме того, не нужно выполнять любые выражения, используемые в заполнителях, включая создание трассировок стека.

Обработчик интерполяции строк может определить, будет ли использоваться форматированная строка, и выполнять только необходимую работу.

Начальная реализация

Начнем с базового класса `Logger`, который поддерживает различные уровни.

```
public enum LogLevel
{
    Off,
    Critical,
    Error,
    Warning,
    Information,
    Trace
}

public class Logger
{
    public LogLevel EnabledLevel { get; init; } = LogLevel.Error;

    public void LogMessage(LogLevel level, string msg)
    {
        if (EnabledLevel < level) return;
        Console.WriteLine(msg);
    }
}
```

Этот `Logger` поддерживает шесть разных уровней. Если сообщение не проходит фильтр уровня журнала, выходные данные отсутствуют. Общедоступный API для средства ведения журнала принимает в качестве сообщения строку (полностью отформатированную). Вся работа по созданию строки уже выполнена.

Реализация шаблона обработчика

Этот шаг нужен для создания *обработчика интерполяции строк*, который воссоздает текущее поведение. Обработчик интерполяции строк — это тип, который должен иметь следующие характеристики:

- System.Runtime.CompilerServices.InterpolatedStringHandlerAttribute, примененный к типу.
- Конструктор с двумя параметрами `int`: `literalLength` и `formatCount`. (Разрешается иметь больше параметров.)
- Открытый метод `AppendLiteral` с сигнатурой: `public void AppendLiteral(string s)`.
- Универсальный открытый метод `AppendFormatted` с сигнатурой: `public void AppendFormatted<T>(T t)`.

На внутреннем уровне построитель создает форматированную строку и предоставляет элемент клиенту для получения этой строки. В следующем коде показан тип `LogInterpolatedStringHandler`, который соответствует этим требованиям:

```
[InterpolatedStringHandler]
public ref struct LogInterpolatedStringHandler
{
    // Storage for the built-up string
    StringBuilder builder;

    // Add the receiver argument:
    public LogInterpolatedStringHandler(int literalLength, int formattedCount)
    {
        builder = new StringBuilder(literalLength);
        Console.WriteLine($"\tliteral length: {literalLength}, formattedCount: {formattedCount}");
    }

    public void AppendLiteral(string s)
    {
        Console.WriteLine($"\tAppendLiteral called: {{{s}}}");

        builder.Append(s);
        Console.WriteLine($"\tAppended the literal string");
    }

    public void AppendFormatted<T>(T t)
    {
        Console.WriteLine($"\tAppendFormatted called: {{{t}}} is of type {typeof(T)}");

        builder.Append(t?.ToString());
        Console.WriteLine($"\tAppended the formatted object");
    }

    internal string GetFormattedText() => builder.ToString();
}
```

Теперь можно добавить перегрузку в `LogMessage` в классе `Logger`, чтобы опробовать новый обработчик интерполяции строк.

```
public void LogMessage(LogLevel level, LogInterpolatedStringHandler builder)
{
    if (EnabledLevel < level) return;
    Console.WriteLine(builder.GetFormattedText());
}
```

Нет необходимости удалять исходный метод `LogMessage`. Если аргумент представляет собой выражение интерполированной строки, то компилятор предпочитает метод с параметром обработчика интерполяции методу с параметром `string`.

Чтобы убедиться, что новый обработчик вызывается, используйте следующий код в качестве основной программы:

```
var logger = new Logger() { EnabledLevel = LogLevel.Warning };
var time = DateTime.Now;

logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time}. This is an error. It will be
printed.");
logger.LogMessage(LogLevel.Trace, $"Trace Level. CurrentTime: {time}. This won't be printed.");
logger.LogMessage(LogLevel.Warning, "Warning Level. This warning is a string, not an interpolated string
expression.");
```

В результате запуска приложения выводятся данные, похожие на следующий текст:

```
        literal length: 65, formattedCount: 1
        AppendLiteral called: {Error Level. CurrentTime: }
        Appended the literal string
        AppendFormatted called: {10/20/2021 12:19:10 PM} is of type System.DateTime
        Appended the formatted object
        AppendLiteral called: {. This is an error. It will be printed.}
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. This is an error. It will be printed.
        literal length: 50, formattedCount: 1
        AppendLiteral called: {Trace Level. CurrentTime: }
        Appended the literal string
        AppendFormatted called: {10/20/2021 12:19:10 PM} is of type System.DateTime
        Appended the formatted object
        AppendLiteral called: {. This won't be printed.}
        Appended the literal string
Warning Level. This warning is a string, not an interpolated string expression.
```

Отслеживая выходные данные, можно увидеть, как компилятор добавляет код для вызова обработчика и построения строки:

- Компилятор добавляет вызов для создания обработчика, передавая общую длину литерального текста в строке форматирования и количество заполнителей.
- Компилятор добавляет вызовы к `AppendLiteral` и `AppendFormatted` для каждого раздела литеральной строки и заполнителя.
- Компилятор вызывает метод `LogMessage`, используя `CoreInterpolatedStringHandler` в качестве аргумента.

Наконец, обратите внимание, что последнее предупреждение не вызывает обработчик интерполяции строк. Аргумент имеет значение `string`, поэтому вызов активирует другую перегрузку со строковым параметром.

Добавление дополнительных возможностей в обработчик

В предыдущей версии обработчика интерполяции строк реализован шаблон. Во избежание обработки каждого выражения заполнителя вам потребуются дополнительные сведения в обработчике. В этом разделе описано, как улучшить обработчик, чтобы он выполнял меньше работы, если построенная строка не будет записываться в журнал. Для указания сопоставления параметров с общедоступным API и параметрами конструктора обработчика используется System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute. Это предоставляет обработчику сведения, необходимые для определения того, должна ли вычисляться интерполированная строка.

Начнем с внесения изменений в обработчик. Сначала, если обработчик включен, добавьте поле для отслеживания. Добавьте в конструктор два параметра: один для указания уровня ведения журнала для этого сообщения, а вторым должна быть ссылка на объект журнала:

```
private readonly bool enabled;

public LogInterpolatedStringHandler(int literalLength, int formattedCount, Logger logger, LogLevel logLevel)
{
    enabled = logger.EnabledLevel >= logLevel;
    builder = new StringBuilder(literalLength);
    Console.WriteLine($"\tliteral length: {literalLength}, formattedCount: {formattedCount}");
}
```

Затем используйте поле, чтобы обработчик добавлял только литералы или отформатированные объекты, когда будет использоваться последняя строка:

```
public void AppendLiteral(string s)
{
    Console.WriteLine($"\tAppendLiteral called: {{{s}}}");
    if (!enabled) return;

    builder.Append(s);
    Console.WriteLine($"\tAppended the literal string");
}

public void AppendFormatted<T>(T t)
{
    Console.WriteLine($"\tAppendFormatted called: {{{t}}} is of type {typeof(T)}");
    if (!enabled) return;

    builder.Append(t?.ToString());
    Console.WriteLine($"\tAppended the formatted object");
}
```

Далее необходимо обновить объявление `LogMessage`, чтобы компилятор передавал дополнительные параметры конструктору обработчика. Это обрабатывается с помощью System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute в аргументе обработчика:

```
public void LogMessage(LogLevel level, [InterpolatedStringHandlerArgument("", "level")]
LogInterpolatedStringHandler builder)
{
    if (EnabledLevel < level) return;
    Console.WriteLine(builder.GetFormattedText());
}
```

Этот атрибут задает список аргументов для `LogMessage`, которые сопоставляются с параметрами, следующими за обязательными параметрами `literalLength` и `formattedCount`. Пустая строка ("") указывает получателя. Компилятор заменяет значение объекта `Logger`, представленного `this`, следующим аргументом конструктора обработчика. Компилятор заменяет значение `level` для следующего аргумента. Можно указать любое количество аргументов для любого записываемого обработчика. Добавляемые аргументы являются строковыми аргументами.

Эту версию можно запустить с помощью того же тестового кода. На этот раз отобразятся следующие результаты:

```
        literal length: 65, formattedCount: 1
        AppendLiteral called: {Error Level. CurrentTime: }
        Appended the literal string
        AppendFormatted called: {10/20/2021 12:19:10 PM} is of type System.DateTime
        Appended the formatted object
        AppendLiteral called: {. This is an error. It will be printed.}
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. This is an error. It will be printed.
        literal length: 50, formattedCount: 1
        AppendLiteral called: {Trace Level. CurrentTime: }
        AppendFormatted called: {10/20/2021 12:19:10 PM} is of type System.DateTime
        AppendLiteral called: {. This won't be printed.}
Warning Level. This warning is a string, not an interpolated string expression.
```

Можно увидеть, что методы `AppendLiteral` и `AppendFormat` вызываются, но не выполняют никаких действий. Обработчик определил, что последняя строка не понадобится, поэтому не создает ее. Осталось внести еще несколько улучшений.

Во-первых, можно добавить перегрузку `AppendFormatted`, ограничивающую аргумент типом, реализующим System.IFormattable. Эта перегрузка позволяет вызывающим объектам добавлять строки

формата в заполнители. При внесении этого изменения мы также изменим тип возвращаемого значения для других методов `AppendFormatted` и `AppendLiteral` с `void` на `bool` (если какой-либо из этих методов имеет другие типы возвращаемого значения, вы получите ошибку компиляции). Это изменение обеспечивает *сокращенное вычисление*. Методы возвращают значение `false`, чтобы указать, что обработку выражения интерполированной строки необходимо остановить. Если возвращено значение `true`, это указывает на то, что она должна продолжаться. В этом примере оно используется для отмены обработки, когда результирующая строка не требуется. Сокращенное вычисление поддерживает более детализированные действия. Для поддержки буферов фиксированной длины обработку выражения можно прерывать после достижения определенной длины. Или определенное условие может означать, что остальные элементы не нужны.

```
public bool AppendFormatted<T>(T t, string format) where T : IFormattable
{
    Console.WriteLine($"\tAppendFormatted (IFormattable version) called: {t} with format {{{format}}} is of
type {typeof(T)},");
    if (!enabled) return false;

    builder.Append(t?.ToString(format, null));
    Console.WriteLine($"\tAppended the formatted object");
    return true;
}
```

С этим дополнением вы можете указать строки форматирования в своем выражении интерполированной строки.

```
var time = DateTime.Now;

logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time}. The time doesn't use formatting.");
logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time:t}. This is an error. It will be
printed.");
logger.LogMessage(LogLevel.Trace, $"Trace Level. CurrentTime: {time:t}. This won't be printed.");
```

`:t` в первом сообщении указывает краткий формат для текущего времени. В предыдущем примере показана одна из перегрузок к методу `AppendFormatted`, который можно создать для обработчика. Не нужно указывать универсальный аргумент для форматируемого объекта. Вы можете воспользоваться более эффективными способами преобразования создаваемых типов в строку. Вы можете записать перегрузки метода `AppendFormatted`, который принимает эти типы вместо универсального аргумента. Компилятор выберет наиболее подходящую перегрузку. Среда выполнения использует этот метод для преобразования System.Span<T> в выходные данные строки. Можно добавить целочисленный параметр для указания *выравнивания* выходных данных с или без IFormattable. Параметр System.Runtime.CompilerServices.DefaultInterpolatedStringHandler, который входит в состав .NET 6, содержит девять перегрузок AppendFormatted для различных применений. Его можно использовать в качестве ссылки при создании обработчика для своих целей.

Запустите пример сейчас, и вы увидите, что для сообщения `Trace` будет вызван только первый `AppendLiteral`:

```
        literal length: 60, formattedCount: 1
        AppendLiteral called: Error Level. CurrentTime:
        Appended the literal string
        AppendFormatted called: 10/20/2021 12:18:29 PM is of type System.DateTime
        Appended the formatted object
        AppendLiteral called: . The time doesn't use formatting.
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:18:29 PM. The time doesn't use formatting.
        literal length: 65, formattedCount: 1
        AppendLiteral called: Error Level. CurrentTime:
        Appended the literal string
        AppendFormatted (IFormattable version) called: 10/20/2021 12:18:29 PM with format {t} is of type
System.DateTime,
        Appended the formatted object
        AppendLiteral called: . This is an error. It will be printed.
        Appended the literal string
Error Level. CurrentTime: 12:18 PM. This is an error. It will be printed.
        literal length: 50, formattedCount: 1
        AppendLiteral called: Trace Level. CurrentTime:
Warning Level. This warning is a string, not an interpolated string expression.
```

Можно выполнить еще последнее обновление конструктора обработчика, которое повышает эффективность. Обработчик может добавить последний параметр `out bool`. Значение `false`, заданное для этого параметра, указывает, что обработчик вообще не должен вызываться для обработки выражения интерполированной строки:

```
public LogInterpolatedStringHandler(int literalLength, int formattedCount, LogLevel level, Logger logger,
out bool isEnabled)
{
    enabled = logger.EnabledLevel >= level;
    Console.WriteLine($"\tliteral length: {literalLength}, formattedCount: {formattedCount}");
    if (enabled)
    {
        builder = new StringBuilder(literalLength);
    }
    isEnabled = enabled;
}
```

При запуске примера выходные данные должны выглядеть следующим образом:

```
        literal length: 60, formattedCount: 1
        AppendLiteral called: Error Level. CurrentTime:
        Appended the literal string
        AppendFormatted called: 10/20/2021 12:19:10 PM is of type System.DateTime
        Appended the formatted object
        AppendLiteral called: . The time doesn't use formatting.
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. The time doesn't use formatting.
        literal length: 65, formattedCount: 1
        AppendLiteral called: Error Level. CurrentTime:
        Appended the literal string
        AppendFormatted (IFormattable version) called: 10/20/2021 12:19:10 PM with format {t} is of type
System.DateTime,
        Appended the formatted object
        AppendLiteral called: . This is an error. It will be printed.
        Appended the literal string
Error Level. CurrentTime: 12:19 PM. This is an error. It will be printed.
        literal length: 50, formattedCount: 1
Warning Level. This warning is a string, not an interpolated string expression.
```

Если указан параметр `LogLevel.Trace`, должны отображаться только выходные данные конструктора.

Обработчик указал, что он не включен, поэтому ни один из методов `Append` не был вызван.

В этом примере показан важный момент для обработчиков интерполяции строк, особенно при использовании библиотек ведения журнала. Никаких побочных эффектов в заполнителях возникнуть не должно. Добавьте следующий код в основную программу и понаблюдайте за этим поведением в действии.

```
int index = 0;
int numberOfIncrements = 0;
for (var level = LogLevel.Critical; level <= LogLevel.Trace; level++)
{
    Console.WriteLine(level);
    logger.LogMessage(level, $"{level}: Increment index a few times {index++}, {index++}, {index++},
{index++}, {index++}");
    numberOfIncrements += 5;
}
Console.WriteLine($"Value of index {index}, value of numberOfIncrements: {numberOfIncrements}");
```

Можно увидеть, что переменная `index` увеличивается в пять раз каждую итерацию цикла. Так как заполнители оцениваются только для уровней `Critical`, `Error` и `Warning`, а не для `Information` и `Trace`, конечное значение `index` не соответствует ожиданиям.

```
Critical
Critical: Increment index a few times 0, 1, 2, 3, 4
Error
Error: Increment index a few times 5, 6, 7, 8, 9
Warning
Warning: Increment index a few times 10, 11, 12, 13, 14
Information
Trace
Value of index 15, value of numberOfIncrements: 25
```

Обработчики интерполяции строк обеспечивают больший контроль над преобразованием выражения интерполированной строки в строку. Команда разработчиков среды выполнения .NET уже использовала эту возможность для повышения производительности в нескольких областях. Вы можете использовать те же возможности в собственных библиотеках. Чтобы узнать больше, ознакомьтесь с System.Runtime.CompilerServices.DefaultInterpolatedStringHandler. Он предоставляет более полную реализацию, чем была создана здесь. Там также есть еще много возможных перегрузок для методов `Append`.

Создание форматированных строк с помощью интерполяции

18.12.2021 • 6 minutes to read

В этом руководстве объясняется, как с помощью интерполяции строк в C# вставить значения в одну результирующую строку. Вы напишете код C# и сможете просмотреть результаты его компиляции и выполнения. Это руководство содержит ряд уроков по вставке значений в строки и форматированию этих значений различными способами.

Для работы с этим руководством вам потребуется компьютер, который можно использовать для разработки. В руководстве .NET по созданию программы Hello World за 10 минут содержатся инструкции по настройке локальной среды разработки в macOS, Windows или Linux. Вы можете также воспользоваться интерактивной версией этого руководства в браузере.

Создание интерполированной строки

Создайте каталог с именем *interpolated*. Сделайте его текущим, выполнив следующую команду в окне консоли:

```
dotnet new console
```

Эта команда создает консольное приложение .NET Core в текущем каталоге.

Откройте файл *Program.cs* в любом редакторе и замените строку `Console.WriteLine("Hello World!");` следующим кодом, указав вместо `<name>` свое имя:

```
var name = "<name>";
Console.WriteLine($"Hello, {name}. It's a pleasure to meet you!");
```

Чтобы выполнить этот код, введите `dotnet run` в окне консоли. При запуске программы отображается одна строка, которая содержит ваше имя в приветствии. Строка, включенная в вызов метода WriteLine, является *выражением интерполированной строки*. Это похоже на шаблон, позволяющий создать одну строку (называемую *результирующей строкой*) из строки, содержащей внедренный код. Интерполированные строки особенно удобны при вставке значений в строку или сцеплении (объединении) строк.

Этот простой пример содержит два элемента, обязательные для каждой интерполированной строки:

- Строковый литерал, который начинается с символа `$`, стоящего до открывающей кавычки. Между символом `$` и знаком кавычки не должно быть пробелов. (Если вы хотите узнать, что при этом случится, вставьте пробел после символа `$`, сохраните файл и снова запустите программу, введя `dotnet run` в окне консоли. Компилятор C# выводит сообщение об ошибке "Ошибка CS1056: Недопустимый символ '$'".)
- Одно или несколько *интерполированных выражений*. Интерполированное выражение обозначено открывающей и закрывающей фигурной скобкой (`{` и `}`). Вы можете указать внутри фигурных скобок любое выражение C#, возвращающее значение (включая `null`).

Давайте рассмотрим еще несколько примеров интерполяции строк с другими типами данных.

Включение разных типов данных

В предыдущем разделе вы использовали интерполяцию строк для вставки одной строки внутрь другой. При этом интерполированное выражение может относиться к любому типу данных. Давайте включим в интерполированную строку значения разных типов данных.

В приведенном ниже примере сначала мы определим тип данных для класса `Vegetable`, обладающего свойством `Name` и методом `ToString`. Этот метод переопределяет поведение метода Object.ToString(). Модификатор доступа `public` делает этот метод доступным любому клиентскому коду и позволяет получить строковое представление экземпляра `Vegetable`. В примере метод `Vegetable.ToString` возвращает значение свойства `Name`, которое инициализируется в конструкторе `Vegetable`.

```
public Vegetable(string name) => Name = name;
```

Затем создайте экземпляр класса `Vegetable` с именем `item`. Для этого воспользуйтесь оператором `new` и укажите имя для конструктора `Vegetable`.

```
var item = new Vegetable("eggplant");
```

Наконец, переменная `item` включается в интерполированную строку, которая также содержит значение DateTime, значение Decimal и значение перечисления `Unit`. Замените весь код C# в редакторе следующим кодом, а затем используйте команду `dotnet run`, чтобы запустить его:

```
using System;

public class Vegetable
{
   public Vegetable(string name) => Name = name;

   public string Name { get; }

   public override string ToString() => Name;
}

public class Program
{
   public enum Unit { item, kilogram, gram, dozen };

   public static void Main()
   {
      var item = new Vegetable("eggplant");
      var date = DateTime.Now;
      var price = 1.99m;
      var unit = Unit.item;
      Console.WriteLine($"On {date}, the price of {item} was {price} per {unit}.");
   }
}
```

Обратите внимание на то, что интерполированное выражение `item` в интерполированной строке разрешается в текст eggplant в результирующей строке. Связано это с тем, что если результат выражения не имеет строковый тип, он разрешается в строку описанным ниже образом.

- Если результатом вычисления интерполированного выражения является `null`, используется пустая строка ("" или String.Empty).
- Если результатом вычисления интерполированного выражения не является `null`, обычно вызывается метод `ToString` результирующего типа. Чтобы проверить это, можно изменить

реализацию метода `Vegetable.ToString`. Возможно, вам вообще не потребуется реализовывать метод `ToString`, так как его реализация в той или иной форме присутствует в каждом типе. Это можно проверить, закомментировав определение метода `Vegetable.ToString` в примере, для чего перед ним нужно поставить символ комментария (`//`). В выходных данных строка "eggplant" заменяется полным именем типа ("Vegetable" в этом примере), что является стандартным поведением метода Object.ToString(). По умолчанию метод `ToString` возвращает для значения перечисления строковое представление значения.

В выходных данных этого примера дата является слишком точной (цена на баклажаны не меняется каждую секунду), а в значении цены не указана единица валюты. В следующем разделе вы узнаете, как устранить эти проблемы, управляя форматом строковых представлений результатов выражений.

Управление форматированием интерполированных выражений

В предыдущем разделе мы вставили две неправильно отформатированные строки в результирующую строку. Первая была значением даты и времени, при этом допустимой была только дата. Вторая была ценой, в которой отсутствовала единица валюты. Обе эти проблемы легко решить. Интерполяция строк позволяет указывать *строки формата*, управляющие форматированием определенных типов. Измените вызов `Console.WriteLine` из предыдущего примера, включив в него строки формата для выражений даты и цены, как показано в следующей строке:

```
Console.WriteLine($"On {date:d}, the price of {item} was {price:C2} per {unit}.");
```

Задайте строку формата, указав ее после интерполированного выражения через двоеточие. "d" — это стандартная строка формата для даты и времени, представляющая краткий формат. "C2" — это стандартная строка числового формата, представляющая число в виде денежной единицы с точностью два знака после запятой.

Некоторые типы в библиотеках .NET поддерживают предопределенный набор строк формата. К ним относятся все числовые типы, а также типы даты и времени. Полный список типов, поддерживающих строки формата, см. в разделе Строки формата и типы библиотек классов .NET статьи Типы форматирования в .NET.

Попробуйте изменить строки формата в текстовом редакторе, возвращаясь в программу после каждого изменения, чтобы узнать, как они влияют на форматирование даты и времени, а также числового значения. Измените "d" в `{date:d}` на "t" (чтобы отобразить краткий формат времени), "y" (чтобы отобразить год и месяц) и "yyyy" (чтобы отобразить год в виде четырехзначного числа). Измените "C2" в `{price:C2}` на "e" (для экспоненциального представления) и "F3" (чтобы получить числовое значение с тремя знаками после запятой).

Кроме форматирования, вы можете управлять шириной поля и выравниванием для форматированных строк, включаемых в результирующую строку. В следующем разделе вы научитесь это делать.

Управление шириной поля и выравниванием для интерполированных выражений

Как правило, когда результат интерполированного выражения форматируется как строка, она включается в результирующую строку без начальных или конечных пробелов. Особенно когда вы работаете с набором данных, возможность управления шириной поля и выравниванием помогает получить более понятные выходные данные. Чтобы увидеть это, замените весь код в текстовом редакторе следующим кодом, а затем введите `dotnet run` для выполнения программы:

```
using System;
using System.Collections.Generic;

public class Example
{
   public static void Main()
   {
      var titles = new Dictionary<string, string>()
      {
          ["Doyle, Arthur Conan"] = "Hound of the Baskervilles, The",
          ["London, Jack"] = "Call of the Wild, The",
          ["Shakespeare, William"] = "Tempest, The"
      };

      Console.WriteLine("Author and Title List");
      Console.WriteLine();
      Console.WriteLine($"|{"Author",-25}|{"Title",30}|");
      foreach (var title in titles)
         Console.WriteLine($"|{title.Key,-25}|{title.Value,30}|");
   }
}
```

Имена авторов выровнены по левому краю, а названия их произведений — по правому. Вы можете указать выравнивание, добавив запятую (,) после интерполированного выражения и назначив *минимальную* ширину поля. Если указанное значение является положительным числом, то поле выравнивается по правому краю. Если оно является отрицательным числом, то поле выравнивается по левому краю.

Попробуйте удалить знаки "минус" из кода `{"Author",-25}` и `{title.Key,-25}`, а затем снова выполните пример, как показано в следующем коде:

```
Console.WriteLine($"|{"Author",25}|{"Title",30}|");
foreach (var title in titles)
   Console.WriteLine($"|{title.Key,25}|{title.Value,30}|");
```

На этот раз сведения об авторе выровнены по правому краю.

Вы можете совмещать описатель выравнивания и строку формата в одном интерполированном выражении. Для этого сначала укажите выравнивание, а затем через двоеточие строку формата. Замените весь код внутри метода `Main` приведенным ниже кодом, который отображает три отформатированные строки с заданной шириной поля. Запустите программу, введя команду `dotnet run`.

```
Console.WriteLine($"[{DateTime.Now,-20:d}] Hour [{DateTime.Now,-10:HH}] [{1063.342,15:N2}] feet");
```

Выходные данные выглядят следующим образом:

```
[04/14/2018          ] Hour [16        ] [       1,063.34] feet
```

Вы завершили работу с руководством по интерполяции строк.

Дополнительные сведения об интерполяции строк см. в разделе Интерполяция строк и в руководстве Интерполяция строк в C#.

Интерполяция строк на C#

18.12.2021 • 5 minutes to read

В этом руководстве описано, как использовать интерполяцию строк для форматирования и включения результатов выражения в строку результатов. В примерах предполагается, что вам знакомы основные принципы C# и форматирования типов .NET. Если вы не знакомы с интерполяцией строк или форматированием типов .NET, сначала ознакомьтесь с интерактивным учебником по интерполяции строк. Дополнительные сведения о форматировании типов в .NET см. в разделе Типы форматирования в .NET.

> **NOTE**
>
> Примеры C# в этой статье выполняются во встроенном средстве выполнения кода и на площадке Try.NET. Нажмите на кнопку **Выполнить**, чтобы выполнить пример в интерактивном окне. После выполнения кода вы можете изменить его и выполнить измененный код, снова нажав на кнопку **Выполнить**. Либо в интерактивном окне выполняется измененный код, либо, если компиляция завершается с ошибкой, в интерактивном окне отображаются все сообщения об ошибках компилятора C#.

Введение

Функция интерполяции строк создана на основе функции составного форматирования и имеет более удобный синтаксис для включения форматированных результатов выражения в строку результатов.

Для определения строкового литерала в качестве интерполированной строки добавьте к началу символ `$`. Вы можете внедрить любое допустимое выражение C#, возвращающее значение в интерполированной строке. В следующем примере после вычисления выражения его результат преобразуется в строку и включается в строку результатов:

```
double a = 3;
double b = 4;
Console.WriteLine($"Area of the right triangle with legs of {a} and {b} is {0.5 * a * b}");
Console.WriteLine($"Length of the hypotenuse of the right triangle with legs of {a} and {b} is
{CalculateHypotenuse(a, b)}");

double CalculateHypotenuse(double leg1, double leg2) => Math.Sqrt(leg1 * leg1 + leg2 * leg2);

// Expected output:
// Area of the right triangle with legs of 3 and 4 is 6
// Length of the hypotenuse of the right triangle with legs of 3 and 4 is 5
```

Как показано в примере, можно включить выражение в интерполированную строку, заключив его в фигурные скобки:

```
{<interpolationExpression>}
```

Интерполированные строки поддерживают все возможности составного форматирования строк. Это делает их более удобочитаемыми по сравнению с использованием метода String.Format.

Как указать строку формата для выражения интерполяции

Задайте строку формата, которая поддерживается типом результата выражения, указав ее после выражения интерполяции через двоеточие:

```
{<interpolationExpression>:<formatString>}
```

В следующем примере показано, как задать стандартные и настраиваемые строки формата для выражений, возвращающих дату и время или числовые результаты:

```
var date = new DateTime(1731, 11, 25);
Console.WriteLine($"On {date:dddd, MMMM dd, yyyy} Leonhard Euler introduced the letter e to denote
{Math.E:F5} in a letter to Christian Goldbach.");

// Expected output:
// On Sunday, November 25, 1731 Leonhard Euler introduced the letter e to denote 2.71828 in a letter to
Christian Goldbach.
```

Дополнительные сведения см. в разделе Компонент строки формата в статье Составное форматирование. Этот раздел содержит ссылки на разделы, описывающие стандартные и настраиваемые строки формата, поддерживаемые базовыми типами .NET.

Управление шириной поля и выравниванием в форматированных выражениях интерполяции

Задайте минимальную ширину поля и выравнивание форматированного результата выражения, указав константное выражение после выражения интерполяции через запятую:

```
{<interpolationExpression>,<alignment>}
```

Если значение *alignment* положительное, форматированное выражение будет выровнено по правому краю, а если отрицательное — по левому.

Если вам нужно задать и выравнивание, и строку формата, начните с компонента выравнивания:

```
{<interpolationExpression>,<alignment>:<formatString>}
```

В следующем примере показано, как задать выравнивание. Текстовые поля разграничены символом вертикальной черты ("|"):

```
const int NameAlignment = -9;
const int ValueAlignment = 7;

double a = 3;
double b = 4;
Console.WriteLine($"Three classical Pythagorean means of {a} and {b}:");
Console.WriteLine($"|{"Arithmetic",NameAlignment}|{0.5 * (a + b),ValueAlignment:F3}|");
Console.WriteLine($"|{"Geometric",NameAlignment}|{Math.Sqrt(a * b),ValueAlignment:F3}|");
Console.WriteLine($"|{"Harmonic",NameAlignment}|{2 / (1 / a + 1 / b),ValueAlignment:F3}|");

// Expected output:
// Three classical Pythagorean means of 3 and 4:
// |Arithmetic|  3.500|
// |Geometric|  3.464|
// |Harmonic |  3.429|
```

В выходных данных в примере видно, что если длина форматированного результата выражения превышает заданную ширину поля, значение *alignment* игнорируется.

Дополнительные сведения см. в разделе Компонент выравнивания в статье Составное форматирование.

Как использовать escape-последовательности в интерполированной строке

Интерполированные строки поддерживают все escape-последовательности, которые могут использоваться в обычных строковых литералах. Дополнительные сведения см. в статье Escape-последовательности строки.

Для литеральной интерпретации escape-последовательности используйте строковый литерал verbatim. Интерполированная строка verbatim начинается с символа `$`, за которым следует символ `@`. Начиная с C# 8.0 маркеры `$` и `@` можно использовать в любом порядке: `$@"..."` и `@$"..."` являются допустимыми интерполированными строками verbatim.

В строке результатов указывайте двойную фигурную скобку "{{" или "}}". Дополнительные сведения см. в разделе escape-скобки в статье Составное форматирование.

В следующем примере показано, как включить фигурные скобки в строку результата и создать интерполированную строку verbatim:

```
var xs = new int[] { 1, 2, 7, 9 };
var ys = new int[] { 7, 9, 12 };
Console.WriteLine($"Find the intersection of the {{{string.Join(", ",xs)}}} and {{{string.Join(", ",ys)}}}
sets.");

var userName = "Jane";
var stringWithEscapes = $"C:\\Users\\{userName}\\Documents";
var verbatimInterpolated = $@"C:\Users\{userName}\Documents";
Console.WriteLine(stringWithEscapes);
Console.WriteLine(verbatimInterpolated);

// Expected output:
// Find the intersection of the {1, 2, 7, 9} and {7, 9, 12} sets.
// C:\Users\Jane\Documents
// C:\Users\Jane\Documents
```

Как использовать троичный условный оператор `?:` в выражении интерполяции

Двоеточие (:) имеет особое значение в элементе выражения интерполяции. Чтобы использовать условный оператор в выражении, заключите это выражение в скобки, как показано в следующем примере:

```
var rand = new Random();
for (int i = 0; i < 7; i++)
{
    Console.WriteLine($"Coin flip: {(rand.NextDouble() < 0.5 ? "heads" : "tails")}");
}
```

Создание строки результата с интерполяцией для определенного языка и региональных параметров

По умолчанию в интерполированной строке используется текущий язык и региональные параметры, определяемые свойством CultureInfo.CurrentCulture для всех операций форматирования. Используйте неявное преобразование интерполированной строки для экземпляра System.FormattableString и вызовите метод ToString(IFormatProvider), чтобы создать строку результата с определенным языком и региональными параметрами. Следующий пример показывает, как это сделать:

```
var cultures = new System.Globalization.CultureInfo[]
{
    System.Globalization.CultureInfo.GetCultureInfo("en-US"),
    System.Globalization.CultureInfo.GetCultureInfo("en-GB"),
    System.Globalization.CultureInfo.GetCultureInfo("nl-NL"),
    System.Globalization.CultureInfo.InvariantCulture
};

var date = DateTime.Now;
var number = 31_415_926.536;
FormattableString message = $"{date,20}{number,20:N3}";
foreach (var culture in cultures)
{
    var cultureSpecificMessage = message.ToString(culture);
    Console.WriteLine($"{culture.Name,-10}{cultureSpecificMessage}");
}

// Expected output is like:
// en-US       5/17/18 3:44:55 PM      31,415,926.536
// en-GB      17/05/2018 15:44:55      31,415,926.536
// nl-NL        17-05-18 15:44:55      31.415.926,536
//            05/17/2018 15:44:55      31,415,926.536
```

Как показано в примере, можно использовать один экземпляр FormattableString для создания нескольких строк результата для различных языков и региональных параметров.

Как создать строку результата с помощью инвариантного языка и региональных параметров

Наряду с методом FormattableString.ToString(IFormatProvider) можно использовать статический метод FormattableString.Invariant, чтобы разрешить интерполированную строку в строке результата для InvariantCulture. Следующий пример показывает, как это сделать:

```
string messageInInvariantCulture = FormattableString.Invariant($"Date and time in invariant culture:
{DateTime.Now}");
Console.WriteLine(messageInInvariantCulture);

// Expected output is like:
// Date and time in invariant culture: 05/17/2018 15:46:24
```

Заключение

В этом руководстве описаны распространенные сценарии использования интерполяции строк. Дополнительные сведения об интерполяции строк см. в разделе Интерполяция строк. Дополнительные сведения о форматировании типов в .NET см. в разделах Типы форматирования в .NET и Составное форматирование.

См. также

- String.Format
- System.FormattableString
- System.IFormattable
- Строки

Консольное приложение

18.12.2021 • 9 minutes to read

Это руководство раскроет для вас некоторые возможности .NET и языка C#. Вы узнаете:

- общие сведения о .NET CLI;
- структура консольного приложения C#;
- консольный ввод-вывод;
- основные сведения об интерфейсах API файлового ввода-вывода в .NET;
- основные сведениях об асинхронном программировании задач в .NET.

Вам предстоит создать приложение, которое считывает текстовый файл и выводит его содержимое в консоль. Вывод в консоль осуществляется с такой скоростью, которая позволяет читать текст вслух. Скорость можно увеличивать или уменьшать клавишами "<" (меньше) и ">" (больше). Это приложение можно запустить в ОС Windows, Linux, macOS или в контейнере Docker.

В этом руководстве описано множество функций. Попробуем собрать их по одному.

Предварительные требования

- Пакет SDK для .NET 6.
- Редактор кода.

Создание приложения

Первым шагом является создание нового приложения. Откройте командную строку и создайте новый каталог для приложения. Перейдите в этот каталог. В командной строке введите команду `dotnet new console`. Эта команда создает начальный набор файлов для базового приложения Hello World.

Прежде чем вносить изменения, давайте запустим простое приложение Hello World. Когда вы создадите приложение, наберите в командной строке команду `dotnet run`. Эта команда запускает процесс восстановления пакета NuGet, создает исполняемый файл приложения и запускает этот файл.

Весь код простого приложения Hello World размещается в файле *Program.cs*. Откройте этот файл в любом текстовом редакторе. Замените код в *Program.cs* на следующий код:

```
namespace TeleprompterConsole;

internal class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```

В верхней части файла вы видите инструкцию `namespace`. Как и другие объектно ориентированные языки, с которыми вы могли работать ранее, C# использует пространства имен для организации типов. В нашей программе Hello World все точно так же. Как вы видите, программа находится в пространстве имен `TeleprompterConsole`.

Чтение и вывод файла

Первая функция, которую мы добавим, будет считывать данные из текстового файла и выводить полученный текст в консоль. Сначала нам нужно добавить текстовый файл. Скопируйте в каталог проекта файл sampleQuotes.txt из репозитория GitHub для этого примера. Он будет источником текста для вашего приложения. Чтобы скачать пример приложения для этого раздела, воспользуйтесь инструкциями в разделе Примеры и руководства.

Теперь добавьте в класс `Program` (он расположен сразу за методом `Main`) следующий метод:

```
static IEnumerable<string> ReadFrom(string file)
{
    string? line;
    using (var reader = File.OpenText(file))
    {
        while ((line = reader.ReadLine()) != null)
        {
            yield return line;
        }
    }
}
```

Этот метод является специальным типом *метода перечислителя* C#. Метод перечислителя возвращает последовательности, для которых применяется отложенное вычисление. Это означает, что каждый элемент в последовательности создается только в тот момент, когда к нему выполняется обращение в коде обработки последовательности. Методы перечислителя содержат один или несколько операторов `yield return`. Возвращаемый методом `ReadFrom` объект содержит код для создания каждого элемента последовательности. В нашем примере он читает следующую строку текста из исходного файла и возвращает эту строку. Каждый раз, когда вызывающий код запрашивает следующий элемент из последовательности, код считывает из файла и возвращает следующую строку текста. Когда файл закончится, последовательность сообщает, что в ней больше нет элементов.

Здесь используются два элемента синтаксиса C#, которые могут быть для вас новыми. Оператор `using` в этом методе управляет освобождением ресурсов. Переменная, которая инициализируется в инструкции `using` (в нашем примере это `reader`) должна реализовывать интерфейс IDisposable. Этот интерфейс определяет единственный метод (`Dispose`), который вызывается для освобождения ресурса. Компилятор создает такой вызов, когда выполнение кода достигает закрывающей скобки инструкции `using`. Созданный компилятором код гарантирует освобождение ресурса даже в том случае, если в блоке кода, определенном инструкцией using, будет создано исключение.

Переменная `reader` определена с ключевым словом `var`. Ключевое слово `var` определяет *неявно типизированную локальную переменную*. Это означает, что тип переменной определяется во время компиляции по типу объекта, присвоенного этой переменной. Здесь это возвращаемое значение метода OpenText(String), то есть объект StreamReader.

Теперь давайте создадим в методе `Main` код для чтения файла:

```
var lines = ReadFrom("sampleQuotes.txt");
foreach (var line in lines)
{
    Console.WriteLine(line);
}
```

Запустите программу командой `dotnet run` и убедитесь в том, что все текстовые строки выводятся в консоль.

Добавление задержек и форматирование выходных данных

Сейчас данные отображаются слишком быстро для чтения. Поэтому нам нужно добавить задержку в процесс вывода. Для этого вы создадите несложный код, выполняющий асинхронную обработку. Но первые наши действия будут нарушать стандартные рекомендации. Эти нарушения мы укажем в комментариях при создании кода, а затем заменим этот код в последующих шагах.

В этом разделе описаны два действия. Во-первых, обновите метод итератора, чтобы он возвращал не всю строку целиком, а каждое слово отдельно. Для этого внесите такие изменения. Замените инструкцию `yield return line;` следующим кодом:

```
var words = line.Split(' ');
foreach (var word in words)
{
    yield return word + " ";
}
yield return Environment.NewLine;
```

Теперь следует изменить код обработки строк файла, добавив задержку после вывода каждого слова. Замените инструкцию `Console.WriteLine(line)` в методе `Main` на такой блок кода:

```
Console.Write(line);
if (!string.IsNullOrWhiteSpace(line))
{
    var pause = Task.Delay(200);
    // Synchronously waiting on a task is an
    // anti-pattern. This will get fixed in later
    // steps.
    pause.Wait();
}
```

Запустите пример и проверьте выходные данные. Теперь слова появляются по одному и с задержками по 200 мс. Но пока с выводом сохраняются некоторые проблемы, поскольку в исходном текстовом файле есть несколько строк длиной более 80 символов, и они выводятся без перевода строки. Это не очень удобно читать с прокруткой. Но эту проблему легко исправить. Вам нужно лишь отслеживать длину каждой строки и создавать новую строку каждый раз, когда эта длина достигает определенного порога. После объявления `words` в методе `ReadFrom` объявите локальную переменную для хранения длины строки:

```
var lineLength = 0;
```

Теперь добавьте следующий код после инструкции `yield return word + " ";` (перед закрывающей фигурной скобкой):

```
lineLength += word.Length + 1;
if (lineLength > 70)
{
    yield return Environment.NewLine;
    lineLength = 0;
}
```

Запустите пример, и теперь вы сможете читать текст вслух в заданном темпе.

Асинхронные задачи

И на последнем этапе мы добавим код, который позволяет выполнять две асинхронные задачи, одна из которых — вывод текста, а вторая — ожидание ввода от пользователя для ускорения, замедления или прекращения вывода текста. Этот этап разделяется на несколько шагов, по завершении которых вы получите все необходимые обновления. Первым шагом является создание асинхронной задачи (Task), которая возвращает метод с тем кодом, который вы создали ранее для чтения и отображения файла.

Добавьте следующий метод в класс `Program`. Этот текст основан на тексте метода `Main`:

```
private static async Task ShowTeleprompter()
{
    var words = ReadFrom("sampleQuotes.txt");
    foreach (var word in words)
    {
        Console.Write(word);
        if (!string.IsNullOrWhiteSpace(word))
        {
            await Task.Delay(200);
        }
    }
}
```

Вы можете заметить два изменения. Во-первых, в тексте нет вызова Wait(), который в синхронном режиме ожидает завершения задачи. Вместо него в этой версии используется ключевое слово `await`. Чтобы это работало, в сигнатуру метода нужно добавить модификатор `async`. Этот метод возвращает `Task`. Обратите внимание, что здесь нет инструкции для возвращения объекта `Task`. Вместо этого объект `Task` создается в коде, который компилятор предоставляет в точке использования оператора `await`. Представьте, что метод завершает выполнение при достижении `await`. Он возвращает `Task` в знак того, что работа еще не завершена. Метод возобновит свою работу, когда завершится ожидаемая задача. Когда работа метода завершится, это будет отражено в возвращаемом объекте `Task`. Вызывающий код может отслеживать состояние полученного `Task`, чтобы определить момент завершения метода.

Теперь наш новый метод можно вызвать из метода `Main`:

```
ShowTeleprompter().Wait();
```

Здесь, в методе `Main`, код синхронно ожидает завершения. Всегда, когда это возможно, следует использовать оператор `await` вместо синхронного ожидания. Но в методе `Main` консольного приложения запрещено использовать оператор `await`. В противном случае приложение завершит работу раньше, чем выполнит все свои задачи.

> **NOTE**
> При использовании C# 7.1 или более поздней версии консольные приложения можно создавать с помощью метода `async Main`.

Теперь следует создать второй асинхронный метод, который позволяет считывать данные ввода из консоли и реагировать на клавиши "<" (меньше), ">" (больше) и "X" или "x". Для выполнения этой задачи добавьте приведенный ниже метод.

```
private static async Task GetInput()
{
    var delay = 200;
    Action work = () =>
    {
        do {
            var key = Console.ReadKey(true);
            if (key.KeyChar == '>')
            {
                delay -= 10;
            }
            else if (key.KeyChar == '<')
            {
                delay += 10;
            }
            else if (key.KeyChar == 'X' || key.KeyChar == 'x')
            {
                break;
            }
        } while (true);
    };
    await Task.Run(work);
}
```

Здесь создается лямбда-выражение, представляющее делегат Action, который считывает нажатие клавиши из консоли и изменяет локальную переменную с длительностью задержки, если пользователь нажал клавишу "<" (меньше) или ">" (больше). Выполнение метода делегата можно завершить, нажав клавишу "X" или "x". Таким образом пользователь может в любой момент прекратить отображение текста. Этот метод использует метод ReadKey(), чтобы блокировать выполнение и ожидать нажатия клавиши.

Чтобы завершить создание этой функции, нам нужна новая инструкция `async Task`, которая вернет метод, запускающий обе задачи (`GetInput` и `ShowTeleprompter`) и управляющий обменом данными между этими задачами.

Пришло время создать класс, который может обрабатывать совместное использование данных двумя задачами. Этот класс содержит два открытых свойства: delay (задержка) и флаг `Done`, который означает, что файл прочитан полностью:

```
namespace TeleprompterConsole;

internal class TelePrompterConfig
{
    public int DelayInMilliseconds { get; private set; } = 200;
    public void UpdateDelay(int increment) // negative to speed up
    {
        var newDelay = Min(DelayInMilliseconds + increment, 1000);
        newDelay = Max(newDelay, 20);
        DelayInMilliseconds = newDelay;
    }
    public bool Done { get; private set; }
    public void SetDone()
    {
        Done = true;
    }
}
```

Поместите этот класс в отдельный новый файл и включите его в пространство имен `TeleprompterConsole`, как показано выше. Также следует добавить оператор `using static` в верхнюю часть файлами, чтобы можно было ссылаться на методы `Min` и `Max` без указания имени внешнего класса или пространства

имен. Оператор `using static` импортирует методы из одного класса. Это отличается от оператора `using` без `static`, который импортирует все классы из пространства имен.

```
using static System.Math;
```

Теперь вам нужно обновить методы `ShowTeleprompter` и `GetInput` для использования нового объекта `config`. И еще одна инструкция `Task`, которая возвращает метод `async`, запускающий обе задачи и завершающий работу после окончания первой задачи:

```
private static async Task RunTeleprompter()
{
    var config = new TelePrompterConfig();
    var displayTask = ShowTeleprompter(config);

    var speedTask = GetInput(config);
    await Task.WhenAny(displayTask, speedTask);
}
```

Новым методом здесь является WhenAny(Task[]). Этот метод создает задачу (`Task`), которая завершается сразу, как только завершится любая из задач в списке аргументов.

Теперь вам нужно обновить методы `ShowTeleprompter` и `GetInput`, чтобы они использовали объект `config` для задержки:

```
private static async Task ShowTeleprompter(TelePrompterConfig config)
{
    var words = ReadFrom("sampleQuotes.txt");
    foreach (var word in words)
    {
        Console.Write(word);
        if (!string.IsNullOrWhiteSpace(word))
        {
            await Task.Delay(config.DelayInMilliseconds);
        }
    }
    config.SetDone();
}

private static async Task GetInput(TelePrompterConfig config)
{
    Action work = () =>
    {
        do {
            var key = Console.ReadKey(true);
            if (key.KeyChar == '>')
                config.UpdateDelay(-10);
            else if (key.KeyChar == '<')
                config.UpdateDelay(10);
            else if (key.KeyChar == 'X' || key.KeyChar == 'x')
                config.SetDone();
        } while (!config.Done);
    };
    await Task.Run(work);
}
```

Новая версия метода `ShowTeleprompter` вызывает новый метод из класса `TeleprompterConfig`. Сейчас нужно изменить метод `Main`, чтобы вместо `ShowTeleprompter` он вызывал `RunTeleprompter`:

```
RunTeleprompter().Wait();
```

Заключение

В этом учебнике мы продемонстрировали вам ряд функций языка C# и библиотек .NET Core, связанных с работой в консольных приложениях. На основе полученных знаний вы сможете развивать свои представления о языке и представленных здесь классах. Вы увидели базовые примеры использования файлового и консольного ввода-вывода, асинхронного программирования на основе задач с блокировкой и без блокировки. Вы узнали о языке C# и структуре программ на C#, а также о .NET CLI.

Дополнительные сведения о файловом вводе-выводе см. в статье Файловый и потоковый ввод-вывод. Дополнительные сведения о модели асинхронного программирования, используемой в учебнике, см. в статьях Асинхронное программирование на основе задач и Асинхронное программирование.

Учебник по созданию HTTP-запросов в консольном приложении .NET на C#

18.12.2021 • 7 minutes to read

В этом учебнике рассказывается, как создать приложение, которое отправляет HTTP-запросы к службе REST на GitHub. Приложение считывает информацию в формате JSON и преобразовывает ее в объекты C#. Преобразование из данных JSON в объекты C# называется *десериализацией*.

В этом учебнике вы научитесь следующему:

- отправлять HTTP-запросы;
- десериализовать JSON-ответы;
- настраивать десериализацию с использованием атрибутов.

Если вы хотите поработать с примером окончательного кода по этому учебнику, вы можете загрузить такой пример. Инструкции по загрузке см. в разделе Просмотр и скачивание примеров.

Предварительные требования

- Пакет Azure SDK 5.0 или более поздней версии
- В примерах ниже используется кроссплатформенный редактор Visual Studio Code с открытым исходным кодом. Пример приложения можно запускать в ОС Windows, Linux, macOS или в контейнере Docker.

Создание клиентского приложения

1. Откройте командную строку и создайте каталог для приложения. Перейдите в этот каталог.
2. Введите следующую команду в окне консоли:

   ```
   dotnet new console --name WebAPIClient
   ```

   Эта команда создает начальный набор файлов для базового приложения Hello World. Имя проекта — "WebAPIClient".

3. Перейдите в каталог WebAPIClient и запустите приложение.

   ```
   cd WebAPIClient
   ```

   ```
   dotnet run
   ```

   `dotnet run` автоматически запускает `dotnet restore`, чтобы восстановить все зависимости, необходимые приложению. Эта команда также запускает `dotnet build` при необходимости.

Создание HTTP-запросов

Это приложение вызывает API GitHub для получения сведений о проектах под зонтичным брендом .NET Foundation. Конечная точка имеет значение https://api.github.com/orgs/dotnet/repos. Для получения

сведений создается HTTP-запрос get. Браузеры также используют HTTP-запросы get, поэтому вы можете указать этот URL-адрес в адресной строке браузера и увидеть, какие сведения вы будете получать и обрабатывать.

Используйте класс HttpClient, чтобы выполнять HTTP-запросы. HttpClient поддерживает только асинхронные методы для длительно выполняющихся API-интерфейсов. Следующие шаги позволяют создать асинхронный метод и вызывают его из метода Main.

1. Откройте файл `Program.cs`, расположенный в каталоге проекта, и добавьте в класс `Program` следующий асинхронный метод:

   ```
   private static async Task ProcessRepositories()
   {
   }
   ```

2. Добавьте директиву `using` в начало файла *Program.cs*, чтобы компилятор C# распознал тип Task:

   ```
   using System.Threading.Tasks;
   ```

   Если запустить `dotnet build` на этом этапе, компиляция выполнится, но отобразится предупреждение о том, что этот метод не содержит операторы `await` и поэтому будет выполнен синхронно. Вы добавите операторы `await` позже по мере заполнения метода.

3. Замените метод `Main` следующим кодом:

   ```
   static async Task Main(string[] args)
   {
       await ProcessRepositories();
   }
   ```

   Этот код выполняет следующие действия:

   - изменяет сигнатуру `Main`, добавляя модификатор `async` и изменяя тип возвращаемого значения на `Task`;
   - заменяет оператор `Console.WriteLine` вызовом `ProcessRepositories`, в котором используется ключевое слово `await`.

4. В классе `Program` создайте статический экземпляр HttpClient для обработки запросов и ответов.

   ```
   namespace WebAPIClient
   {
       class Program
       {
           private static readonly HttpClient client = new HttpClient();

           static async Task Main(string[] args)
           {
               //...
           }
       }
   }
   ```

5. В методе `ProcessRepositories` вызовите конечную точку GitHub, которая возвращает список всех репозиториев .NET Foundation:

```
private static async Task ProcessRepositories()
{
    client.DefaultRequestHeaders.Accept.Clear();
    client.DefaultRequestHeaders.Accept.Add(
        new MediaTypeWithQualityHeaderValue("application/vnd.github.v3+json"));
    client.DefaultRequestHeaders.Add("User-Agent", ".NET Foundation Repository Reporter");

    var stringTask = client.GetStringAsync("https://api.github.com/orgs/dotnet/repos");

    var msg = await stringTask;
    Console.Write(msg);
}
```

Этот код выполняет следующие действия:

- задает заголовки HTTP для всех запросов, а именно
  - `Accept` для приема ответов в формате JSON;
  - `User-Agent`. Код сервера GitHub проверяет эти заголовки, они необходимы для извлечения сведений из GitHub;
- вызывает HttpClient.GetStringAsync(String), чтобы выполнить веб-запрос и получить ответ. Этот метод запускает задачу, которая выполняет веб-запрос. При возврате запроса задача считывает поток ответа и извлекает содержимое из этого потока. В качестве текста ответа возвращаются данные типа String. Они доступны по завершении задачи;
- ожидает задачу для строки ответа и выводит ответ на консоль.

6. Добавьте две директивы `using` в начало файла:

```
using System.Net.Http;
using System.Net.Http.Headers;
```

7. Выполните сборку приложения и запустите его.

```
dotnet run
```

Предупреждение о сборке отсутствует, так как `ProcessRepositories` теперь содержит оператор `await`.

Выходные данные являются длинным текстом JSON.

Десериализация результата JSON

Шаги ниже позволяют преобразовать ответ в формате JSON в объекты C#. Используйте класс System.Text.Json.JsonSerializer, чтобы десериализовать данные JSON в объекты.

1. Создайте файл с именем *repo.cs* и добавьте следующий код:

```
using System;

namespace WebAPIClient
{
    public class Repository
    {
        public string name { get; set; }
    }
}
```

Код выше определяет класс, представляющий объект JSON, возвращаемый из API GitHub. Этот класс будет использоваться для вывода списка имен репозиториев.

Данные JSON для объекта репозитория содержат десятки свойств, но только свойство `name` будет десериализовано. Сериализатор автоматически игнорирует свойства JSON, для которых нет совпадения в целевом классе. Эта функция упрощает создание типов, работающих только с подмножеством полей из пакета JSON.

Соглашение C# должно сделать прописными первые буквы имен свойств, но свойство `name` здесь начинается со строчной буквы, так как точно соответствует свойству в JSON. Далее вы узнаете, как использовать имена свойств на C#, которые не соответствуют именам свойств JSON.

2. Используйте сериализатор для преобразования пакета JSON в объекты C#. В методе `ProcessRepositories` замените вызов GetStringAsync(String) следующими строками:

```
var streamTask = client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
var repositories = await JsonSerializer.DeserializeAsync<List<Repository>>(await streamTask);
```

В обновленном коде GetStringAsync(String) заменится на GetStreamAsync(String). Метод сериализатора использует в качестве источника поток, а не строку.

Первый аргумент JsonSerializer.DeserializeAsync<TValue>(Stream, JsonSerializerOptions, CancellationToken) является выражением `await`. Выражения `await` могут использоваться почти в любом месте кода, хотя пока мы их применяли только в операторе назначения. Два других параметра, `JsonSerializerOptions` и `CancellationToken`, необязательны и не включены во фрагмент кода.

Метод `DeserializeAsync` является *общим*, что означает, что необходимо предоставить аргументы типа согласно типу объектов, создаваемых на основе текста JSON. В этом примере выполняется десериализация в объект `List<Repository>`, который является еще одним общим объектом, System.Collections.Generic.List<T>. Класс `List<T>` хранит коллекцию объектов. Аргумент типа определяет тип объектов, хранящихся в `List<T>`. Аргумент типа является классом `Repository`, так как текст JSON представляет собой коллекцию объектов репозитория.

3. Добавьте код для вывода имени каждого репозитория. Удалите вот эти строки кода:

```
var msg = await stringTask;
Console.Write(msg);
```

на новый код:

```
foreach (var repo in repositories)
    Console.WriteLine(repo.name);
```

4. Добавьте следующие директивы `using` в начало файла:

```
using System.Collections.Generic;
using System.Text.Json;
```

5. Запустите приложение.

```
dotnet run
```

Оно выведет список имен всех репозиториев, которые принадлежат к .NET Foundation.

Настройка десериализации

1. В файле *repo.cs* измените свойство `name` на `Name` и добавьте атрибут `[JsonPropertyName]`, чтобы указать, как это свойство отображается в данных JSON.

```
[JsonPropertyName("name")]
public string Name { get; set; }
```

2. Добавьте пространство имен System.Text.Json.Serialization к директивам `using`:

```
using System.Text.Json.Serialization;
```

3. В файле *Program.cs* обновите код, чтобы использовать свойство `Name` с прописной буквы:

```
Console.WriteLine(repo.Name);
```

4. Запустите приложение.

   Результат тот же самый.

Рефакторинг кода

Метод `ProcessRepositories` может выполнять работу в асинхронном режиме и возвращает коллекцию репозиториев. Измените метод так, чтобы он возвращал `List<Repository>`, а фрагмент кода, который записывает информацию, переместите в метод `Main`.

1. Измените сигнатуру `ProcessRepositories`, чтобы этот метод возвращал задачу, результатом которой является список объектов `Repository`:

```
private static async Task<List<Repository>> ProcessRepositories()
```

2. Верните репозитории после обработки JSON-ответа:

```
var streamTask = client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
var repositories = await JsonSerializer.DeserializeAsync<List<Repository>>(await streamTask);
return repositories;
```

   Компилятор создает объект `Task<T>` в качестве выходных данных, так как этот метод обозначен ключевым словом `async`.

3. Измените метод `Main`, чтобы он собирал результаты и записывал имя каждого репозитория в консоль. Метод `Main` теперь выглядит следующим образом:

```
public static async Task Main(string[] args)
{
    var repositories = await ProcessRepositories();

    foreach (var repo in repositories)
        Console.WriteLine(repo.Name);
}
```

4. Запустите приложение.

   Результат тот же самый.

Десериализация дополнительных свойств

Следующие шаги позволяют добавить код для обработки дополнительных свойств в полученном пакете JSON. Вы, вероятно, не будете обрабатывать каждое свойство, но добавление еще нескольких свойств позволит продемонстрировать другие возможности C#.

1. Добавьте следующие свойства в определение класса `Repository`:

   ```
   [JsonPropertyName("description")]
   public string Description { get; set; }

   [JsonPropertyName("html_url")]
   public Uri GitHubHomeUrl { get; set; }

   [JsonPropertyName("homepage")]
   public Uri Homepage { get; set; }

   [JsonPropertyName("watchers")]
   public int Watchers { get; set; }
   ```

   Типы Uri и `int` имеют встроенные функции для преобразования в строковое представление и из него. Для десериализации из строкового формата JSON в эти целевые типы не требуется дополнительного кода. Если пакет JSON содержит данные, которые не преобразуются в целевой тип, действие сериализации создает исключение.

2. Обновите метод `Main`, чтобы отобразить значения свойств:

   ```
   foreach (var repo in repositories)
   {
       Console.WriteLine(repo.Name);
       Console.WriteLine(repo.Description);
       Console.WriteLine(repo.GitHubHomeUrl);
       Console.WriteLine(repo.Homepage);
       Console.WriteLine(repo.Watchers);
       Console.WriteLine();
   }
   ```

3. Запустите приложение.

   Теперь список содержит дополнительные свойства.

Добавление свойства даты

Дата последней операции push-уведомления в ответе JSON имеет следующий формат:

```
2016-02-08T21:27:00Z
```

Этот формат предназначен для времени в формате UTC, поэтому результатом десериализации является значение DateTime, свойство Kind которого равно Utc.

Чтобы получить дату и время в вашем часовом поясе, вам необходимо написать пользовательский метод преобразования.

1. В файле *repo.cs* добавьте свойство `public` для представления даты и времени в формате UTC и

свойство `LastPush` `readonly`, которое возвращает дату, преобразованную в местное время:

```
[JsonPropertyName("pushed_at")]
public DateTime LastPushUtc { get; set; }

public DateTime LastPush => LastPushUtc.ToLocalTime();
```

Свойство `LastPush` определяется с помощью *члена, заданного выражением* для метода доступа `get`. Метод доступа `set` отсутствует. Путем пропуска метода доступа `set` в C# определяется свойство, доступное *только для чтения*. (Да, вы можете создать в C# даже свойства *только для записи*, но для них трудно найти применение.)

2. Добавьте еще одну инструкцию вывода данных в файл *Program.cs*:

```
Console.WriteLine(repo.LastPush);
```

3. Запустите приложение.

   Выходные данные включают дату и время последней отправки в каждый репозиторий.

Дальнейшие действия

В этом руководстве вы создали приложение, которое выполняет веб-запросы и анализирует результаты. Теперь версия вашего приложения должна совпадать с полной версией примера.

Дополнительные сведения о настройке сериализации JSON см. в статье Сериализация и десериализация (маршалирование и демаршалирование) JSON в .NET.

Работа с запросами LINQ

18.12.2021 • 14 minutes to read

Введение

В этом руководстве описаны возможности .NET Core и C#. Вы узнаете, как выполнять такие задачи.

- создавать последовательности с помощью LINQ;
- писать методы, которые можно применять в запросах LINQ;
- различать упреждающее и отложенное вычисление.

Вы освоите эти методы на примере приложения, которое демонстрирует один из основных навыков любого иллюзиониста: тасовка по методу фаро. Так называют метод тасовки, при котором колода делится ровно на две части, а затем собирается заново так, что карты из каждой половины следуют строго поочередно.

Этот метод очень удобен для иллюзионистов, поскольку положение каждой карты после каждой тасовки точно известно, и через несколько циклов порядок карт восстанавливается.

Здесь же он используется в качестве не слишком серьезного примера для процессов управления последовательностями данных. Приложение, которое вы создадите, будет моделировать колоду карт и выполнять для них серию тасовок, выводя новый порядок карт после каждой из них. Вы сможете сравнить новый порядок карт с исходным.

Это руководство описывает несколько шагов. После каждого из них вы сможете запустить приложение и оценить результаты. Готовый пример доступен в репозитории dotnet/samples на сайте GitHub. Инструкции по загрузке см. в разделе Просмотр и скачивание примеров.

Предварительные требования

Компьютер должен быть настроен для выполнения .NET Core. Инструкции по установке см. на странице Загрузка.NET Core. Это приложение можно запустить в ОС Windows, Ubuntu Linux или OS X, а также в контейнере Docker. Вам потребуется редактор кода, но вы можете выбрать любой привычный для вас. В примерах ниже используется кроссплатформенный редактор Visual Studio Code с открытым исходным кодом. Вы можете заменить его на любое другое средство, с которым вам удобно работать.

Создание приложения

Первым шагом является создание нового приложения. Откройте командную строку и создайте новый каталог для приложения. Перейдите в этот каталог. В командной строке введите команду `dotnet new console`. Эта команда создает начальный набор файлов для базового приложения Hello World.

Если вы раньше никогда не работали с C#, изучите структуру программы C# по этому руководству. Мы рекомендуем сначала ознакомиться с ним, а затем вернуться сюда и продолжить изучение LINQ.

Создание набора данных

Перед началом работы убедитесь, что в верхней части файла `Program.cs`, созданного `dotnet new console`, находятся следующие строки:

```
// Program.cs
using System;
using System.Collections.Generic;
using System.Linq;
```

Если эти три строки (инструкции `using`) находятся не в верхней части файла, наша программа не будет компилироваться.

Теперь, когда у вас есть все необходимые ссылки, посмотрите, из чего состоит колода карт. Как правило, в колоде игральных карт четыре масти и в каждой масти по тринадцать значений. Вы можете создать класс `Card` сразу же и заполнить коллекцию объектами `Card` вручную. С помощью LINQ колоду карт можно создать гораздо быстрее, чем обычным способом. Вместо класса `Card` вы можете создать две последовательности, представляющие масти и ранги соответственно. Вы создадите два очень простых *метода итератора*, которые будут создавать ранги и масти как IEnumerable<T> строк:

```
// Program.cs
// The Main() method

static IEnumerable<string> Suits()
{
    yield return "clubs";
    yield return "diamonds";
    yield return "hearts";
    yield return "spades";
}

static IEnumerable<string> Ranks()
{
    yield return "two";
    yield return "three";
    yield return "four";
    yield return "five";
    yield return "six";
    yield return "seven";
    yield return "eight";
    yield return "nine";
    yield return "ten";
    yield return "jack";
    yield return "queen";
    yield return "king";
    yield return "ace";
}
```

Разместите их под методом `Main` в вашем файле `Program.cs`. Оба эти два метода используют синтаксис `yield return`, создавая последовательность по мере выполнения. Компилятор создаст объект, который реализует интерфейс IEnumerable<T>, и сохранит в него последовательность строк по мере их получения.

Теперь с помощью этих методов итератора создайте колоду карт. Поместите запрос LINQ в метод `Main`. Это описано ниже:

```
// Program.cs
static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                       from r in Ranks()
                       select new { Suit = s, Rank = r };

    // Display each card that we've generated and placed in startingDeck in the console
    foreach (var card in startingDeck)
    {
        Console.WriteLine(card);
    }
}
```

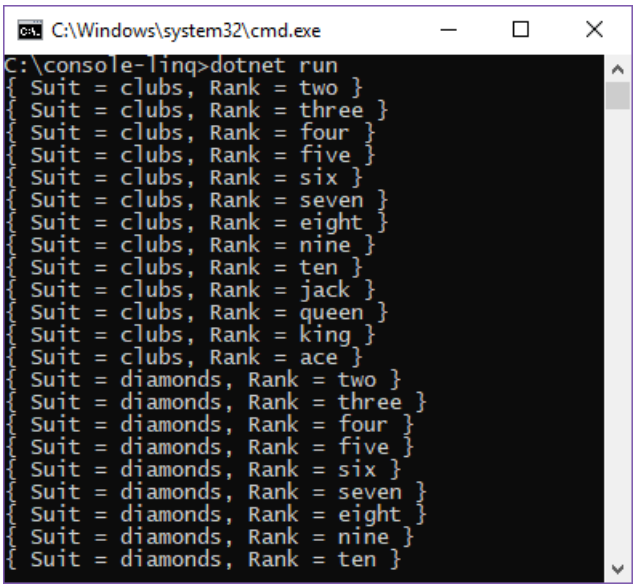
Несколько выражений `from` создают запрос SelectMany, который формирует одну последовательность из сочетаний каждого элемента первой последовательности с каждым элементом второй последовательности. Для нашего примера важен порядок последовательности. Первый элемент первой последовательности (масти) поочередно сочетается с каждым элементом второй последовательности (ранги). В итоге мы получаем все тринадцать карт первой масти. Этот процесс повторяется для каждого элемента первой последовательности (масти). Конечным результатом является колода карт, упорядоченная сначала по мастям, а затем по достоинствам.

Важно помнить, что независимо от того, будете ли вы записывать LINQ в синтаксисе запросов, показанном выше, или вместо этого будете использовать синтаксис методов, вы всегда можете перейти от одной формы синтаксиса к другой. Приведенный выше запрос, записанный в синтаксисе запросов, можно записать в синтаксис метода следующим образом:

```
var startingDeck = Suits().SelectMany(suit => Ranks().Select(rank => new { Suit = suit, Rank = rank }));
```

Компилятор преобразует инструкции LINQ, написанные с помощью синтаксиса запросов, в эквивалентный синтаксис вызова метода. Таким образом, независимо от выбранного синтаксиса, две версии запроса дают одинаковый результат. Выберите, какой синтаксис лучше всего подходит для вашей ситуации. Например, если вы работаете в команде, в которой у некоторых участников есть сложности с синтаксисом метода, попробуйте использовать синтаксис запроса.

Теперь давайте выполним пример, который вы создали к этому моменту. Он отобразит все 52 карты колоды. Возможно, вам будет интересно выполнить этот пример в отладчике и проследить за выполнением методов `Suits()` и `Ranks()`. Вы сможете заметить, что каждая строка в каждой последовательности создается только по мере необходимости.



Обработка порядка

Теперь рассмотрим, как вы будете тасовать карты в колоде. Чтобы хорошо потасовать, сначала необходимо разделить колоду на две части. Эту возможность вам предоставят методы Take и Skip, входящие в интерфейсы API LINQ. Поместите их под циклом `foreach`:

```
// Program.cs
public static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                       from r in Ranks()
                       select new { Suit = s, Rank = r };

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    // 52 cards in a deck, so 52 / 2 = 26
    var top = startingDeck.Take(26);
    var bottom = startingDeck.Skip(26);
}
```

В стандартной библиотеке нет метода для тасовки, которым можно было бы воспользоваться, поэтому вам нужно написать собственный. Метод для тасовки, который вы создадите, иллюстрирует несколько приемов, которые вы будете использовать в программах на основе LINQ, поэтому каждая часть этого процесса будет описана в действиях.

Чтобы добавить некоторые функции для взаимодействия с IEnumerable<T>, который будет возвращен в запросах LINQ, вам потребуется написать особые методы, называемые методами расширения. Короче говоря, метод расширения представляет собой специализированный *статический метод*, добавляющий новые функциональные возможности в уже имеющийся тип без изменения исходного типа, в который необходимо добавить функциональные возможности.

Переместите методы расширения в другое расположение, добавив новый файл *статического* класса в программу с именем `Extensions.cs`, а затем приступите к разработке первого метода расширения:

```
// Extensions.cs
using System;
using System.Collections.Generic;
using System.Linq;

namespace LinqFaroShuffle
{
    public static class Extensions
    {
        public static IEnumerable<T> InterleaveSequenceWith<T>(this IEnumerable<T> first, IEnumerable<T>
second)
        {
            // Your implementation will go here soon enough
        }
    }
}
```

Обратите внимание на сигнатуру метода, в частности на параметры:

```
public static IEnumerable<T> InterleaveSequenceWith<T> (this IEnumerable<T> first, IEnumerable<T> second)
```

Как вы видите, для первого аргумента этого метода добавлен модификатор `this`. Это означает, что

метод вызывается так, как если бы он был методом-членом и имел тип, указанный для первого аргумента. Такое объявление методов соответствует стандартному принципу, по которому для входа и выхода используется тип `IEnumerable<T>`. Такая практика позволяет объединять методы LINQ в цепочку, чтобы создавать более сложные запросы.

Очевидно, что так как вы разделили колоду на две части, их необходимо соединить. В коде это означает, что необходимо перечислить обе последовательности, полученные с помощью Take и Skip за один раз, применяя команду `interleaving` к элементам и создавая одну последовательность: колода карт, которая тасуется сейчас. Чтобы создать метод LINQ, который работает с двумя последовательностями, важно хорошо понимать принципы работы IEnumerable<T>.

Интерфейс IEnumerable<T> содержит один метод: GetEnumerator. Этот метод GetEnumerator возвращает объект, у которого есть метод для перехода к следующему элементу и свойство, которое возвращает текущий элемент в последовательности. С помощью этих двух членов вы выполните перебор всей коллекции и получение элементов. Метод Interleave будет реализован как метод итератора, поэтому вы не будете создавать и возвращать коллекцию, а примените описанный выше синтаксис `yield return`.

Так выглядит реализация этого метода:

```
public static IEnumerable<T> InterleaveSequenceWith<T>
    (this IEnumerable<T> first, IEnumerable<T> second)
{
    var firstIter = first.GetEnumerator();
    var secondIter = second.GetEnumerator();

    while (firstIter.MoveNext() && secondIter.MoveNext())
    {
        yield return firstIter.Current;
        yield return secondIter.Current;
    }
}
```

Теперь, добавив в проект этот метод, вернитесь к методу `Main` и один раз перетасуйте колоду:

```
// Program.cs
public static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                       from r in Ranks()
                       select new { Suit = s, Rank = r };

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    var top = startingDeck.Take(26);
    var bottom = startingDeck.Skip(26);
    var shuffle = top.InterleaveSequenceWith(bottom);

    foreach (var c in shuffle)
    {
        Console.WriteLine(c);
    }
}
```

Сравнение

Через сколько тасовок колода снова соберется в исходном порядке? Чтобы узнать это, вам нужно

написать метод, который проверяет равенство двух последовательностей. Создав такой метод, вы поместите код тасовки колоды в цикл, в котором будете проверять, расположены ли карты в правильном порядке.

Метод, который сравнивает две последовательности, будет очень простым. По структуре он похож на метод, который мы создали для тасовки колоды. Но теперь вместо команды `yield return`, которая возвращает элементы, вы будете сравнивать элементы каждой последовательности. Если перечисление последовательности завершилось и все элементы попарно совпадают, то последовательности считаются одинаковыми:

```
public static bool SequenceEquals<T>
    (this IEnumerable<T> first, IEnumerable<T> second)
{
    var firstIter = first.GetEnumerator();
    var secondIter = second.GetEnumerator();

    while (firstIter.MoveNext() && secondIter.MoveNext())
    {
        if (!firstIter.Current.Equals(secondIter.Current))
        {
            return false;
        }
    }

    return true;
}
```

Здесь мы видим в действии второй принцип LINQ: терминальные методы. Они принимают последовательность в качестве входных данных (или две последовательности, как в нашем примере) и возвращают скалярное значение. В цепочке методов для запроса LINQ терминальные методы всегда используются последними, отсюда и название "терминальный".

Этот принцип мы применяем при определении того, находится ли колода в исходном порядке. Поместите код тасовки в цикл, который будет останавливаться в том случае, когда порядок последовательности восстановлен. Для проверки примените метод `SequenceEquals()`. Как вы уже поняли, этот метод всегда будет последним в любом запросе, поскольку он возвращает одиночное значение, а не последовательность:

```
// Program.cs
static void Main(string[] args)
{
    // Query for building the deck

    // Shuffling using InterleaveSequenceWith<T>();

    var times = 0;
    // We can re-use the shuffle variable from earlier, or you can make a new one
    shuffle = startingDeck;
    do
    {
        shuffle = shuffle.Take(26).InterleaveSequenceWith(shuffle.Skip(26));

        foreach (var card in shuffle)
        {
            Console.WriteLine(card);
        }
        Console.WriteLine();
        times++;

    } while (!startingDeck.SequenceEquals(shuffle));

    Console.WriteLine(times);
}
```

Выполните код и обратите внимание на то, как выполняется переупорядочивание колоды при каждой тасовке. После 8 тасовок (итераций цикла do-while), колода возвращается к исходной конфигурации, в которой она находилась при создании из начального запроса LINQ.

Оптимизация

Пример, который вы создали к этому моменту, выполняет *внутреннюю тасовку*, то есть первая и последняя карты колоды сохраняют свои позиции после каждой итерации. Давайте внесем одно изменение: вместо этого мы будем использовать *внешнюю тасовку*, при которой все 52 карты изменяют свои позиции. Для этого колоду нужно собирать так, чтобы первой картой в колоде стала первая карта из нижней половины. Тогда самой нижней картой станет последняя карта из верхней половины колоды. Это простое изменение в одной строке кода. Обновите текущий запрос тасовки, переключив положения Take и Skip. Это поменяет местами нижнюю и верхнюю половины колоды:

```
shuffle = shuffle.Skip(26).InterleaveSequenceWith(shuffle.Take(26));
```

Снова запустите программу, и вы увидите, что для восстановления исходного порядка теперь требуется 52 итерации. Также вы могли обратить внимание, что по мере выполнения программы она заметным образом замедляется.

Для этого есть сразу несколько причин. Вы можете решить одну из самых существенных причин спада производительности — неэффективное использование *отложенного вычисления*.

Короче говоря, отложенное вычисление означает, что вычисление инструкции не выполняется, пока не понадобится ее значение. Запросы LINQ — это инструкции, которые обрабатываются отложенным образом. Последовательности создаются только тогда, когда происходит обращение к их элементам. Обычно это дает LINQ огромное преимущество. Но в некоторых программах, таких как в нашем примере, это приводит к экспоненциальному росту времени выполнения.

Помните, что мы создали исходную колоду с помощью запроса LINQ. Каждая последующая тасовка выполняет три запроса LINQ к колоде, полученной на предыдущем этапе. И все эти запросы выполняются отложенно. В частности, это означает, что запросы выполняются каждый раз при

обращении к последовательности. Таким образом, пока вы доберетесь до 52-й итерации, исходная колода будет заново создана очень много раз. Чтобы наглядно это продемонстрировать, давайте создадим журнал выполнения. Затем вы исправите эту проблему.

В вашем файле `Extensions.cs` введите или скопируйте приведенный ниже метод. Этот метод расширения создает файл с именем `debug.log` в каталоге проекта и записывает в файл журнала, какой запрос выполняется в данный момент. Этот метод расширения можно добавить к любому запросу, чтобы зафиксировать его выполнение.

```
public static IEnumerable<T> LogQuery<T>
    (this IEnumerable<T> sequence, string tag)
{
    // File.AppendText creates a new file if the file doesn't exist.
    using (var writer = File.AppendText("debug.log"))
    {
        writer.WriteLine($"Executing Query {tag}");
    }

    return sequence;
}
```

Вы увидите красную волнистую линию под `File`, означающую, что его не существует. Он не будет компилироваться, поскольку компилятор не знает, что такое `File`. Чтобы решить эту проблему, добавьте следующую строку кода под самой первой строкой в `Extensions.cs`:

```
using System.IO;
```

Это позволит решить проблему, и красная линия ошибки исчезнет.

Теперь давайте дополним определение каждого запроса сообщением для журнала:

```
// Program.cs
public static void Main(string[] args)
{
    var startingDeck = (from s in Suits().LogQuery("Suit Generation")
                        from r in Ranks().LogQuery("Rank Generation")
                        select new { Suit = s, Rank = r }).LogQuery("Starting Deck");

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    Console.WriteLine();
    var times = 0;
    var shuffle = startingDeck;

    do
    {
        // Out shuffle
        /*
        shuffle = shuffle.Take(26)
            .LogQuery("Top Half")
            .InterleaveSequenceWith(shuffle.Skip(26)
            .LogQuery("Bottom Half"))
            .LogQuery("Shuffle");
        */

        // In shuffle
        shuffle = shuffle.Skip(26).LogQuery("Bottom Half")
                .InterleaveSequenceWith(shuffle.Take(26).LogQuery("Top Half"))
                .LogQuery("Shuffle");

        foreach (var c in shuffle)
        {
            Console.WriteLine(c);
        }

        times++;
        Console.WriteLine(times);
    } while (!startingDeck.SequenceEquals(shuffle));

    Console.WriteLine(times);
}
```

Обратите внимание, что запись в журнал не нужно выполнять при обращении к запросу. Она выполняется только при создании исходного запроса. Программа по-прежнему работает очень долго, но теперь вы хорошо видите, почему. Если у вас не хватит терпения выполнять внешнюю тасовку с ведением журнала, переключите программу обратно на внутреннюю тасовку. На ней вы также заметите влияние отложенного вычисления. За один запуск программа выполняет 2592 запроса, если учитывать все создания мастей и достоинств.

Вы можете повысить производительность кода, чтобы уменьшить количество выполнений. Простой способ исправить — *кэшировать* результаты исходного запроса LINQ, который создает колоду карт. В настоящее время вы выполняете запросы снова и снова каждый раз, когда цикл do-while проходит через итерацию, повторно создавая и перетасовывая колоду карт. Чтобы кэшировать колоду карт, вы можете использовать методы LINQ ToArray и ToList. Когда вы добавляете их в запросы, они будут выполнять те действия, которые вы указали, но теперь они будут хранить результаты в массиве или списке в зависимости от того, какой метод вы вызовете. Добавьте метод LINQ ToArray в оба запроса и снова запустите программу:

```
public static void Main(string[] args)
{
    var startingDeck = (from s in Suits().LogQuery("Suit Generation")
                        from r in Ranks().LogQuery("Value Generation")
                        select new { Suit = s, Rank = r })
                        .LogQuery("Starting Deck")
                        .ToArray();

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    Console.WriteLine();

    var times = 0;
    var shuffle = startingDeck;

    do
    {
        /*
        shuffle = shuffle.Take(26)
            .LogQuery("Top Half")
            .InterleaveSequenceWith(shuffle.Skip(26).LogQuery("Bottom Half"))
            .LogQuery("Shuffle")
            .ToArray();
        */

        shuffle = shuffle.Skip(26)
            .LogQuery("Bottom Half")
            .InterleaveSequenceWith(shuffle.Take(26).LogQuery("Top Half"))
            .LogQuery("Shuffle")
            .ToArray();

        foreach (var c in shuffle)
        {
            Console.WriteLine(c);
        }

        times++;
        Console.WriteLine(times);
    } while (!startingDeck.SequenceEquals(shuffle));

    Console.WriteLine(times);
}
```

Теперь при внутренней тасовке выполняется всего 30 запросов. Переключите программу на внешнюю тасовку, и вы заметите аналогичное улучшение: теперь выполняется 162 запроса.

Обратите внимание, что этот пример лишь **демонстрирует** варианты использования, в которых отложенное вычисление приводит к проблемам с производительностью. Хотя очень важно знать, когда отложенное вычисление может повлиять на производительность кода, не менее важно понимать, что не все запросы должны выполняться упреждающе. Если не использовать ToArray, производительность снизится. Это связано с тем, что каждое новое расположение карт вычисляется на основе предыдущего расположения. Использование отложенного вычисления означает, что каждое расположение колоды строится с самого начала, из исходной колоды, включая вызов кода для создания `startingDeck`. Это создает огромный объем дополнительной работы.

На практике некоторые алгоритмы хорошо работают с упреждающим вычислением, а другие хорошо выполняются с отложенным вычислением. Для ежедневного использования отложенное вычисление обычно дает более хороший результат, если в качестве источника данных используется отдельный процесс, например база данных. Для баз данных отложенное вычисление позволяет сложным запросам

выполнять только один круговой путь к процессу базы данных и обратно к оставшемуся коду. LINQ является гибким, независимо от того, используете ли вы отложенное или упреждающее вычисление, поэтому измерьте процессы и выберите тип вычислений, который обеспечивает наилучшую производительность.

Заключение

В этом проекте вы изучили:

- использование запросов LINQ для агрегирования данных в осмысленную последовательность;
- запись методов расширения для добавления собственных пользовательских функций в запросы LINQ;
- поиск областей в коде, где могут возникнуть проблемы с производительностью наших запросов LINQ, например снижение скорости;
- упреждающее и отложенное вычисление в отношении запросов LINQ и их влияние на производительность запросов.

Помимо LINQ вы узнали об использовании метода, который иллюзионисты используют для карточных фокусов. Они используют тасовку по методу Фаро, потому что она позволяет хорошо контролировать положение каждой карты в колоде. Теперь, когда вы все это знаете, не рассказывайте это остальным!

Дополнительные сведения о LINQ см. в следующих статьях:

- LINQ
- Введение в LINQ
- Основные операции запросов LINQ (C#)
- Преобразования данных с помощью LINQ (C#)
- Синтаксис запросов и синтаксис методов в LINQ (C#)
- Возможности C#, поддерживающие LINQ

Использование атрибутов в C#

18.12.2021 • 6 minutes to read

Атрибуты предоставляют возможность декларативно связать информацию с кодом. Также этот элемент можно многократно использовать повторно для разнообразных целевых объектов.

Рассмотрим для примера атрибут `[Obsolete]`. Его можно применять к классам, структурам, методам, конструкторам и т. д. Он *объявляет*, что соответствующий элемент является устаревшим. Компилятор C# проверят наличие этого атрибута и выполняет некоторые действия, если он присутствует.

В этом руководстве мы покажем вам, как можно добавить атрибуты в код, как создавать и применять собственные атрибуты, а также использовать некоторые встроенные атрибуты .NET Core.

Предварительные требования

Компьютер должен быть настроен для выполнения .NET Core. Инструкции по установке см. на странице скачиваемых файлов .NET Core. Это приложение можно запустить в ОС Windows, Ubuntu Linux, macOS или в контейнере Docker. Вам потребуется редактор кода, но вы можете выбрать любой привычный для вас. В примерах ниже используется кроссплатформенный редактор Visual Studio Code с открытым исходным кодом. Вы можете заменить его на любое другое средство, с которым вам удобно работать.

Создание приложения

Теперь, когда вы установили все нужные средства, создайте новое приложение .NET Core. Чтобы использовать генератор из командной строки, выполните следующую команду в любой оболочке:

```
dotnet new console
```

Эта команда создает простые файлы проекта для .NET Core. Нужно также выполнить команду `dotnet restore`, чтобы восстановить зависимости, необходимые для компиляции проекта.

Вам не нужно выполнять команду `dotnet restore`, так как она выполняется неявно всеми командами, которые требуют восстановления, например `dotnet new`, `dotnet build`, `dotnet run`, `dotnet test`, `dotnet publish` и `dotnet pack`. Чтобы отключить неявное восстановление, используйте параметр `--no-restore`.

Команду `dotnet restore` по-прежнему удобно использовать в некоторых сценариях, где необходимо явное восстановление, например в сборках с использованием непрерывной интеграции в Azure DevOps Services или системах сборки, где требуется явно контролировать время восстановления.

Сведения об управлении веб-каналами NuGet см. в документации по `dotnet restore`.

Чтобы выполнить программу, используйте `dotnet run`. Она выведет в консоль сообщение "Hello, World".

Добавление атрибутов к коду

В C# атрибуты представляют собой классы, наследующие от базового класса `Attribute`. Любой класс, который наследует от `Attribute`, можно использовать как своего рода "тег" на другие части кода. Например, существует атрибут с именем `ObsoleteAttribute`. С его помощью можно обозначить, что код устарел и больше не должен использоваться. Этот атрибут можно поместить в класс, используя квадратные скобки.

```
[Obsolete]
public class MyClass
{
}
```

Обратите внимание, что полное название класса — `ObsoleteAttribute`, но в коде достаточно указать только `[Obsolete]`. Это стандартное соглашение в C#. При желании вы можете использовать полное имя `[ObsoleteAttribute]`.

Когда вы отмечаете класс как устаревший, желательно предоставить некоторые сведения о том, *почему* он устарел и (или) *что* можно использовать вместо него. Для этого передайте в атрибут Obsolete строковый параметр.

```
[Obsolete("ThisClass is obsolete. Use ThisClass2 instead.")]
public class ThisClass
{
}
```

Эта строка передается в качестве аргумента в конструктор `ObsoleteAttribute`, как если бы вы использовали синтаксис `var attr = new ObsoleteAttribute("some string")`.

В конструкторе атрибута можно использовать в качестве параметров только простые типы и литералы `bool, int, double, string, Type, enums, etc` и массивы этих типов. Нельзя использовать переменные или выражения. Но вы можете свободно использовать позиционные или именованные параметры.

Как создать собственный атрибут

Чтобы создать собственный атрибут, достаточно унаследовать его от базового класса `Attribute`.

```
public class MySpecialAttribute : Attribute
{
}
```

Добавив такой код, вы сможете использовать `[MySpecial]` (или `[MySpecialAttribute]`) в качестве атрибута в любом месте кода.

```
[MySpecial]
public class SomeOtherClass
{
}
```

Атрибуты в библиотеке базовых классов .NET, например `ObsoleteAttribute`, активируют определенный действия компилятора. Но созданные вами атрибуты сами по себе лишь выполняют роль метаданных и не влекут за собой исполнение какого-либо кода в классе атрибута. Вы должны проверять эти метаданные и выполнять необходимые действия в другой части кода (подробнее об этом написано дальше в этом руководстве).

Здесь есть один подвох, которого следует остерегаться. Как упоминалось выше, в качестве аргументов при использовании атрибутов можно передавать только некоторые определенные типы. Но компилятор C# не помешает вам указать другие параметры при создании типа атрибута. В следующем примере кода я создаю атрибут с конструктором, который отлично компилируется.

```
public class GotchaAttribute : Attribute
{
    public GotchaAttribute(Foo myClass, string str) {
    }
}
```

Но объект с таким конструктором вы не сможете использовать в роли атрибута.

```
[Gotcha(new Foo(), "test")] // does not compile
public class AttributeFail
{
}
```

Такой код вызовет ошибку компиляции, например такую:

```
Attribute constructor parameter 'myClass' has type 'Foo', which is not a valid attribute parameter type
```

Как ограничить использование атрибута

Атрибуты можно использовать для разных целевых объектов. В примере выше мы применили их для классов, но целевым объектом может быть любой из этого списка:

- Сборка
- Класс
- Конструктор
- Делегат
- Перечисление
- Событие
- Поле
- универсальный параметр;
- Интерфейс
- Метод
- Модуль
- Параметр
- Свойство.
- Возвращаемое значение
- Структура

Когда вы создаете класс атрибута, C# по умолчанию позволяет использовать этот атрибут для любого из допустимых целевых объектов. Если вы хотите, чтобы атрибут можно было использовать только для некоторых из целевых объектов, используйте `AttributeUsageAttribute` в классе атрибута. Да-да, именно так, атрибут для атрибута!

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
public class MyAttributeForClassAndStructOnly : Attribute
{
}
```

Если вы попробуете применить описанный выше атрибут для сущности, которая не является классом или структурой, вы получите ошибку компиляции такого рода:

```
Attribute 'MyAttributeForClassAndStructOnly' is not valid on this declaration type. It is only valid on
'class, struct' declarations
```

```
public class Foo
{
    // if the below attribute was uncommented, it would cause a compiler error
    // [MyAttributeForClassAndStructOnly]
    public Foo()
    { }
}
```

Как использовать атрибуты, прикрепленные к элементу кода

Атрибуты выполняют роль метаданных. Без применения внешних сил они по сути ничего не делают.

Чтобы находить атрибуты и реагировать на них, обычно используется отражение. Мы не будем здесь подробно описывать отражения, ограничимся лишь основной идеей: отражение позволяет написать на C# код, который проверяет другой код.

Например, с помощью отражения можно получить сведения о классе (добавьте `using System.Reflection;` в начало кода):

```
TypeInfo typeInfo = typeof(MyClass).GetTypeInfo();
Console.WriteLine("The assembly qualified name of MyClass is " + typeInfo.AssemblyQualifiedName);
```

Этот код выведет такие данные:

```
The assembly qualified name of MyClass is ConsoleApplication.MyClass, attributes, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null
```

Если у вас есть объект `TypeInfo` (или `MemberInfo`, `FieldInfo` и т. д.), вы можете использовать метод `GetCustomAttributes`. Он возвращает коллекцию объектов `Attribute`. Можно также использовать `GetCustomAttribute`, указав тип атрибута.

Ниже вы видите пример использования `GetCustomAttributes` для экземпляра `MemberInfo` класса `MyClass` (как мы продемонстрировали ранее, он имеет атрибут `[Obsolete]`).

```
var attrs = typeInfo.GetCustomAttributes();
foreach(var attr in attrs)
    Console.WriteLine("Attribute on MyClass: " + attr.GetType().Name);
```

Этот код выведет в консоль текст: `Attribute on MyClass: ObsoleteAttribute`. Попробуйте добавить другие атрибуты для `MyClass`.

Обратите особое внимание, что к таким объектам `Attribute` применяется отложенное создание экземпляров. При вызове `GetCustomAttribute` или `GetCustomAttributes` экземпляры не создаются. Кроме того, они создаются заново при каждом обращении. Выполнив `GetCustomAttributes` два раза подряд, вы получите два различных экземпляра `ObsoleteAttribute`.

Популярные атрибуты в библиотеке базовых классов (BCL)

Атрибуты используются многими средствами и платформами. NUnit использует такие атрибуты, как `[Test]` и `[TestFixture]`, которые нужны для средства тестового запуска NUnit. ASP.NET MVC использует такие атрибуты, как `[Authorize]`, и предоставляет платформу фильтра действий, которая позволяет использовать перекрестные функции для действий MVC. PostSharp использует синтаксис атрибутов для реализации аспектно-ориентированного программирования на языке C#.

Ниже приведены несколько важных атрибутов, используемых в библиотеках базовых классов .NET Core.

- `[Obsolete]`. Этот атрибут мы уже использовали в примерах выше. Он размещен в пространстве имен `System`. С его помощью удобно создавать декларативную документацию об изменении кодовой базы. К нему можно добавить строковое сообщение, а дополнительный логический параметр позволяет повысить уровень сообщений компилятора с предупреждения до ошибки.
- `[Conditional]`. Этот атрибут находится в пространстве имен `System.Diagnostics`. Его может применять к методам или классам атрибутов. В его конструктор необходимо передать строку. Если эта строка не совпадает с директивой `#define`, то компилятор C# будет удалять все вызовы этого метода (но не сам метод). Обычно это используется для целей отладки или диагностики.
- `[CallerMemberName]`. Этот атрибут можно применить для параметров. Он размещен в пространства имен `System.Runtime.CompilerServices`. Этот атрибут позволяет передать имя метода добавления, который вызывает другой метод. Обычно он используется для устранения "волшебных строк" при реализации INotifyPropertyChanged в различных платформах взаимодействия с пользователем. Например:

```
public class MyUIClass : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    public void RaisePropertyChanged([CallerMemberName] string propertyName = null)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }

    private string _name;
    public string Name
    {
        get { return _name;}
        set
        {
            if (value != _name)
            {
                _name = value;
                RaisePropertyChanged();   // notice that "Name" is not needed here explicitly
            }
        }
    }
}
```

В приведенном выше коде не обязательно использовать литеральную строку `"Name"`. Это помогает предотвратить ошибки, связанные с опечатками, а также позволяет плавно выполнять рефакторинг и (или) переименование.

Сводка

Атрибуты позволяют реализовать в C# возможности декларативного синтаксиса. Но они являются разновидностью метаданных и сами по себе не выполняют действия.

Ссылочные типы, допускающие значение null

18.12.2021 • 15 minutes to read

До C# 8.0 все ссылочные типы допускали значение NULL. *Ссылочные типы, допускающие значение NULL*, относятся к группе функций, появившихся в C# 8.0, которые можно использовать, чтобы снизить вероятность возникновения исключения System.NullReferenceException в среде выполнения. *Ссылочные типы, допускающие значения NULL*, содержат три функции, которые помогают избежать возникновения этих исключений, а также возможность явно помечать ссылочный тип как *допускающий значение NULL*.

- Улучшенный анализ статического потока, который определяет, может ли переменная быть `null`, перед ее разыменованием.
- Атрибуты, аннотирующие API, чтобы анализ потока определял *состояние NULL*.
- Аннотации переменных, которые разработчики используют для явного объявления предполагаемого *состояния NULL* для переменной.

Анализ состояния NULL и аннотации переменных по умолчанию отключены для существующих проектов. Это означает, что все ссылочные типы продолжают допускать значение NULL. Начиная с .NET 6 они включены по умолчанию для *новых* проектов. Сведения о включении этих функций путем объявления *контекста аннотации, допускающего значение NULL*, см. в статье Контексты, допускающие значение NULL.

В оставшейся части этой статьи описывается, как эти три функциональные области позволяют создавать предупреждения, когда код **разыменовывает** значение `null`. Разыменование переменной означает доступ к одному из ее членов с помощью оператора `.` (точка), как показано в следующем примере:

```
string message = "Hello, World!";
int length = message.Length; // dereferencing "message"
```

При разыменовании переменной, значение которой равно `null`, среда выполнения создает исключение System.NullReferenceException.

Анализ состояния NULL

***Анализ состояния** NULL отслеживает состояние NULL* ссылок. Этот статический анализ выдает предупреждения, когда код разыменовывает `null`. Эти предупреждения можно устранить, чтобы избежать случаев, когда среда выполнения выдает исключение System.NullReferenceException. Компилятор использует статический анализ для определения *состояния NULL* переменной. Переменная может быть *not-null* или *maybe-null*. Компилятор определяет, что переменная *not-null*, двумя способами:

1. Переменной присваивается значение, которое точно *not-null*.
2. Переменная была проверена на предмет `null` и не изменялась с момента проверки.

Любая переменная, которую компилятор не определил как *not-null*, считается *maybe-null*. Анализ выдает предупреждения в ситуациях, когда вы можете случайно разыменовать значение `null`. Компилятор создает предупреждения на основе *состояния NULL*.

- Если переменная *not-null*, ее можно безопасно разыменовать.
- Если это переменная *maybe-null*, ее необходимо проверить, чтобы убедиться, что она не равна `null`, перед разыменованием.

Рассмотрим следующий пример.

```
string message = null;

// warning: dereference null.
Console.WriteLine($"The length of the message is {message.Length}");

var originalMessage = message;
message = "Hello, World!";

// No warning. Analysis determined "message" is not null.
Console.WriteLine($"The length of the message is {message.Length}");

// warning!
Console.WriteLine(originalMessage.Length);
```

В предыдущем примере компилятор определяет, что переменная `message` *maybe-null* при выводе первого сообщения. Для второго сообщения предупреждение не выводится. Последняя строка кода выдает предупреждение, поскольку `originalMessage` может иметь значение NULL. В следующем примере показано более практичное использование для обхода дерева узлов до корня с обработкой каждого узла во время обхода:

```
void FindRoot(Node node, Action<Node> processNode)
{
    for (var current = node; current != null; current = current.Parent)
    {
        processNode(current);
    }
}
```

Предыдущий код не создает предупреждений для разыменования переменной `current`. Статический анализ определяет, что переменная `current` никогда не будет разыменована, если она *maybe-null*. Переменная `current` проверяется на `null` до доступа к `current.Parent` и перед передачей `current` в действие `ProcessNode`. В предыдущих примерах показано, как компилятор определяет *состояние NULL* для локальных переменных при инициализации, назначении или сравнении с `null`.

> **NOTE**
> В C# 10 добавлен ряд улучшений для определенного присваивания и анализа состояния NULL. При обновлении до C# 10 сократится число ложноположительных результатов с предупреждением о NULL. Дополнительные сведения об улучшениях см. в статье о спецификации функций для определенного присваивания.

Атрибуты в сигнатурах API

Анализ состояния NULL требует указаний от разработчиков для понимания семантики API. Некоторые API обеспечивают проверку значений NULL и должны изменять *состояние NULL* переменной с *maybe-null* на *not-null*. Другие API возвращают выражения, которые являются *not-null* или *maybe-null* в зависимости от *состояния NULL* входных аргументов. Для примера рассмотрим следующий код, в котором отображается сообщение:

```
public void PrintMessage(string message)
{
    if (!string.IsNullOrWhiteSpace(message))
    {
        Console.WriteLine($"{DateTime.Now}: {message}");
    }
}
```

Основываясь на проверке, любой разработчик будет считать этот код безопасным и не должен создавать предупреждения. Компилятор не знает, что `IsNullOrWhiteSpace` обеспечивает проверку значения NULL. Атрибуты применяются для информирования компилятора о том, что переменная `message` является *not-null*, только в том случае, если `IsNullOrWhiteSpace` возвращает `false`. В предыдущем примере сигнатура содержит `NotNullWhen`, чтобы показать состояние NULL для `message`:

```
public static bool IsNullOrWhiteSpace([NotNullWhen(false)] string message);
```

Атрибуты предоставляют подробные сведения о состоянии NULL аргументов, возвращаемых значений и элементов экземпляра объекта, используемого для вызова элемента. Подробные сведения о каждом атрибуте можно найти в справочнике по языку об атрибутах ссылки, допускающих значения NULL. Все API среды выполнения .NET были аннотированы в .NET 5. Статический анализ можно улучшить, аннотировав API, чтобы предоставить семантическую информацию о *состоянии NULL* аргументов и возвращаемых значений.

Аннотации для переменных, допускающих значения NULL

Анализ *состояния NULL* надежен для большинства переменных. Компилятору требуются дополнительные сведения о переменных элементов. Компилятор не может делать предположения о порядке, в котором осуществляется доступ к открытым элементам. Доступ к любому открытому элементу можно получить в любом порядке. Любой из доступных конструкторов можно использовать для инициализации объекта. Если для поля элемента можно задать значение `null`, в начале каждого метода компилятор должен предположить, что его *состояние NULL — maybe-null*.

Используйте аннотации, которые могут объявлять, является ли переменная **ссылочным типом, допускающим значение** NULL, или **ссылочным типом, не допускающим значения** NULL. Эти аннотации указывают на *состояние NULL* для переменных:

- **Ссылка не должна иметь значение** NULL. Состоянием по умолчанию для ссылочной переменной, не допускающей значение NULL, является *не равно NULL*. Компилятор принудительно применяет правила, которые гарантируют, что можно разыменовать эти переменные без предварительной проверки того, что они не имеют значение NULL:
  - Переменную необходимо инициализировать со значением, отличным от NULL.
  - Переменной не может быть присвоено значение `null`. Компилятор выдает предупреждение, когда код присваивает выражение *maybe-null* переменной, которая не должна иметь значение NULL.
- **Ссылка может иметь значение** NULL. Состоянием по умолчанию для ссылочной переменной, допускающей значение NULL, является *может быть NULL*. Компилятор применяет правила, чтобы убедиться в правильности проверки для ссылки `null`:
  - Переменная может быть разыменована только в том случае, когда компилятор может гарантировать, что значение не равно `null`.
  - Эти переменные могут быть инициализированы со значением по умолчанию `null`, и им можно присвоить значение `null` в другом коде.
  - Компилятор не выдает предупреждения, когда код присваивает выражение *maybe-null*

переменной, которая может иметь значение NULL.

Любая ссылочная переменная, которая не должна быть равна `null`, имеет *состояние NULL not-null*. Любая ссылочная переменная, которая может иметь значение `null`, изначально имеет *состояние NULL maybe-null*.

Ссылочный тип, допускающий значение NULL использует тот же синтаксис, что и тип значения, допускающего значение NULL: к типу переменной добавляется `?`. Например, следующее объявление переменной представляет строковую переменную, допускающую значение NULL, `name`:

```
string? name;
```

Любая переменная, где `?` не добавляется к имени типа, является **ссылочным типом, не допускающим значение** NULL. Сюда входят все переменные ссылочных типов в существующем коде, если вы включили эту функцию. Однако любые неявные типизированные локальные переменные (объявленные с помощью `var`) являются **ссылочными типами, допускающими значение** NULL. Как было показано в предыдущих разделах, статический анализ определяет *состояние NULL* локальных переменных, чтобы узнать, являются ли они *maybe-null*.

Иногда необходимо переопределить предупреждение, если известно, что переменная не имеет значение NULL, но компилятор определяет ее *состояние NULL* как *maybe-null*. Используйте оператор, допускающий значение NULL `!` перед именем переменной, чтобы принудительно задать для *состояния NULL not-null*. Например, если вы знаете, что переменная `name` не имеет значение `null`, а компилятор выдает предупреждение, напишите следующий код, чтобы переопределить анализ компилятора:

```
name!.Length;
```

Ссылочные типы, допускающие значение NULL, и типы значений, допускающие значение NULL, предоставляют аналогичную семантическую концепцию: переменная может представлять значение или объект, либо эта переменная может быть `null`. Однако ссылочные типы, допускающие значение NULL, и типы значений, допускающие значение NULL, реализуются по-разному: типы значений, допускающие значение NULL, реализуются с помощью System.Nullable<T>, а ссылочные типы, допускающие значение NULL, реализуются атрибутами, которые считывает компилятор. Например, `string?` и `string` представлены одним и тем же типом: System.String. Однако `int?` и `int` представлены `System.Nullable<System.Int32>` и System.Int32 соответственно.

Универсальные шаблоны

Для универсальных шаблонов требуются подробные правила для обработки `T?` для любых типов параметров `T`. Правила должны быть подробными, учитывая журнал и другую реализацию типа значения, допускающего значение NULL, и ссылочного типа, допускающего значение NULL. Типы значений, допускающие значение NULL, реализуются с помощью структуры System.Nullable<T>. Ссылочные типы, допускающие значения NULL, реализуются как аннотации типа, предоставляющие компилятору семантические правила.

В C# 8.0 при использовании `T?` без ограничений для `T` значением `struct` или `class` компиляция была невозможна. Это позволило компилятору четко интерпретировать `T?`. Это ограничение было удалено в C# 9.0 благодаря определению следующих правил для неограниченного параметра `T`.

- Если аргумент типа для `T` является ссылочным типом, `T?` ссылается на соответствующий тип ссылки, допускающий значение NULL. Например, если `T` равен `string`, то `T?` — `string?`.
- Если аргумент типа для `T` является типом значения, `T?` ссылается на тот же тип значения, `T`.

Например, если `T` имеет значение `int`, то `T?` также является `int`.

- Если аргумент типа для `T` является ссылочным типом, допускающим значение NULL, `T?` ссылается на тот же ссылочный тип, допускающий значение NULL. Например, если `T` равен `string?`, то `T?` — `string?`.
- Если аргумент типа для `T` является типом значения, допускающим значение NULL, `T?` ссылается на тот же тип значения, допускающий значение NULL. Например, если `T` равен `int?`, то `T?` — `int?`.

Для возвращаемых значений `T?` эквивалентно `[MaybeNull]T`; для значений аргументов аргумент `T?` эквивалентен `[AllowNull]T`. Дополнительные сведения см. в статье об атрибутах для анализа состояния NULL в справочнике по языку.

Можно указать другое поведение с помощью ограничений.

- Ограничение `class` означает, что `T` должен быть ссылочным типом, не допускающим значения NULL (например, `string`). Компилятор выдает предупреждение при использовании ссылочного типа, допускающего значение NULL, например `string?` для `T`.
- Это ограничение `class?` означает, что `T` должен быть ссылочным типом, не допускающим значение NULL (`string`), или ссылочным типом, допускающим значение NULL (например, `string?`). Если параметр типа является ссылочным типом, допускающим значение NULL, например `string?`, выражение `T?` ссылается на тот же ссылочный тип, допускающий значение NULL, например `string?`.
- Ограничение `notnull` означает, что `T` должен быть ссылочным типом, не допускающим значения NULL, или типом значения, не допускающим значения NULL. Если для параметра типа используется ссылочный тип, допускающий значение NULL, или тип значения, допускающий значение NULL, компилятор выдает предупреждение. Кроме того, если `T` является типом значения, возвращаемое значение относится к этому типу значения, а не к соответствующему типу значения, допускающему значение NULL.

Эти ограничения помогают компилятору получить дополнительные сведения о том, как будет использоваться `T`. Это помогает разработчикам выбирать тип для `T` и обеспечивает улучшенный анализ *состояния NULL* при использовании экземпляра универсального типа.

Контексты допустимости значения NULL

При включении в существующей базе кода новые функции, которые защищают от исключения System.NullReferenceException, могут нарушать работу:

- Все явно типизированные ссылочные переменные обрабатываются как ссылочные типы, не допускающие значения NULL.
- Значение ограничения `class` в универсальных шаблонах изменилось на ссылочный тип, не допускающий значения NULL.
- В связи с новыми правилами создаются новые предупреждения.

Для использования этих функций их необходимо явно включить в существующих проектах. Это позволяет предоставить путь миграции и сохранить обратную совместимость. Контексты допустимости значения NULL детально контролируют, как компилятор интерпретирует переменные ссылочного типа. **Контекст аннотаций**, **допускающий значение** NULL, определяет поведение компилятора. Существует четыре значения для **контекста аннотаций**, **допускающего значение** NULL:

- *disabled*: компилятор ведет себя так же, как в C# 7.3 и более ранних версиях:
  - Предупреждения о значении NULL отключены.
  - Все переменные ссылочного типа являются ссылочными типами, допускающими значение NULL.
  - Нельзя объявить переменную как ссылочный тип, допускающий значение NULL, с помощью суффикса `?` для типа.

    - Можно использовать оператор, допускающий NULL, `!`, но это ни на что не повлияет.
- *enabled*: компилятор включает анализ всех пустых ссылок и все языковые функции.
    - Включены все новые предупреждения о значениях NULL.
    - Для объявления ссылочного типа, допускающего значение NULL, можно использовать суффикс `?`.
    - Все остальные переменные ссылочного типа являются ссылочными типами, не допускающими значение NULL.
    - Оператор, допускающий значение NULL, подавляет предупреждения о возможном назначении `null`.
- *warnings*: компилятор выполняет полный анализ значений NULL и выдает предупреждения, если код может разыменовать `null`.
    - Включены все новые предупреждения о значениях NULL.
    - При использовании суффикса `?` для объявления ссылочного типа, допускающего значение NULL, выдается предупреждение.
    - Все переменные ссылочного типа могут иметь значение NULL. Однако элементы имеют *состояние NULL not-null* в открывающей фигурной скобке всех методов, если только они не объявлены с суффиксом `?`.
    - Может использовать оператор, допускающий NULL, `!`.
- *annotations*: компилятор не выполняет анализ значений NULL и не выдает предупреждения, если код может разыменовать `null`.
    - Все новые предупреждения о значении NULL отключены.
    - Для объявления ссылочного типа, допускающего значение NULL, можно использовать суффикс `?`.
    - Все остальные переменные ссылочного типа являются ссылочными типами, не допускающими значение NULL.
    - Можно использовать оператор, допускающий NULL, `!`, но это ни на что не повлияет.

Контекст аннотаций о допустимости значения NULL и контекст с предупреждениями о допустимости значения NULL можно задать для проекта с помощью элемента `<Nullable>` в файле *CSPROJ*. Этот элемент настраивает, как компилятор интерпретирует допустимость значений NULL для типов и какие предупреждения выдаются. В следующей таблице показаны допустимые значения и приводится сводка по задаваемым контекстам.

КОНТЕКСТ	ПРЕДУПРЕЖДЕНИЯ ОБ РАЗЫМЕНОВАНИИ	ПРЕДУПРЕЖДЕНИЯ О НАЗНАЧЕНИИ	ССЫЛОЧНЫЕ ТИПЫ	СУФФИКС `?`	ОПЕРАТОР `!`
`disabled`	Выключено	Выключено	Все допускают значение NULL	Не может использоваться	Не оказывает влияния
`enabled`	Активировано	Активировано	Не допускают значение NULL, если не объявлены с помощью `?`	Объявляет тип, допускающий значение NULL	Подавляет предупреждения о возможном назначении `null`

КОНТЕКСТ	ПРЕДУПРЕЖДЕНИЯ ОБ РАЗЫМЕНОВАНИИ	ПРЕДУПРЕЖДЕНИЯ О НАЗНАЧЕНИИ	ССЫЛОЧНЫЕ ТИПЫ	СУФФИКС `?`	ОПЕРАТОР `!`
`warnings`	Активировано	Неприменимо	Все допускают значение NULL, но при открывающих фигурных скобках методов элементы считаются *не NULL*	Выдает предупреждение	Подавляет предупреждения о возможном назначении `null`
`annotations`	Выключено	Выключено	Не допускают значение NULL, если не объявлены с помощью `?`	Объявляет тип, допускающий значение NULL	Не оказывает влияния

Переменные ссылочного типа в коде, скомпилированном в версиях до C# 8, или в *отключенном* контексте, имеют свойство *nullable-oblivious*. Можно назначить литерал `null` или переменную *может быть NULL* переменной, которая имеет свойство *nullable oblivious*. Однако по умолчанию для переменной со свойством *nullable oblivious* установлено состояние *не равно NULL*.

Вы можете выбрать оптимальный параметр для своего проекта:

- Выберите *disabled* для устаревших проектов, которые не нужно обновлять на основе диагностики или новых функций.
- Выберите *warnings*, чтобы определить, где код может выдавать исключения System.NullReferenceException. Вы можете устранить эти предупреждения перед изменением кода, чтобы включить ссылочные типы, не допускающие значения NULL.
- Выберите *annotations*, чтобы выразить свое намерение, прежде чем включать предупреждения.
- Выберите *enabled* для новых проектов и активных проектов, в которых необходима защита от исключений пустых ссылок.

Пример:

```
<Nullable>enable</Nullable>
```

Также можно использовать директивы для задания этих контекстов в любом месте в исходном коде. Они наиболее полезны при переносе большой базы кода.

- `#nullable enable`. Устанавливает контекст с заметками о допустимости значений NULL и контекст с предупреждениями о допустимости значений NULL в режим **включен**.
- `#nullable disable`. Устанавливает контекст с заметками о допустимости значений NULL и контекст с предупреждениями о допустимости значений NULL в режим **отключен**.
- `#nullable restore`. Восстанавливает контекст с заметками о допустимости значений NULL и контекст с предупреждениями о допустимости значений NULL в соответствии с параметрами проекта.
- `#nullable disable warnings`. Устанавливает контекст с предупреждениями о допустимости значения NULL в режим **отключен**.
- `#nullable enable warnings`. Устанавливает контекст с предупреждениями о допустимости значения NULL в режим **включен**.

- `#nullable restore warnings`. Восстанавливает контекст с предупреждениями о допустимости NULL в соответствии с параметрами проекта.
- `#nullable disable annotations`. Устанавливает контекст с заметками о допустимости значения NULL в режим **отключен**.
- `#nullable enable annotations`. Устанавливает контекст с заметками о допустимости значения NULL в режим **включен**.
- `#nullable restore annotations`. Восстанавливает контекст с предупреждениями о заметках в соответствии с параметрами проекта.

Для любой строки кода можно задать любое из следующих сочетаний:

КОНТЕКСТ ПРЕДУПРЕЖДЕНИЯ	КОНТЕКСТ АННОТАЦИЙ	ИСПОЛЬЗОВАНИЕ
проект по умолчанию	проект по умолчанию	По умолчанию
Включено	disabled	Исправление предупреждения при анализе
Включено	проект по умолчанию	Исправление предупреждения при анализе
проект по умолчанию	Включено	Добавление аннотации для типа
Включено	Включено	Код уже перенесен
disabled	Включено	Добавление аннотации к коду перед исправлением предупреждений
disabled	disabled	Добавление устаревшего кода в перенесенный проект
проект по умолчанию	disabled	Редко
disabled	проект по умолчанию	Редко

Эти девять сочетаний позволяют точно контролировать диагностику, выдаваемую компилятором для кода. Вы можете включить дополнительные компоненты в любой обновляемой области, не получая дополнительных предупреждений, которые вы еще не можете устранить.

IMPORTANT

Глобальный контекст, допускающий значения NULL, не применяется для созданных файлов кода. В любом случае контекст, допускающий значение NULL, *отключен* для любого исходного файла, помеченного как созданный. Это означает, что все интерфейсы API в создаваемых файлах не заносятся в заметки. Существует четыре способа пометки файла как созданного:

1. В файле. editorconfig укажите `generated_code = true` в разделе, который применяется к этому файлу.
2. Вставьте `<auto-generated>` или `<auto-generated/>` в комментарий в верхней части файла. Он может находиться в любой строке комментария, однако блок комментариев должен быть первым элементом в файле.
3. Имя файла следует начинать с *TemporaryGeneratedFile_*
4. В конце имени файла следует указать *.designer.cs*, *.generated.cs*, *.g.cs* или *.g.i.cs*.

Генераторы могут явно использовать директиву препроцессора `#nullable`.

По умолчанию контексты с заметками и предупреждениями о допустимости значения NULL **отключены**. Это означает, что существующий код компилируется без изменений и без создания новых предупреждений. Начиная с .NET 6 во все шаблоны новых проектов входит элемент `<Nullable>enable</Nullable>`.

Эти параметры предоставляют две отдельные стратегии для обновления существующей базы кода так, чтобы она могла использовать ссылочные типы, допускающие значение NULL.

Известные ошибки

Массивы и структуры, содержащие ссылочные типы, являются известными проблемами в ссылках, допускающих значения NULL, и статическом анализе, который определяет безопасность значения NULL. В обоих случаях ссылка, не допускающая значения NULL, может быть инициализирована как `null` без предупреждений.

Структуры

Структуре, которая содержит ссылочные типы, не допускающие значения NULL, может быть присвоено значение `default` без предупреждения. Рассмотрим следующий пример.

```
using System;

#nullable enable

public struct Student
{
    public string FirstName;
    public string? MiddleName;
    public string LastName;
}

public static class Program
{
    public static void PrintStudent(Student student)
    {
        Console.WriteLine($"First name: {student.FirstName.ToUpper()}");
        Console.WriteLine($"Middle name: {student.MiddleName?.ToUpper()}");
        Console.WriteLine($"Last name: {student.LastName.ToUpper()}");
    }

    public static void Main() => PrintStudent(default);
}
```

В предыдущем примере в `PrintStudent(default)` не возвращается предупреждение, хотя ссылочные типы `FirstName` и `LastName`, не допускающие значения NULL, имеют значение NULL.

Еще один более распространенный случай связан с работой с универсальными структурами. Рассмотрим следующий пример.

```
#nullable enable

public struct Foo<T>
{
    public T Bar { get; set; }
}

public static class Program
{
    public static void Main()
    {
        string s = default(Foo<string>).Bar;
    }
}
```

В предыдущем примере свойство `Bar` во время выполнения будет иметь значение `null` и присваивается строке, не допускающей значения NULL. При этом предупреждения не возвращаются.

Массивы

При использовании ссылочных типов, допускающих значения NULL, также могут возникать известные ошибки, связанные с массивами. Рассмотрим следующий пример, в котором не выдаются предупреждения:

```
using System;

#nullable enable

public static class Program
{
    public static void Main()
    {
        string[] values = new string[10];
        string s = values[0];
        Console.WriteLine(s.ToUpper());
    }
}
```

В предыдущем примере объявление массива показывает, что он содержит строки, не допускающие значения NULL, а все элементы инициализируются с использованием значения `null`. После этого переменной `s` присваивается значение `null` (первый элемент массива). Наконец, переменная `s` разыменовывается, в результате чего во время выполнения возникает исключение.

См. также

- Ссылочные типы, допускающие значение NULL. Предложение
- Ссылочные типы, допускающие значение NULL. Черновик спецификации
- Аннотации неограниченных параметров типа
- Учебник "Введение в ссылки, допускающие значения NULL"
- **Nullable** (параметры компилятора C#)

Обновление базы кода путем добавления ссылочных типов, допускающих значение NULL, чтобы повысить качество диагностических предупреждений о значениях NULL

18.12.2021 • 6 minutes to read

Ссылочные типы, допускающие значения NULL, позволяют объявить, будет ли переменным ссылочного типа присваиваться значение `null`. Статический анализ и вывод предупреждений компилятора, когда код разыменовывает `null`, является самым важным преимуществом этой функции. После включения компилятор создает предупреждения, которые помогут предотвратить вывод исключения System.NullReferenceException во время выполнения кода.

Если база кода имеет относительно небольшой размер, можно включить функцию в проекте, устранить предупреждения и воспользоваться преимуществами улучшенной диагностики. Более крупные базы кода со временем могут потребовать более структурированного подхода к устранению предупреждений, чтобы в некоторых случаях эту функцию можно было использовать при устранении предупреждений в различных типах или файлах. В этой статье описываются различные стратегии обновления базы кода и их слабые места. Перед началом миграции ознакомьтесь с кратким обзором ссылочных типов, допускающих значения NULL. В нем рассматривается статический анализ компилятора, значения *с состоянием NULL* в аннотациях, допускающих значение NULL, со значениями *может быть NULL* и *не равно NULL*. После изучения этих понятий и терминов вы готовы к переносу кода.

Планирование миграции

Независимо от способа обновления базы кода, целью является включение в проект предупреждений и аннотаций, допускающих значения NULL. После достижения этой цели у вас в проекте будет параметр `<nullable>Enable</nullable>`. Для настройки параметров в других местах директивы pragma не требуются.

Первый вариант — задание значения по умолчанию для проекта. Доступны такие варианты:

1. ***Отключение параметра Nullable в качестве значения по умолчанию*** *_:_disable* является параметром по умолчанию, если в файл проекта не добавляется элемент `Nullable`. Используйте это значение по умолчанию, если вы не добавляете новые файлы в базу кода. Основным действием является обновление библиотеки для использования ссылочных типов, допускающих значение NULL. Выбор этого значения по умолчанию означает, что при обновлении кода в каждый файл добавляется директива pragma, допускающая значение NULL.
2. ***Разрешение значений NULL в качестве значений по умолчанию***: установите это значение по умолчанию, если вы активно разрабатываете новые функции. Требуется, чтобы новый код использовал ссылочные типы, допускающие значение NULL, и статический анализ, допускающий значение NULL. Использование этого значения по умолчанию означает, что необходимо добавить `#pragma nullable disable` в верхнюю часть каждого файла. Вы удалите эту директиву pragma, как только приступите к устранению предупреждений в этом файле.
3. ***Предупреждения, допускающие значения NULL, в качестве значения по умолчанию*** *_: выберите это значение по умолчанию для двухэтапной миграции. На первом этапе необходимо устранить предупреждения. На втором этапе включите аннотации для объявления ожидаемой переменной _null-state*. Использование этого значения по умолчанию означает, что необходимо добавить `#pragma nullable disable` в верхнюю часть каждого файла.

4. ***Аннотации, допускающие значение NULL***, в качестве значений по умолчанию. Добавьте аннотацию к коду перед тем, как устранить предупреждения.

Выбор параметра "Допускаются значения NULL" в качестве параметра по умолчанию добавляет работы на начальном этапе, поскольку нужно включить директиву pragma в каждый файл. Преимущество заключается в том, что для каждого нового файла кода, добавленного в проект, будет разрешено значение null. Любая новая работа будет поддерживать значения null; необходимо будет обновить только существующий код. Отключение этого параметра как параметра по умолчанию работает лучше, если библиотека является стабильной и основной задачей разработки является внедрение ссылочных типов, допускающих значение NULL. Ссылочные типы, допускающие значение null, включаются по мере аннотирования API. По завершении работы ссылочные типы, допускающие значение null, включаются для всего проекта. При создании нового файла необходимо добавить директивы pragma и настроить в нем разрешения для значений NULL. Если кто-то из разработчиков вашей команды забудет это сделать, новый код перейдет в разряд невыполненной работы по включению во всем коде поддержки значений null.

Выбор стратегии зависит от объема активной разработки в проекте. Чем более проект зрелый и стабильный, тем лучше работает вторая стратегия. Чем больше возможностей находится в разработке, тем лучше работает первая стратегия.

> **IMPORTANT**
> Глобальный контекст, допускающий значения NULL, не применяется для созданных файлов кода. В любом случае контекст, допускающий значение NULL, *отключен* для любого исходного файла, помеченного как созданный. Это означает, что все интерфейсы API в создаваемых файлах не заносятся в заметки. Существует четыре способа пометки файла как созданного:
>
> 1. В файле. editorconfig укажите `generated_code = true` в разделе, который применяется к этому файлу.
> 2. Вставьте `<auto-generated>` или `<auto-generated/>` в комментарий в верхней части файла. Он может находиться в любой строке комментария, однако блок комментариев должен быть первым элементом в файле.
> 3. Имя файла следует начинать с *TemporaryGeneratedFile_*
> 4. В конце имени файла следует указать *.designer.cs*, *.generated.cs*, *.g.cs* или *.g.i.cs*.
>
> Генераторы могут явно использовать директиву препроцессора `#nullable`.

Общие сведения о контекстах и предупреждениях

Включение предупреждений и аннотаций позволяет определять, как компилятор отображает ссылочные типы и допустимость значений NULL. Все типы поддерживают одно из трех состояний допустимости значений NULL.

- *oblivious*: все ссылочные типы имеют свойство *oblivious* и допускают значение NULL, если контекст аннотации отключен.
- *nonnullable*: ссылочный тип без аннотации `C` *не допускает значение NULL*, если включен контекст аннотации.
- *nullable*: ссылочный тип с аннотацией `C?` *допускает значение NULL*, но при отключении контекста аннотации может выводиться предупреждение. Переменные, объявленные с помощью `var`, *допускают значение NULL*, если включен контекст аннотации.

Компилятор создает предупреждения с учетом допустимости значений NULL.

- Типы, *не допускающие значения NULL*, вызывают предупреждения при присвоении им потенциального значения `null`.
- Типы, *допускающие значение NULL*, выдают предупреждения, если они выполняют разыменование,

когда *возможно значение NULL*.

- Типы со свойством *oblivious* вызывают предупреждения, если они выполняют разыменование, когда активно значение *может быть NULL* и включен контекст предупреждения.

Каждая переменная имеет состояние по умолчанию, допускающее значение NULL, которое зависит от допустимости значений NULL.

- Переменные с *состоянием NULL по умолчанию*, равным *может быть NULL*.
- Переменные с *состоянием NULL по умолчанию* со значением *может быть NULL*.
- Переменные, допускающие значение NULL, со свойством oblibious имеют *состояние NULL* по умолчанию со значением *не равно NULL*.

Перед включением ссылочных типов, допускающих значения NULL, все объявления в базе кода имеют значение *nullable oblivious*. Это важно, поскольку означает, что все ссылочные типы имеют *состояние NULL* по умолчанию, а не *не равно NULL*.

Устранение предупреждений

Если в проекте используется Entity Framework Core, следует ознакомиться с рекомендациями по работе со ссылочными типами, допускающими значение NULL.

При запуске миграции следует для начала включить только предупреждения. Все объявления по-прежнему имеют значение *nullable oblivious*, однако вы увидите предупреждения при разыменовании значения после изменения *состояния NULL* на *может быть NULL*. По мере устранения этих предупреждений в других расположениях будет выполняться проверка того, является ли значение значением NULL, и база кода станет более отказоустойчивой. Сведения о конкретных приемах, используемых в различных ситуациях, см. в статье Способы устранения предупреждений, допускающих значения NULL.

Перед тем как продолжить работу с другим кодом, можно устранить предупреждения и включить аннотации в каждом файле или классе. Однако зачастую более эффективным способом было бы устранить предупреждения, созданные во время использования контекста *предупреждений* перед тем, как включить аннотации типа. Таким образом, все типы будут иметь свойство *oblivious* до тех пор, пока не будет устранен первый набор предупреждений.

Включение аннотаций для типа

После устранения первого набора предупреждений можно включить *контекст аннотации*. При этом свойство *oblivious* ссылочных типов будет изменено на свойство *nonnullable*. Все переменные, объявленные с помощью `var`, *допускают значение NULL*. При этом изменении зачастую появляются новые предупреждения. Первым шагом в исправлении предупреждений компилятора является добавление заметок `?` к параметрам и возвращаемым типам, чтобы указать, когда аргументы или возвращаемые значения могут иметь значение `null`. При выполнении этой задачи ваша цель заключается не только в устранении предупреждений. Более важная задача заключается в том, чтобы компилятор понял смысл возможных значений null.

Атрибуты расширяют аннотации типов

Для выражения дополнительных сведений о состоянии NULL для переменных добавлено несколько атрибутов. Правила для API, скорее всего, будут более сложными, чем *не равно NULL*, или будут иметь свойство *может быть NULL* для всех параметров и возвращаемых значений. Для многих API действуют более сложные правила в отношении того, когда переменные могут или не могут иметь значение `null`. В таких случаях для выражения этих правил используются атрибуты. Атрибуты, описывающие семантику API, см. в статье, посвященной атрибутам, влияющим на анализ значений NULL.

Дальнейшие действия

После включения аннотаций и устранения всех предупреждений можно задать контекст по умолчанию *enabled* для проекта. Если в код были добавлены директивы pragma для контекста аннотации или предупреждения, допускающего значение NULL, их можно удалить. Со временем могут появиться новые предупреждения. Вы можете написать код, который представляет предупреждения. Можно обновить зависимость библиотеки для ссылочных типов, допускающих значение NULL. Эти обновления изменят типы в этой библиотеке со свойством *nullable oblivious* на *nonnullable* или *nullable*.

Методы в (C#)

18.12.2021 • 19 minutes to read

Метод — это блок кода, содержащий ряд инструкций. Программа инициирует выполнение инструкций, вызывая метод и указывая все аргументы, необходимые для этого метода. В C# все инструкции выполняются в контексте метода. Метод `Main` является точкой входа для каждого приложения C#, и вызывается общеязыковой средой выполнения (CLR) при запуске программы.

> **NOTE**
> В этом разделе рассматриваются названные методы. Дополнительные сведения об анонимных функциях см. в статье Лямбда-выражения.

Сигнатуры методов

Методы объявляются с помощью ограничений `class`, `record` или `struct`, для которых указываются следующие данные.

- Уровень доступа (необязательно), например `public` или `private`. Значение по умолчанию — `private`.
- Необязательные модификаторы, например `abstract` или `sealed`.
- Возвращаемое значение или `void`, если у метода его нет.
- Имя метода.
- Любые параметры методов. Параметры метода заключаются в скобки и разделяются запятыми. Пустые скобки указывают, что параметры методу не требуются.

Вместе все эти части формируют сигнатуру метода.

> **IMPORTANT**
> Тип возврата метода не является частью сигнатуры метода в целях перегрузки метода. Однако он является частью сигнатуры метода при определении совместимости между делегатом и методом, который он указывает.

В следующем примере определяется класс с именем `Motorcycle`, содержащий пять методов:

```
using System;

abstract class Motorcycle
{
    // Anyone can call this.
    public void StartEngine() {/* Method statements here */ }

    // Only derived classes can call this.
    protected void AddGas(int gallons) { /* Method statements here */ }

    // Derived classes can override the base class implementation.
    public virtual int Drive(int miles, int speed) { /* Method statements here */ return 1; }

    // Derived classes can override the base class implementation.
    public virtual int Drive(TimeSpan time, int speed) { /* Method statements here */ return 0; }

    // Derived classes must implement this.
    public abstract double GetTopSpeed();
}
```

Обратите внимание на то, что класс `Motorcycle` включает перегруженный метод `Drive`. Оба метода называются одинаково, но различаются по типам параметров.

Вызов метода

Можно использовать метод *instance* или *static*. Для того чтобы вызвать метод instance, необходимо создать экземпляр объекта и вызвать для него метод; метод instance будет применен к этому экземпляру и его данным. Статический метод вызывается путем ссылки на имя типа, к которому относится метод; статические методы не работают с данными экземпляров. При попытке вызвать статический метод с помощью экземпляра объекта возникает ошибка компилятора.

Вызов метода аналогичен доступу к полю. После имени объекта (при вызове метода экземпляра) или имени типа (при вызове метода `static`) добавьте точку, имя метода и круглые скобки. Аргументы перечисляются в этих скобках и разделяются запятыми.

Определение метода задает имена и типы всех необходимых параметров. Когда вызывающий код вызывает метод, он предоставляет конкретные значения, называемые аргументами, для каждого параметра. Аргументы должны быть совместимы с типом параметра, но имя аргумента (если оно используется в вызывающем коде) может не совпадать с именем параметра, указанным в методе. В следующем примере метод `Square` имеет один параметр типа `int` с именем *i*. Первый вызов метода передает методу `Square` переменную типа `int` с именем *num*; второй — числовую константу, а третий — выражение.

```
public class SquareExample
{
   public static void Main()
   {
      // Call with an int variable.
      int num = 4;
      int productA = Square(num);

      // Call with an integer literal.
      int productB = Square(12);

      // Call with an expression that evaluates to int.
      int productC = Square(productA * 3);
   }

   static int Square(int i)
   {
      // Store input argument in a local variable.
      int input = i;
      return input * input;
   }
}
```

В наиболее распространенной форме вызова методов используются позиционные аргументы; они передаются в том же порядке, что и параметры метода. Таким образом, методы класса `Motorcycle` могут вызываться, как показано в следующем примере. Например, вызов метода `Drive` включает два аргумента, которые соответствуют двум параметрам в синтаксисе метода. Первый становится значением параметра `miles`, а второй — значением параметра `speed`.

```
class TestMotorcycle : Motorcycle
{
    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {

        TestMotorcycle moto = new TestMotorcycle();

        moto.StartEngine();
        moto.AddGas(15);
        moto.Drive(5, 20);
        double speed = moto.GetTopSpeed();
        Console.WriteLine("My top speed is {0}", speed);
    }
}
```

При вызове метода вместо позиционных аргументов можно также использовать *именованные аргументы*. При использовании именованных аргументов необходимо указать имя параметра, двоеточие (":"), а затем аргумент. Аргументы для метода могут отображаться в любом порядке, при условии, что все обязательные аргументы присутствуют. В следующем примере для вызова метода `TestMotorcycle.Drive` используются именованные аргументы. В этом примере именованные аргументы передаются из списка параметров метода в обратном порядке.

```
using System;

class TestMotorcycle : Motorcycle
{
    public override int Drive(int miles, int speed)
    {
        return (int)Math.Round(((double)miles) / speed, 0);
    }

    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {

        TestMotorcycle moto = new TestMotorcycle();
        moto.StartEngine();
        moto.AddGas(15);
        var travelTime = moto.Drive(speed: 60, miles: 170);
        Console.WriteLine("Travel time: approx. {0} hours", travelTime);
    }
}
// The example displays the following output:
//      Travel time: approx. 3 hours
```

Метод можно вызывать, используя и позиционные, и именованные аргументы. Однако позиционные аргументы могут следовать за именованными аргументами, только если именованные аргументы находятся в правильных позициях. В следующем примере метод `TestMotorcycle.Drive` из предыдущего примера вызывается с использованием одного позиционного и одного именованного аргумента.

```
var travelTime = moto.Drive(170, speed: 55);
```

Унаследованные и переопределенные методы

Помимо членов, определенных в нем явно, тип наследует члены, определенные в его базовых классах. Так как все типы в системе управляемых типов напрямую или косвенно наследуются из класса Object, все типы наследуют его члены, такие как Equals(Object), GetType() и ToString(). В следующем примере определяется класс `Person`, который создает экземпляры двух объектов `Person` и вызывает метод `Person.Equals`, чтобы определить, равны ли эти объекты. При этом метод `Equals` в классе `Person` не определяется, он наследуется из Object.

```
using System;

public class Person
{
   public String FirstName;
}

public class ClassTypeExample
{
   public static void Main()
   {
      var p1 = new Person();
      p1.FirstName = "John";
      var p2 = new Person();
      p2.FirstName = "John";
      Console.WriteLine("p1 = p2: {0}", p1.Equals(p2));
   }
}
// The example displays the following output:
//      p1 = p2: False
```

Типы могут переопределять унаследованные члены, используя ключевое слово `override` и обеспечивая реализацию переопределенного метода. Сигнатура метода должна быть такой же, как у переопределенного метода. Следующий пример аналогичен предыдущему за тем исключением, что переопределяет метод Equals(Object). (Он также переопределяет метод GetHashCode(), поскольку оба эти метода предназначены для получения согласованных результатов.)

```
using System;

public class Person
{
    public String FirstName;

    public override bool Equals(object obj)
    {
        var p2 = obj as Person;
        if (p2 == null)
            return false;
        else
            return FirstName.Equals(p2.FirstName);
    }

    public override int GetHashCode()
    {
        return FirstName.GetHashCode();
    }
}

public class Example
{
    public static void Main()
    {
        var p1 = new Person();
        p1.FirstName = "John";
        var p2 = new Person();
        p2.FirstName = "John";
        Console.WriteLine("p1 = p2: {0}", p1.Equals(p2));
    }
}
// The example displays the following output:
//      p1 = p2: True
```

Передача параметров

Типы в C# делятся на *типы значений* и *ссылочные типы*. Список встроенных типов значений см. в разделе Типы. По умолчанию и типы значений, и ссылочные типы передаются в метод по значению.

Передача параметров по значению

При передаче типа значения в метод по значению вместо самого объекта передается его копия. Это значит, что изменения объекта в вызываемом методе не отражаются на исходном объекте, когда управление возвращается вызывающему объекту.

Код в следующем примере передает тип значения в метод по значению, а вызываемый метод пытается изменить значение типа значения. Он определяет переменную типа `int`, который является типом значения, присваивает ему значение 20 и передает его в метод с именем `ModifyValue`, который изменяет значение переменной на 30. Однако, когда метод возвращается, значение переменной остается неизменным.

```
using System;

public class ByValueExample
{
   public static void Main()
   {
      int value = 20;
      Console.WriteLine("In Main, value = {0}", value);
      ModifyValue(value);
      Console.WriteLine("Back in Main, value = {0}", value);
   }

   static void ModifyValue(int i)
   {
      i = 30;
      Console.WriteLine("In ModifyValue, parameter value = {0}", i);
      return;
   }
}
// The example displays the following output:
//      In Main, value = 20
//      In ModifyValue, parameter value = 30
//      Back in Main, value = 20
```

Если объект ссылочного типа передается в метод по значению, ссылка на этот объект передается по значению. Это значит, что метод получает не сам объект, а аргумент, который указывает расположение объекта. Если с помощью этой ссылки в член объекта вносится изменение, это изменение отражается в объекте, даже если управление возвращается вызывающему объекту. При этом изменения в объекте, переданном в метод, не отражаются на исходном объекте, когда управление возвращается вызывающему объекту.

В следующем примере определяется класс (ссылочного типа) с именем `SampleRefType`. Он создает экземпляр объекта `SampleRefType`, задает в его поле `value` значение 44 и передает объект в метод `ModifyObject`. В этом примере, в сущности, происходит то же самое, что и в предыдущем, — аргумент передается в метод по значению. Однако поскольку здесь используется ссылочный тип, результат будет другим. В данном случае в методе `ModifyObject` изменено поле `obj.value`, при этом поле `value` аргумента `rt` в методе `Main` также изменяется на 33, как видно из результатов в предыдущем примере.

```
using System;

public class SampleRefType
{
    public int value;
}

public class ByRefTypeExample
{
    public static void Main()
    {
        var rt = new SampleRefType();
        rt.value = 44;
        ModifyObject(rt);
        Console.WriteLine(rt.value);
    }

    static void ModifyObject(SampleRefType obj)
    {
        obj.value = 33;
    }
}
```

Передача параметров по ссылке

Параметр передается по ссылке, когда нужно изменить значение аргумента в методе и сохранить это изменение после того, как управление вернется вызывающему методу. Для передачи параметра по ссылке используйте ключевое слово `ref` или `out`. Можно также передать значение по ссылке, чтобы предотвратить копирование, и при этом запретить внесение изменений с помощью ключевого слова `in`.

Следующий пример идентичен предыдущему за тем исключением, что значение передается в метод `ModifyValue` по ссылке. Если значение параметра в методе `ModifyValue` будет изменено, при возвращении управления вызывающему объекту это изменение не сохранится.

```
using System;

public class ByRefExample
{
   public static void Main()
   {
      int value = 20;
      Console.WriteLine("In Main, value = {0}", value);
      ModifyValue(ref value);
      Console.WriteLine("Back in Main, value = {0}", value);
   }

   static void ModifyValue(ref int i)
   {
      i = 30;
      Console.WriteLine("In ModifyValue, parameter value = {0}", i);
      return;
   }
}
// The example displays the following output:
//      In Main, value = 20
//      In ModifyValue, parameter value = 30
//      Back in Main, value = 30
```

Общий шаблон, в котором используются параметры по ссылке, включает замену значений переменных. Когда две переменные передаются в метод по ссылке, он меняет их содержимое местами. В следующем примере меняются местами целочисленные значения.

```
using System;

public class RefSwapExample
{
   static void Main()
   {
      int i = 2, j = 3;
      System.Console.WriteLine("i = {0}  j = {1}" , i, j);

      Swap(ref i, ref j);

      System.Console.WriteLine("i = {0}  j = {1}" , i, j);
   }

   static void Swap(ref int x, ref int y)
   {
      int temp = x;
      x = y;
      y = temp;
   }
}
// The example displays the following output:
//      i = 2  j = 3
//      i = 3  j = 2
```

Передача параметров ссылочного типа позволяет изменить значение самой ссылки, а не отдельных ее элементов или полей.

Массивы параметров

В некоторых случаях требование об указании точного числа аргументов для метода является строгим. Если параметр в массиве параметров указывается с помощью ключевого слова `params`, метод можно вызывать с переменным числом аргументов. Параметр, помеченный ключевым словом `params`, должен быть типом массива и занимать последнюю позицию в списке параметров метода.

После этого вызывающий объект можно вызвать одним из четырех способов:

- передавая массив соответствующего типа, содержащий требуемое число элементов;
- передавая в метод список отдельных аргументов соответствующего типа, разделенный запятыми;
- Путем передачи `null`.
- не передавая никакие аргументы в массив параметров.

В следующем примере определяется метод с именем `GetVowels`, возвращающий все гласные из массива параметров. Метод `Main` демонстрирует все четыре способа вызова метода. От вызывающих объектов не требуются аргументы для параметров, которые включают модификатор `params`. В этом случае параметр является пустым массивом.

```
using System;
using System.Linq;

class ParamsExample
{
    static void Main()
    {
        string fromArray = GetVowels(new[] { "apple", "banana", "pear" });
        Console.WriteLine($"Vowels from array: '{fromArray}'");

        string fromMultipleArguments = GetVowels("apple", "banana", "pear");
        Console.WriteLine($"Vowels from multiple arguments: '{fromMultipleArguments}'");

        string fromNull = GetVowels(null);
        Console.WriteLine($"Vowels from null: '{fromNull}'");

        string fromNoValue = GetVowels();
        Console.WriteLine($"Vowels from no value: '{fromNoValue}'");
    }

    static string GetVowels(params string[] input)
    {
        if (input == null || input.Length == 0)
        {
            return string.Empty;
        }

        var vowels = new char[] { 'A', 'E', 'I', 'O', 'U' };
        return string.Concat(
            input.SelectMany(
                word => word.Where(letter => vowels.Contains(char.ToUpper(letter)))));
    }
}

// The example displays the following output:
//     Vowels from array: 'aeaaaea'
//     Vowels from multiple arguments: 'aeaaaea'
//     Vowels from null: ''
//     Vowels from no value: ''
```

Необязательные параметры и аргументы

В определении метода может быть указано, являются его параметры обязательными или нет. По умолчанию параметры обязательны. Для определения необязательных параметров значения параметра по умолчанию включаются в определение метода. Если при вызове метода никакие аргументы для необязательного параметры не указываются, вместо них используется значение по умолчанию.

Значение параметра по умолчанию должно быть назначено одним из следующих видов выражений:

- Константа, например, строковый литерал или число.
- Выражение, имеющее вид `default(SomeType)`, где `SomeType` может быть либо типом значения, либо ссылочным типом. Использование ссылочного типа практически эквивалентно указанию `null`. Начиная с C# 7.1 вы можете использовать литерал `default`, поскольку компилятор может выводить тип из объявления параметра.
- Выражение в форме `new ValType()`, где `ValType` — это тип значения. Обратите внимание на то, что при этом вызывается не имеющий параметров неявный конструктор типа значения, который не является фактическим членом типа.

> **NOTE**
> В C# 10 и более поздних версиях, когда выражение формы `new ValType()` вызывает явно определенный конструктор без параметров типа значения, компилятор выдает ошибку, так как значение параметра по умолчанию должно быть константой времени компиляции. Используйте выражение `default(ValType)` или литерал `default` для предоставления значения параметра по умолчанию. Дополнительные сведения о конструкторах без параметров см. в разделе Конструкторы без параметров и инициализаторы полей статьи Типы структур.

Если метод содержит как обязательные, так и необязательные параметры, необязательные параметры определяются в конце списка параметров после всех обязательных параметров.

В следующем примере определяется метод `ExampleMethod`, который имеет один обязательный и два необязательных параметра.

```
using System;

public class Options
{
    public void ExampleMethod(int required, int optionalInt = default,
                              string? description = default)
   {
        var msg = $"{description ?? "N/A"}: {required} + {optionalInt} = {required + optionalInt}";
        Console.WriteLine(msg);
   }
}
```

Если для вызова метода с несколькими необязательными аргументами используются позиционные аргументы, вызывающий объект должен предоставить аргумент для всех необязательных параметров, для которых предоставлен аргумент, от первого до последнего. Например, если при использовании метода `ExampleMethod` вызывающий объект предоставляет аргумент для параметра `description`, он должен также предоставить его для параметра `optionalInt`.
`opt.ExampleMethod(2, 2, "Addition of 2 and 2");` — допустимый вызов метода;
`opt.ExampleMethod(2, , "Addition of 2 and 0");` вызывает ошибку компилятора "Аргумент отсутствует".

Если метод вызывается с помощью именованных аргументов или комбинации позиционных и именованных аргументов, вызывающий объект может опустить любые аргументы, следующие за последним позиционным аргументом в вызове метода.

В следующем примере метод `ExampleMethod` вызывается трижды. В первых двух вызовах метода используются позиционные аргументы. В первом пропускаются оба необязательных аргумента, а во втором — последний. Третий вызов метода предоставляет позиционный аргумент для обязательного параметра, но использует именованный аргумент для передачи значения в параметр `description`, в то время как аргумент `optionalInt` опускается.

```
public class OptionsExample
{
   public static void Main()
   {
      var opt = new Options();
      opt.ExampleMethod(10);
      opt.ExampleMethod(10, 2);
      opt.ExampleMethod(12, description: "Addition with zero:");
   }
}
// The example displays the following output:
//      N/A: 10 + 0 = 10
//      N/A: 10 + 2 = 12
//      Addition with zero:: 12 + 0 = 12
```

Использование необязательных параметров влияет на *разрешение перегрузки* или на способ, с помощью которого компилятор C# определяет, какая именно перегрузка должна вызываться при вызове метода, следующим образом:

- Метод, индексатор или конструктор является кандидатом на выполнение, если каждый из его параметров необязателен либо по имени или позиции соответствует одному и тому же аргументу в операторе вызова, и этот аргумент можно преобразовать в тип параметра.
- Если найдено более одного кандидата, правила разрешения перегрузки для предпочтительных преобразований применяются к аргументам, указанным явно. Опущенные аргументы для необязательных параметров игнорируются.
- Если два кандидата определяются как равно подходящие, предпочтение отдается кандидату без необязательных параметров, аргументы которых в вызове были опущены. Это — последовательность определения приоритетов в разрешении перегрузки для кандидатов с меньшим числом параметров.

Возвращаемые значения

Методы могут возвращать значение вызывающему объекту. Если тип возврата, указываемый перед именем метода, не `void`, этот метод может возвращать значение с помощью ключевого слова `return`. Инструкция с ключевым словом `return`, за которым следует переменная, константа или выражение, соответствующие типу возврата, будут возвращать это значение объекту, вызвавшему метод. Методы с типом возврата, отличным от void, должны использовать ключевое слово `return` для возврата значения. Ключевое слове `return` также останавливает выполнение метода.

Если тип возврата — `void`, инструкцию `return` без значения по-прежнему можно использовать для завершения выполнения метода. Без ключевого слова `return` этот метод будет останавливать выполнение при достижении конца блока кода.

Например, в следующих двух методах ключевое слово `return` используется для возврата целочисленных значений.

```
class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2)
    {
        return number1 + number2;
    }

    public int SquareANumber(int number)
    {
        return number * number;
    }
}
```

Чтобы использовать значение, возвращаемое из метода, вызывающий метод может применять сам вызов метода везде, где будет достаточно значения того же типа. Можно также назначить возвращаемое значение переменной. Например, следующие два примера кода достигают одной и той же цели.

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

```
result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));
// The result is 9.
Console.WriteLine(result);
```

Использование локальной переменной, в данном случае `result`, для сохранения значения является необязательным. Это может улучшить читаемость кода или может оказаться необходимым, если нужно сохранить исходное значение аргумента для всей области метода.

В некоторых случаях нужно, чтобы метод возвращал больше одного значения. Начиная с C# версии 7.0, это легко можно сделать с помощью *типов кортежей* и *литералов кортежей*. Тип кортежа определяет типы данных для элементов кортежа. Литералы кортежей предоставляют фактические значения возвращаемого кортежа. В следующем примере `(string, string, string, int)` определяет тип кортежа, возвращаемый методом `GetPersonalInfo`. Выражение `(per.FirstName, per.MiddleName, per.LastName, per.Age)` представляет собой литерал кортежа; метод возвращает имя, отчество и фамилию, а также возраст объекта `PersonInfo`.

```
public (string, string, string, int) GetPersonalInfo(string id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

После этого вызывающий объект может использовать возвращенный кортеж в коде следующего вида:

```
var person = GetPersonalInfo("111111111")
Console.WriteLine($"{person.Item1} {person.Item3}: age = {person.Item4}");
```

Имена могут также назначаться элементам кортежа в определении типа кортежа. В следующих примерах демонстрируется альтернативная версия метода `GetPersonalInfo`, в котором используются именованные элементы:

```
public (string FName, string MName, string LName, int Age) GetPersonalInfo(string id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

После этого предыдущий вызов метода `GetPersonInfo` можно изменить следующим образом:

```
var person = GetPersonalInfo("111111111");
Console.WriteLine($"{person.FName} {person.LName}: age = {person.Age}");
```

Если в качестве аргумента метод получает массив, а затем изменяет значение отдельных элементов, он может не возвращать массив, однако при желании вы можете это изменить для соблюдения правильного стиля или обеспечения эффективного потока передачи значений. Это связано с тем, что C# передает все ссылочные типы по значению, а значением ссылки на массив является указатель на массив. В следующем примере изменения в содержимом массива `values`, сделанные в методе `DoubleValues`, может отслеживать любой код, имеющий ссылку на этот массив.

```
using System;

public class ArrayValueExample
{
   static void Main(string[] args)
   {
      int[] values = { 2, 4, 6, 8 };
      DoubleValues(values);
      foreach (var value in values)
         Console.Write("{0}  ", value);
   }

   public static void DoubleValues(int[] arr)
   {
      for (int ctr = 0; ctr <= arr.GetUpperBound(0); ctr++)
         arr[ctr] = arr[ctr] * 2;
   }
}
// The example displays the following output:
//       4  8  12  16
```

Методы расширения

Как правило, добавлять методы в существующий тип можно двумя способами:

- Изменение исходного кода для этого типа. Конечно, если вы не владеете исходным кодом этого типа, сделать это невозможно. Если при этом в поддержку метода также добавляются поля закрытых данных, это изменение становится критическим.
- Определение нового метода в производном классе. Нельзя добавить метод этим способом, используя наследование для других типов, таких как структуры и перечисления. Кроме того, оно не позволяет "добавить" метод в запечатанный класс.

Методы расширения позволяют "добавить" метод в существующий тип, не меняя сам тип и не реализуя новый метод в наследуемом типе. Кроме того, метод расширения может не входить в ту же сборку, в которую входит расширяемый им тип. Вызовите метод расширения, как будто он является определенным членом типа.

Дополнительные сведения см. в статье Методы расширения.

Асинхронные методы

С помощью функции async можно вызывать асинхронные методы, не прибегая к использованию явных обратных вызовов или ручному разделению кода между несколькими методами или лямбда-выражениями.

Если пометить метод с помощью модификатора async , можно использовать в этом методе инструкцию await . Если ожидаемая задача не завершена, то достигнув выражения `await` в асинхронном методе, управление возвращается вызывающему объекту, а выполнение метода с ключевым словом `await` приостанавливается до завершения выполнения ожидаемой задачи. После завершения задачи можно возобновить выполнение в методе.

> **NOTE**
> Асинхронный метод возвращается в вызывающий объект, когда он встречает первый ожидаемый объект, выполнение которого еще не завершено, или когда выполнение асинхронного метода доходит до конца — в зависимости от того, что происходит раньше.

Асинхронный метод обычно имеет тип возвращаемого значения Task<TResult>, Task, IAsyncEnumerable<T> или `void`. Тип возвращаемого значения `void` в основном используется для определения обработчиков событий, где требуется возвращать тип `void`. Асинхронный метод, который возвращает тип `void`, не может быть ожидающим. Вызывающий объект метода, возвращающего значение типа void, не может перехватывать исключения, которые выдает этот метод. Начиная с C# 7.0 асинхронный метод может возвращать любой тип вида задачи.

В следующем примере `DelayAsync` представляет собой асинхронный метод с оператором return, который возвращает целое число. Поскольку этот метод асинхронный, его объявление должно иметь тип возвращаемого значения `Task<int>`. Поскольку тип возврата — `Task<int>`, вычисление выражения `await` в `DoSomethingAsync` создает целое число, как показывает следующий оператор: `int result = await delayTask`.

```
using System;
using System.Threading.Tasks;

class Program
{
    static Task Main() => DoSomethingAsync();

    static async Task DoSomethingAsync()
    {
        Task<int> delayTask = DelayAsync();
        int result = await delayTask;

        // The previous two statements may be combined into
        // the following statement.
        //int result = await DelayAsync();

        Console.WriteLine($"Result: {result}");
    }

    static async Task<int> DelayAsync()
    {
        await Task.Delay(100);
        return 5;
    }
}
// Example output:
//   Result: 5
```

Асинхронный метод не может объявлять параметры in, ref или out, но может вызывать методы, имеющие такие параметры.

Дополнительные сведения об асинхронных методах см. в разделах Асинхронное программирование с использованием ключевых слов async и await (C#) и Типы возвращаемых значений асинхронных операций.

Элементы, воплощающие выражение

Часто используются определения методов, которые просто немедленно возвращаются с результатом выражения или которые имеют единственную инструкцию в тексте метода. Для определения таких методов существует сокращенный синтаксис с использованием `=>`:

```
public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public void Print() => Console.WriteLine(First + " " + Last);
// Works with operators, properties, and indexers too.
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);
```

Если метод возвращает `void` или является асинхронным, текст этого метода должен быть выражением оператора (как и при использовании лямбда-выражений). Свойства и индексаторы должны быть доступны только для чтения, и использовать ключевое слово метода доступа `get` не следует.

Iterators

Итератор выполняет настраиваемую итерацию по коллекции, например по списку или массиву. Итератор использует инструкцию yield return для возврата всех элементов по одному. По достижении оператора `yield return` текущее расположение запоминается, чтобы вызывающий объект мог запросить следующий элемент в последовательности.

Тип возврата итератора может быть IEnumerable, IEnumerable<T>, IEnumerator или IEnumerator<T>.

Дополнительные сведения см. в разделе Итераторы.

См. также раздел

- Модификаторы доступа
- Статические классы и члены статических классов
- Наследование
- Абстрактные и запечатанные классы и члены классов
- params
- out
- ref
- in
- Передача параметров

Свойства

18.12.2021 • 8 minutes to read

Свойства являются привилегированными компонентами C#. Язык определяет синтаксис, который позволяет разработчикам писать код, отражающий цели их проекта.

Свойства при обращении к ним ведут себя как поля. Однако в отличие от полей свойства реализуются с помощью методов доступа, которые определяют инструкции, выполняемые при обращении к свойству или при его назначении.

Синтаксис свойства

Синтаксис свойств является естественным расширением полей. Поле определяет место хранения:

```
public class Person
{
    public string FirstName;
    // remaining implementation removed from listing
}
```

Определение свойства содержит объявления для методов доступа `get` и `set`, которые получают и устанавливают значение этого свойства:

```
public class Person
{
    public string FirstName { get; set; }

    // remaining implementation removed from listing
}
```

Синтаксис, показанный выше, является синтаксисом *автосвойств*. Компилятор создает место хранения для поля, поддерживающего свойство. Компилятор также реализует тело методов доступа `get` и `set`.

Бывает, что свойство необходимо инициализировать со значением, отличным от значения по умолчанию для его типа. C# позволяет это сделать, указав значение после закрывающей фигурной скобки свойства. В этом случае в качестве начального значения для свойства `FirstName` можно задать пустую строку, а не `null`. Для этого используется следующий код:

```
public class Person
{
    public string FirstName { get; set; } = string.Empty;

    // remaining implementation removed from listing
}
```

Как вы увидите далее в этой статье, конкретная инициализация особенно полезна для свойств, предназначенных только для чтения.

Вы можете определить хранилище самостоятельно, как показано ниже:

```
public class Person
{
    public string FirstName
    {
        get { return firstName; }
        set { firstName = value; }
    }
    private string firstName;
    // remaining implementation removed from listing
}
```

Если реализация свойства представляет собой одиночное выражение, в качестве метода получения или задания можно использовать *элементы, воплощающие выражение*.

```
public class Person
{
    public string FirstName
    {
        get => firstName;
        set => firstName = value;
    }
    private string firstName;
    // remaining implementation removed from listing
}
```

Такой упрощенный синтаксис будет применяться в этой статье везде, где это возможно.

В примере выше определяется свойство для чтения и записи. Обратите внимание на ключевое слово `value` в методе доступа set. Метод доступа `set` всегда имеет один параметр с именем `value`. Метод доступа `get` должен возвращать значение, которое можно преобразовать в свойство (`string` в этом примере).

Это основные сведения о синтаксисе. Существует множество различных вариантов, поддерживающих разные идиомы. Рассмотрим их, а также соответствующие параметры синтаксиса.

Сценарии

Приведенные выше примеры демонстрируют один из простейших вариантов определения свойств: свойство для чтения и записи без проверки. Путем написания нужного кода в методах доступа `get` и `set` можно реализовать много разных сценариев.

Проверка

Можно написать код в методе доступа `set`, чтобы гарантировать, что значения, представленные свойством, всегда будут допустимыми. Например, предположим, что одно из правил для класса `Person` заключается в том, что имя не может быть пустым или содержать только пробелы. Это можно реализовать следующим образом:

```
public class Person
{
    public string FirstName
    {
        get => firstName;
        set
        {
            if (string.IsNullOrWhiteSpace(value))
                throw new ArgumentException("First name must not be blank");
            firstName = value;
        }
    }
    private string firstName;
    // remaining implementation removed from listing
}
```

Предыдущий пример можно упростить, воспользовавшись выражением `throw` в рамках проверки метода задания свойства:

```
public class Person
{
    public string FirstName
    {
        get => firstName;
        set => firstName = (!string.IsNullOrWhiteSpace(value)) ? value : throw new ArgumentException("First
name must not be blank");
    }
    private string firstName;
    // remaining implementation removed from listing
}
```

В приведенном выше примере код применяет правило о том, что имя не может быть пустым или содержать только пробелы. Если разработчик пишет

```
hero.FirstName = "";
```

Это назначение создает исключение `ArgumentException`. Поскольку метод доступа set свойства должен иметь тип возвращаемого значения void, чтобы сообщить об ошибках в методе доступа set, создается исключение.

Этот синтаксис можно расширить для любых компонентов в вашем сценарии. Можно проверить отношения между разными свойствами или соответствие любым внешним условиям. Любые допустимые операторы C# являются допустимыми в методе доступа свойства.

Только для чтения

До этого момента все примеры определения свойств определяли свойства для чтения и записи с помощью открытых методов доступа. Это не единственные операции доступа для свойств. Можно создать свойства, доступные только для чтения, или назначить другой уровень доступа для методов set и get. Предположим, ваш класс `Person` должен допускать изменение значения свойства `FirstName` только из других методов этого класса. Вы можете предоставить методу доступа set уровень доступа `private`, а не `public`:

```
public class Person
{
    public string FirstName { get; private set; }

    // remaining implementation removed from listing
}
```

Теперь к свойству `FirstName` можно получать доступ из любого кода, но назначить его можно только из другого кода в классе `Person`.

Вы можете добавить любой ограничивающий модификатор доступа для методов доступа set или get. Модификатор доступа, установленный для отдельного метода доступа, должен задавать более строгие ограничения, чем модификатор доступа для определения свойства. Приведенный выше пример допустим, так как свойство `FirstName` является открытым (`public`) а метод доступа set — закрытым (`private`). Нельзя объявить свойство `private` с методом доступа `public`. Свойство также можно объявить как `protected`, `internal`, `protected internal` или даже `private`.

Также допускается размещение более строгих модификаторов для метода доступа `get`. Например, свойство `public` может быть открытым, а метод доступа `get` ограничен типом `private`. Этот сценарий редко реализуется на практике.

Кроме того, можно ограничить изменения в свойстве, разрешив задавать его только в конструкторе или инициализаторе свойств. Внести соответствующие изменения в класс `Person` можно следующим образом:

```
public class Person
{
    public Person(string firstName) => this.FirstName = firstName;

    public string FirstName { get; }

    // remaining implementation removed from listing
}
```

Эта функция чаще всего используется для инициализации коллекций, которые представляются как свойства, доступные только для чтения:

```
public class Measurements
{
    public ICollection<DataPoint> points { get; } = new List<DataPoint>();
}
```

Вычисляемые свойства

Свойство не обязательно должно просто возвращать значение поля члена. Можно создать свойства, возвращающие вычисляемое значение. Расширим объект `Person` так, чтобы он возвращал полное имя, вычисляемое путем объединения имени и фамилии:

```
public class Person
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    public string FullName { get { return $"{FirstName} {LastName}"; } }
}
```

В примере выше используется функция интерполяции строк для создания форматированной строки для полного имени.

Можно также использовать *члены, воплощающие выражения*, которые обеспечивают более краткий способ создания вычисляемого свойства `FullName`:

```
public class Person
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    public string FullName => $"{FirstName} {LastName}";
}
```

Члены, воплощающие выражения, используют синтаксис *лямбда-выражений* для определения метода, который содержит одно выражение. Здесь это выражение возвращает полное имя объекта person.

Свойства с вычислением в кэше

Вы можете сочетать концепцию вычисляемого свойства с хранением и созданием *свойства с вычислением в кэше*. Например, можно изменить свойство `FullName` таким образом, чтобы форматирование строки выполнялось только при первом доступе к нему.

```
public class Person
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    private string fullName;
    public string FullName
    {
        get
        {
            if (fullName == null)
                fullName = $"{FirstName} {LastName}";
            return fullName;
        }
    }
}
```

Обратите внимание, что приведенный выше код содержит ошибку. Если код изменяет значение свойств `FirstName` или `LastName`, ранее вычисленное поле `fullName` является недопустимым. Вам потребуется изменить методы доступа `set` свойств `FirstName` и `LastName`, чтобы вычислить поле `fullName` еще раз:

```
public class Person
{
    private string firstName;
    public string FirstName
    {
        get => firstName;
        set
        {
            firstName = value;
            fullName = null;
        }
    }

    private string lastName;
    public string LastName
    {
        get => lastName;
        set
        {
            lastName = value;
            fullName = null;
        }
    }

    private string fullName;
    public string FullName
    {
        get
        {
            if (fullName == null)
                fullName = $"{FirstName} {LastName}";
            return fullName;
        }
    }
}
```

Эта окончательная версия вычисляет свойство `FullName` только при необходимости. Если ранее вычисленная версия является допустимой, используется она. Если другое изменение состояния делает ранее вычисленную версию недействительной, она будет пересчитана. Разработчикам, использующим этот класс, необязательно знать детали реализации. Ни одно из этих внутренних изменений не влияет на использование объекта person. Это главная причина для использования свойств для предоставления доступа к членам данных объекта.

Присоединение атрибутов к автоматически реализуемым свойствам

Начиная с C# 7.3 атрибуты полей можно прикреплять к созданному компилятором резервному полю в автоматически реализуемых свойствах. Например, рассмотрим изменение класса `Person`, который добавляет уникальное целочисленное свойство `Id`. Можно записать свойство `Id` с помощью автоматически реализуемого свойства, но в вашем коде не предусмотрено сохранение свойства `Id`. NonSerializedAttribute можно прикреплять только к полям, а не свойствам. Можно прикрепить NonSerializedAttribute к резервному полю для свойства `Id` с помощью описателя `field:` в атрибуте, как показано в следующем примере:

```
public class Person
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    [field:NonSerialized]
    public int Id { get; set; }

    public string FullName => $"{FirstName} {LastName}";
}
```

Этот способ подходит для любого атрибута, который вы прикрепляете к резервному полю в автоматически реализуемом свойстве.

Реализация INotifyPropertyChanged

Последним сценарием, где необходимо написать код в методе доступа к свойству, является поддержка интерфейса INotifyPropertyChanged, используемого для уведомления клиентов привязки данных об изменении значения. При изменении значения свойства объект вызывает событие INotifyPropertyChanged.PropertyChanged, указывающее на изменение. Библиотеки привязки данных, в свою очередь, изменяют отображаемые элементы на основе этого изменения. В следующем примере кода показано, как можно реализовать свойства `INotifyPropertyChanged` для `FirstName` этого класса person.

```
public class Person : INotifyPropertyChanged
{
    public string FirstName
    {
        get => firstName;
        set
        {
            if (string.IsNullOrWhiteSpace(value))
                throw new ArgumentException("First name must not be blank");
            if (value != firstName)
            {
                firstName = value;
                PropertyChanged?.Invoke(this,
                    new PropertyChangedEventArgs(nameof(FirstName)));
            }
        }
    }
    private string firstName;

    public event PropertyChangedEventHandler PropertyChanged;
    // remaining implementation removed from listing
}
```

Оператор `?.` называется оператором *объединения со значением NULL*. Он проверяет наличие пустой ссылки перед вычислением правой стороны оператора. В конечном итоге, если нет подписчиков на событие `PropertyChanged`, код для вызова события не выполняется. В этом случае без такой проверки будет создано исключение `NullReferenceException`. Дополнительные сведения см. на веб-сайте `events`. В этом примере также используется новый оператор `nameof` для преобразования символа имени свойства в текстовое представление. С помощью `nameof` можно сократить количество ошибок, возникающих при неправильном вводе имени свойства.

Реализация INotifyPropertyChanged — это пример случая, когда можно писать код в методах доступа для поддержки необходимых сценариев.

Подведем итоги

Свойства — это своего рода интеллектуальные поля в классе или объекте. Из-за пределов объекта они представляются полями в объекте. Однако для реализации свойства можно использовать полную палитру функциональных возможностей C#. Вы можете предоставлять разные уровни доступа, выполнять проверки, отложенное вычисление или любые другие требования, необходимые в вашем сценарии.

Индексаторы

18.12.2021 • 8 minutes to read

Индексаторы аналогичны свойствам. Во многих отношениях индексаторы основаны на тех же компонентах языка, что и свойства. Индексаторы обеспечивают поддержку *индексированных* свойств: свойств, на которые ссылаются с помощью одного или нескольких аргументов. Эти аргументы предоставляют индекс в определенную коллекцию значений.

Синтаксис индексаторов

Для обращения к индексатору используется имя переменной и квадратные скобки. Аргументы индексатора необходимо поместить в квадратные скобки:

```
var item = someObject["key"];
someObject["AnotherKey"] = item;
```

Для объявления индексаторов используется ключевое слово `this` в качестве имени свойства, а аргументы объявляются в квадратных скобках. Это объявление соответствует примеру использования, показанному в предыдущем абзаце:

```
public int this[string key]
{
    get { return storage.Find(key); }
    set { storage.SetAt(key, value); }
}
```

Наш первый пример помогает понять связь между синтаксисом свойств и индексаторов. Эта аналогия справедлива для большей части синтаксических правил для индексаторов. Индексаторы могут иметь любые допустимые модификаторы доступа (общедоступный, защищенный внутренний, защищенный, внутренний, закрытый или закрытый защищенный). Они могут быть запечатанными, виртуальными или абстрактными. Как и для свойств, вы можете указать разные модификаторы доступа для методов доступа set и get в индексаторе. Можно также указать индексаторы только для чтения (опуская метод доступа set) или только для записи (опуская метод доступа get).

К индексаторам можно применять практически все возможности, предусмотренные для свойств. Единственное исключение из этого правила — *автоматически реализуемые свойства*. Компилятор не всегда может создать правильное хранилище для индексатора.

Наличие аргументов для ссылки на элемент в наборе элементов отличает индексаторы от свойств. Можно определить несколько индексаторов для типа, при условии что списки аргументов для каждого индексатора являются уникальными. Рассмотрим различные сценарии, в которых можно использовать один или несколько индексаторов в определении класса.

Сценарии

Вам потребуется определить *индексаторы* в типе, если его API моделирует некоторую коллекцию, где определяются аргументы для этой коллекции. Индексаторы могут сопоставляться или не сопоставляться напрямую с типами коллекций, которые являются частью основной платформы .NET. Тип может иметь другие обязанности, помимо моделирования коллекции. Индексаторы позволяют предоставить API, который соответствует абстракции данного типа, не раскрывая внутренних сведений о том, как хранятся

или вычисляются значения этой абстракции.

Давайте рассмотрим некоторые распространенные сценарии использования *индексаторов*. Вы можете воспользоваться папкой с примерами индексаторов. Инструкции по загрузке см. в разделе Просмотр и скачивание примеров.

Массивы и векторы

Один из наиболее распространенных сценариев для создания индексаторов — когда тип моделирует массив или вектор. Можно создать индексатор для моделирования упорядоченного набора данных.

Преимущество создания собственного индексатора заключается в том, что вы можете определить хранилище для этой коллекции в соответствии с потребностями. Представьте себе ситуацию, когда ваш тип моделирует исторические данные, которые слишком велики для загрузки в память за один раз. Вам потребуется загружать и выгружать разделы коллекции по мере использования. Следующий пример моделирует такое поведение. Он сообщает о том, сколько точек данных существует. Он создает страницы для хранения разделов данных по требованию. Он удаляет страницы из памяти, чтобы освободить место для страниц, которые требуются более поздним запросам.

```
public class DataSamples
{
    private class Page
    {
        private readonly List<Measurements> pageData = new List<Measurements>();
        private readonly int startingIndex;
        private readonly int length;
        private bool dirty;
        private DateTime lastAccess;

        public Page(int startingIndex, int length)
        {
            this.startingIndex = startingIndex;
            this.length = length;
            lastAccess = DateTime.Now;

            // This stays as random stuff:
            var generator = new Random();
            for(int i=0; i < length; i++)
            {
                var m = new Measurements
                {
                    HiTemp = generator.Next(50, 95),
                    LoTemp = generator.Next(12, 49),
                    AirPressure = 28.0 + generator.NextDouble() * 4
                };
                pageData.Add(m);
            }
        }
        public bool HasItem(int index) =>
            ((index >= startingIndex) &&
            (index < startingIndex + length));

        public Measurements this[int index]
        {
            get
            {
                lastAccess = DateTime.Now;
                return pageData[index - startingIndex];
            }
            set
            {
                pageData[index - startingIndex] = value;
                dirty = true;
                lastAccess = DateTime.Now;
            }
```

```
        }

        public bool Dirty => dirty;
        public DateTime LastAccess => lastAccess;
    }

    private readonly int totalSize;
    private readonly List<Page> pagesInMemory = new List<Page>();

    public DataSamples(int totalSize)
    {
        this.totalSize = totalSize;
    }

    public Measurements this[int index]
    {
        get
        {
            if (index < 0)
                throw new IndexOutOfRangeException("Cannot index less than 0");
            if (index >= totalSize)
                throw new IndexOutOfRangeException("Cannot index past the end of storage");

            var page = updateCachedPagesForAccess(index);
            return page[index];
        }
        set
        {
            if (index < 0)
                throw new IndexOutOfRangeException("Cannot index less than 0");
            if (index >= totalSize)
                throw new IndexOutOfRangeException("Cannot index past the end of storage");
            var page = updateCachedPagesForAccess(index);

            page[index] = value;
        }
    }

    private Page updateCachedPagesForAccess(int index)
    {
        foreach (var p in pagesInMemory)
        {
            if (p.HasItem(index))
            {
                return p;
            }
        }
        var startingIndex = (index / 1000) * 1000;
        var newPage = new Page(startingIndex, 1000);
        addPageToCache(newPage);
        return newPage;
    }

    private void addPageToCache(Page p)
    {
        if (pagesInMemory.Count > 4)
        {
            // remove oldest non-dirty page:
            var oldest = pagesInMemory
                .Where(page => !page.Dirty)
                .OrderBy(page => page.LastAccess)
                .FirstOrDefault();
            // Note that this may keep more than 5 pages in memory
            // if too much is dirty
            if (oldest != null)
                pagesInMemory.Remove(oldest);
        }
        pagesInMemory.Add(p);
    }
```

```
}
```

Вы можете следовать этой идиоме проектирования для моделирования коллекций любых типов, если у вас имеются веские причины для того, чтобы не загружать весь набор данных в коллекции в память. Обратите внимание, что класс `Page` является закрытым вложенным классом, который не является частью открытого интерфейса. Эти подробности скрыты от пользователей этого класса.

Словари

Другим распространенным сценарием является необходимость моделирования словаря или карты. Этот сценарий подразумевает хранение в типе значений на основе ключа, как правило, текстовых ключей. В этом примере создается словарь, сопоставляющий аргументы командной строки с лямбда-выражениями, управляющими этими параметрами. В следующем примере показано два класса: класс `ArgsActions`, сопоставляющий параметр командной строки делегату `Action`, и `ArgsProcessor`, использующий `ArgsActions` для выполнения каждого объекта `Action` при обнаружении соответствующего параметра.

```
public class ArgsProcessor
{
    private readonly ArgsActions actions;

    public ArgsProcessor(ArgsActions actions)
    {
        this.actions = actions;
    }

    public void Process(string[] args)
    {
        foreach(var arg in args)
        {
            actions[arg]?.Invoke();
        }
    }

}
public class ArgsActions
{
    readonly private Dictionary<string, Action> argsActions = new Dictionary<string, Action>();

    public Action this[string s]
    {
        get
        {
            Action action;
            Action defaultAction = () => {} ;
            return argsActions.TryGetValue(s, out action) ? action : defaultAction;
        }
    }

    public void SetOption(string s, Action a)
    {
        argsActions[s] = a;
    }
}
```

В этом примере коллекция `ArgsAction` точно соответствует базовой коллекции. `get` определяет, настроен ли данный параметр. Если это так, он возвращает объект `Action`, связанный с этим параметром. В противном случае он возвращает объект `Action`, не выполняющий никаких действий. Открытый метод доступа не включает метод доступа `set`. Вместо этого в проекте используется общедоступный метод для настройки параметров.

Многомерные сопоставления

Можно создавать индексаторы, использующие несколько аргументов. Кроме того, эти аргументы не ограничиваются одним типом. Рассмотрим два примера.

В первом примере показан класс, который создает значения для множества Мандельброта. Дополнительные сведения о математических принципах этого множества см. в этой статье. Индексатор использует два значения double для определения точки на плоскости X и Y. Метод доступа get вычисляет количество итераций до определения точки, не входящей в это множество. Если достигается максимальное количество итераций, точка находится в множестве и возвращается значение maxIterations класса. (Компьютер создает изображения на основе заданных цветов множества Мандельброта для числа итераций, необходимых для определения того, что точка находится за пределами множества.)

```
public class Mandelbrot
{
    readonly private int maxIterations;

    public Mandelbrot(int maxIterations)
    {
        this.maxIterations = maxIterations;
    }

    public int this [double x, double y]
    {
        get
        {
            var iterations = 0;
            var x0 = x;
            var y0 = y;

            while ((x*x + y * y < 4) &&
                (iterations < maxIterations))
            {
                var newX = x * x - y * y + x0;
                y = 2 * x * y + y0;
                x = newX;
                iterations++;
            }
            return iterations;
        }
    }
}
```

Множество Мандельброта определяет значения в каждой координате (x, y) для значений реальных чисел. Это определяет словарь, который может содержать бесконечное количество значений. Таким образом, множество не основано на хранилище. Напротив, этот класс вычисляет значение для каждой точки, когда код вызывает метод доступа `get`. Базовое хранилище не используется.

Теперь рассмотрим один из последних примеров использования индексаторов, где индексатор принимает несколько аргументов разных типов. Рассмотрим программу, которая управляет историческими данными о температуре. Этот индексатор использует город и дату для задания или получения высоких и низких температур для этого расположения:

```
using DateMeasurements =
    System.Collections.Generic.Dictionary<System.DateTime, IndexersSamples.Common.Measurements>;
using CityDataMeasurements =
    System.Collections.Generic.Dictionary<string, System.Collections.Generic.Dictionary<System.DateTime,
IndexersSamples.Common.Measurements>>;

public class HistoricalWeatherData
{
    readonly CityDataMeasurements storage = new CityDataMeasurements();

    public Measurements this[string city, DateTime date]
    {
        get
        {
            var cityData = default(DateMeasurements);

            if (!storage.TryGetValue(city, out cityData))
                throw new ArgumentOutOfRangeException(nameof(city), "City not found");

            // strip out any time portion:
            var index = date.Date;
            var measure = default(Measurements);
            if (cityData.TryGetValue(index, out measure))
                return measure;
            throw new ArgumentOutOfRangeException(nameof(date), "Date not found");
        }
        set
        {
            var cityData = default(DateMeasurements);

            if (!storage.TryGetValue(city, out cityData))
            {
                cityData = new DateMeasurements();
                storage.Add(city, cityData);
            }

            // Strip out any time portion:
            var index = date.Date;
            cityData[index] = value;
        }
    }
}
```

В этом примере создается индексатор, который сопоставляет данные о погоде по двум разным аргументам: городу (представленному `string`) и дате (представленной `DateTime`). Внутреннее хранилище использует два класса `Dictionary`, представляющие двухмерный словарь. Открытый API больше не представляет базовое хранилище. Вместо этого функции языка для индексаторов позволяют создать открытый интерфейс, который представляет абстракцию, несмотря на то, что базовое хранилище должно использовать разные базовые типы коллекции.

Этот код включает две части, которые могут быть незнакомы некоторым разработчикам. Эти две директивы `using`:

```
using DateMeasurements = System.Collections.Generic.Dictionary<System.DateTime,
IndexersSamples.Common.Measurements>;
using CityDataMeasurements = System.Collections.Generic.Dictionary<string,
System.Collections.Generic.Dictionary<System.DateTime, IndexersSamples.Common.Measurements>>;
```

создают *псевдоним* сконструированного универсального типа. Эти операторы позволяют коду позднее использовать более описательные имена `DateMeasurements` и `CityDataMeasurements` вместо универсальной конструкции `Dictionary<DateTime, Measurements>` и `Dictionary<string, Dictionary<DateTime, Measurements> >`.

Эта конструкция требует использования полных имен типов в правой части равенства `=`.

Второй прием — отбросить части времени любого объекта `DateTime`, используемого для индексации в коллекции. .NET не включает тип "только дата". Разработчики используют тип `DateTime`, однако использование свойства `Date` для проверки того, что любой объект `DateTime` принадлежит к заданной дате, равноценно.

Заключение

Индексаторы следует создавать при наличии в вашем классе элемента, подобного свойству, если такое свойство представляет не одно значение, а коллекцию значений, где каждый отдельный элемент определяется набором аргументов. Эти аргументы могут однозначно определять, на какой элемент в коллекции необходимо ссылаться. Индексаторы расширяют концепцию свойств, где член обрабатывается как элемент данных за пределами класса и как метод — в его пределах. Индексаторы позволяют аргументам находить в свойстве один элемент, который представляет набор элементов.

Iterators

18.12.2021 • 5 minutes to read

Почти каждой написанной вами программе придется выполнять итерацию определенной коллекции. Для этого вы напишете код, проверяющий каждый элемент в коллекции.

Кроме того, вы создадите методы итератора, то есть методы, которые создают *итератор* для элементов соответствующего класса. *Итератор* — это объект, который выполняет обход контейнера, в частности списков. Итераторы можно использовать для следующих целей:

- Выполнение определенного действия с каждым элементом в коллекции.
- Перечисление настраиваемой коллекции.
- Расширение LINQ или других библиотек.
- Создание конвейера данных, обеспечивающего эффективный поток данных через методы итератора.

Язык C# предоставляет возможности для создания и использования последовательностей. Эти последовательности можно создавать и использовать синхронно или асинхронно. В этой статье представлены общие сведения об этих функциях.

Итерация для каждого

Вы полнить перечисление коллекции несложно. Ключевое слово `foreach` перечисляет коллекцию, выполняя внедренный оператор по одному разу для каждого элемента в коллекции:

```
foreach (var item in collection)
{
    Console.WriteLine(item.ToString());
}
```

Вот и все. Для итерации содержимого той или иной коллекции нужен только это оператор `foreach`. При этом в работе оператора `foreach` нет ничего сложного. Он создает код, необходимый для итерации коллекции, опираясь на два универсальных интерфейса, определенных в библиотеке ядра .NET: `IEnumerable<T>` и `IEnumerator<T>`. Более подробно этот механизм рассматривается ниже.

Оба этих интерфейса также имеют неуниверсальные аналоги: `IEnumerable` и `IEnumerator`. Универсальные версии более предпочтительны для современного кода.

Если последовательность создается асинхронно, можно использовать инструкцию `await foreach` для асинхронного использования этой последовательности.

```
await foreach (var item in asyncSequence)
{
Console.WriteLine(item.ToString());
}
```

Если последовательность — это System.Collections.Generic.IEnumerable<T>, используется `foreach`. Если последовательность — это System.Collections.Generic.IAsyncEnumerable<T>, используется `await foreach`. В последнем случае последовательность создается асинхронно.

Источники перечисления с применением методов итератора

Еще одна полезная функция языка C# позволяет выполнять сборку методов, создающих источник для перечисления. Эти методы называются *методами итератора*. Метод итератора определяет, какие образом будут создаваться объекты в последовательности по запросу. Метод итератора определяется с помощью контекстных ключевых слов `yield return`.

Напишем метод итератора, выдающий последовательность целых чисел от 0 до 9:

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    yield return 0;
    yield return 1;
    yield return 2;
    yield return 3;
    yield return 4;
    yield return 5;
    yield return 6;
    yield return 7;
    yield return 8;
    yield return 9;
}
```

Отдельные операторы `yield return` в этом коде показывают, что в любом методе итератора можно использовать сразу несколько дискретных операторов `yield return`. Другие языковые конструкции можно (и нужно) включать для того, чтобы код метода итератора стал более простым. Точно такую же последовательность чисел выдает определение метода, приведенное ниже:

```
public IEnumerable<int> GetSingleDigitNumbersLoop()
{
    int index = 0;
    while (index < 10)
        yield return index++;
}
```

Выбирать какой-то один из этих вариантов необязательно. В код можно добавлять столько операторов `yield return`, сколько требуется для вашего метода:

```
public IEnumerable<int> GetSetsOfNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    index = 100;
    while (index < 110)
        yield return index++;
}
```

Все приведенные выше примеры будут иметь асинхронный аналог. В каждом случае необходимо заменить тип возвращаемого значения `IEnumerable<T>` на `IAsyncEnumerable<T>`. Например, в предыдущем примере будет создана следующая асинхронная версия:

```
public async IAsyncEnumerable<int> GetSetsOfNumbersAsync()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    await Task.Delay(500);

    yield return 50;

    await Task.Delay(500);

    index = 100;
    while (index < 110)
        yield return index++;
}
```

Это синтаксис применим как для синхронных, так и для асинхронных итераторов. Давайте рассмотрим практический пример. Допустим, вы занимаетесь проектом IoT и имеете дело с датчиками устройств, которые создают большой поток данных. Чтобы получить представление об этих данных, можно написать метод, формирующий выборку из каждого N-го элемента данных. С этой задачей справится вот такой небольшой метод итератора:

```
public static IEnumerable<T> Sample<T>(this IEnumerable<T> sourceSequence, int interval)
{
    int index = 0;
    foreach (T item in sourceSequence)
    {
        if (index++ % interval == 0)
            yield return item;
    }
}
```

Если при чтении из устройства Интернета вещей создается асинхронная последовательность, метод необходимо изменить, как показано в следующем методе:

```
public static async IAsyncEnumerable<T> Sample<T>(this IAsyncEnumerable<T> sourceSequence, int interval)
{
    int index = 0;
    await foreach (T item in sourceSequence)
    {
        if (index++ % interval == 0)
            yield return item;
    }
}
```

Для методов итератора действует одно важное ограничение: в одном и том же методе не могут одновременно присутствовать операторы `return` и `yield return`. Следующий код не будет компилироваться:

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    // generates a compile time error:
    var items = new int[] {100, 101, 102, 103, 104, 105, 106, 107, 108, 109 };
    return items;
}
```

Обычно это ограничение не вызывает проблем. Вы можете либо использовать в методе операторы `yield return`, либо разделить исходный метод на несколько отдельных методов, одни из которых будут включать оператор `return`, а другие — `yield return`.

Немного изменим последний метод, вставив в каждом случае оператор `yield return`:

```
public IEnumerable<int> GetFirstDecile()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    var items = new int[] {100, 101, 102, 103, 104, 105, 106, 107, 108, 109 };
    foreach (var item in items)
        yield return item;
}
```

В некоторых случаях метод итератора лучше разбить на два разных метода. В одном будет использоваться оператор `return`, а в другом — `yield return`. Допустим, вам нужно получить пустую коллекцию или первые пять нечетных чисел, используя логический аргумент. Для этого можно написать следующие два метода:

```
public IEnumerable<int> GetSingleDigitOddNumbers(bool getCollection)
{
    if (getCollection == false)
        return new int[0];
    else
        return IteratorMethod();
}

private IEnumerable<int> IteratorMethod()
{
    int index = 0;
    while (index < 10)
    {
        if (index % 2 == 1)
            yield return index;
        index++;
    }
}
```

Посмотрите на приведенные выше методы. В первом используется стандартный оператор `return`, который возвращает либо пустую коллекцию, либо итератор, созданный вторым методом. Второй метод включает оператор `yield return`, создающий запрошенную последовательность.

Подробнее об операторе `foreach`

Оператор `foreach` разворачивается в стандартную идиому, которая выполняет итерацию всех элементов в коллекции с помощью интерфейсов `IEnumerable<T>` и `IEnumerator<T>`. Кроме того, он сводит к минимуму ошибки, допускаемые разработчиками в результате неправильного управления ресурсами.

Компилятор преобразует цикл `foreach`, показанный в первом примере, в конструкцию следующего вида:

```
IEnumerator<int> enumerator = collection.GetEnumerator();
while (enumerator.MoveNext())
{
    var item = enumerator.Current;
    Console.WriteLine(item.ToString());
}
```

На практике компилятор создает более сложный код и устраняет ситуации, когда объект, возвращаемый методом `GetEnumerator()`, реализует интерфейс `IDisposable`. Полная версия кода выглядит так:

```
{
    var enumerator = collection.GetEnumerator();
    try
    {
        while (enumerator.MoveNext())
        {
            var item = enumerator.Current;
            Console.WriteLine(item.ToString());
        }
    }
    finally
    {
        // dispose of enumerator.
    }
}
```

Компилятор преобразует первую асинхронную выборку в конструкцию следующего вида:

```
{
    var enumerator = collection.GetAsyncEnumerator();
    try
    {
        while (await enumerator.MoveNextAsync())
        {
            var item = enumerator.Current;
            Console.WriteLine(item.ToString());
        }
    }
    finally
    {
        // dispose of async enumerator.
    }
}
```

Способ ликвидации перечислителя зависит от характеристик типа `enumerator`. В общем случае синхронного использования предложение `finally` разворачивается следующим образом:

```
finally
{
   (enumerator as IDisposable)?.Dispose();
}
```

Общий случай асинхронного использования разворачивается следующим образом:

```
finally
{
    if (enumerator is IAsyncDisposable asyncDisposable)
        await asyncDisposable.DisposeAsync();
}
```

Однако если тип `enumerator` является запечатанным, а тип `enumerator` не подвергается явному преобразованию в тип `IDisposable` или `IAsyncDisposable`, предложение `finally` разворачивается в пустой блок:

```
finally
{
}
```

Если же тип `enumerator` подвергается неявному преобразованию в тип `IDisposable`, а `enumerator` является типом значения, не допускающим значение NULL, предложение `finally` разворачивается следующим образом:

```
finally
{
   ((IDisposable)enumerator).Dispose();
}
```

К счастью, запоминать все это не нужно. Оператор `foreach` обрабатывает все эти нюансы в фоновом режиме, а компилятор создает правильный код для любой из этих конструкций.

Общие сведения о делегатах

18.12.2021 • 2 minutes to read

Делегаты предоставляют механизм *позднего связывания* в .NET. Позднее связывание означает, что создается алгоритм, где вызывающий объект также предоставляет по крайней мере один метод, который реализует часть алгоритма.

Например, рассмотрим сортировку списка звезд в астрономическом приложении. Можно отсортировать звезды по расстоянию от Земли, по величине или по воспринимаемой яркости.

Во всех этих случаях метод Sort() выполняет, по сути, одно и то же: упорядочивает элементы в списке на основе некоего сравнения. Для каждого порядка сортировки используется разный код, сравнивающий две звезды.

Такого рода решения использовались в программном обеспечении в течение полувека. Концепция использования делегатов в языке C# обеспечивает первоклассную поддержку языка и безопасность типов.

Как вы увидите далее в этой серии статей, код C#, создаваемый для подобных алгоритмов, является строго типизированным и использует правила языка и компилятор для соответствия типов аргументам и типам возвращаемых значений.

В C# 9 для похожих сценариев, где требуется больший контроль над соглашением о вызовах, были добавлены указатели на функции. Код, связанный с делегатом, вызывается с помощью виртуального метода, добавленного к типу делегата. Используя указатели функций, можно указать другие соглашения.

Цели разработки языка для делегатов

Разработчики, использующие язык, определили несколько целей для функций, которые в итоге стали делегатами.

Группе разработчиков требовалась общая языковая конструкция, которую можно было бы использовать для любых алгоритмов позднего связывания. Делегаты позволяют разработчикам изучить одну концепцию и применять ее для решения множества различных задач программного обеспечения.

Во-вторых, команде нужна поддержка одиночных и многоадресных вызовов методов. (Многоадресные делегаты — это делегаты, которые объединяют в цепочку несколько вызовов методов. Вы увидите примеры далее в этой серии.)

Группа разработчиков хотела, чтобы делегаты поддерживали ту же безопасность типа, ожидаемую от всех конструкций C#.

Наконец группа пришла к выводу, что шаблон событий является определенным шаблоном, где использование делегатов (или любого алгоритма позднего связывания) очень эффективно. Разработчикам требовалось, чтобы код для делегатов служил основой для шаблона событий .NET.

Результатом всей этой работы стала поддержка делегатов и событий в C# и .NET. В оставшихся статьях в этом разделе будут рассматриваться возможности языка, поддержка библиотек и распространенные идиомы, которые применяются при работе с делегатами.

Вы узнаете о ключевом слове `delegate` и коде, который он создает. Вы узнаете о функциях в классе `System.Delegate` и их использовании. Вы научитесь создавать типобезопасные делегаты и ознакомитесь со способами создания методов, которые можно вызывать с помощью делегатов. Вы также узнаете, как

работать с делегатами и событиями с помощью лямбда-выражения. Вы увидите, каким образом делегаты становятся одними из стандартных блоков для LINQ. Вы узнаете, что делегаты являются основой для шаблона событий .NET, и определите их отличия.

Итак, начнем.

Вперед

System.Delegate и ключевое слово `delegate`

18.12.2021 • 5 minutes to read

Назад

В этой статье описываются классы в .NET, поддерживающие делегаты, и рассматриваются способы их сопоставления с ключевым словом `delegate`.

Определение типов делегатов

Начнем с ключевого слова "delegate", потому что это основной элемент, используемый при работе с делегатами. Код, который компилятор создает при использовании ключевого слова `delegate`, будет сопоставляться с вызовами методов, вызывающих члены классов Delegate и MulticastDelegate.

Тип делегата определяется с помощью синтаксиса, подобному синтаксису определения сигнатуры метода. К определению нужно просто добавить ключевое слово `delegate`.

В качестве примера будем по-прежнему использовать метод List.Sort(). Первым шагом является создание типа для делегата сравнения.

```
// From the .NET Core library

// Define the delegate type:
public delegate int Comparison<in T>(T left, T right);
```

Компилятор создает класс, производный от `System.Delegate`, соответствующий используемой сигнатуре (в данном случае — метод, который возвращает целое число и имеет два аргумента). Тип делегата — `Comparison`. Тип делегата `Comparison` является универсальным типом. Подробные сведения об универсальных типах см. здесь.

Обратите внимание, что синтаксис может отображаться так, будто он объявляет переменную, но на самом деле он объявляет *тип*. Можно определить типы делегатов внутри классов, непосредственно внутри пространств имен или даже в глобальном пространстве имен.

> **NOTE**
> Не рекомендуется объявлять типы делегатов (или другие типы) непосредственно в глобальном пространстве имен.

Компилятор также создает обработчики добавления и удаления для этого нового типа, чтобы клиенты этого класса могли добавлять и удалять методы из списка вызовов экземпляра. Компилятор будет обеспечивать соответствие подписи добавляемого или удаляемого метода подписи, используемой при объявлении метода.

Объявление экземпляров делегатов

После определения делегата можно создать экземпляр этого типа. Как и все переменные в C#, экземпляры делегата нельзя объявлять непосредственно в пространстве имен или в глобальном пространстве имен.

```
// inside a class definition:

// Declare an instance of that type:
public Comparison<T> comparator;
```

Тип переменной — `Comparison<T>`, тип делегата определен ранее. Имя переменной — `comparator`.

В приведенном выше фрагменте кода была объявлена переменная-член в классе. Можно также объявить переменные делегатов, локальные переменные или аргументов для методов.

Вызов делегатов

Чтобы вызвать методы, которые находятся в списке вызова делегата, нужно вызвать этот делегат. В методе `Sort()` код вызовет метод сравнения, чтобы определить порядок размещения объектов:

```
int result = comparator(left, right);
```

В строке выше код *вызывает* метод, подключенный к делегату. Переменная считается именем метода и вызывается с помощью обычного синтаксиса вызова метода.

Эта строка кода содержит небезопасное условие: нет никакой гарантии, что целевой объект было добавлен к делегату. Если целевые объекты не были вложены, строка выше приведет к возникновению исключения `NullReferenceException`. Идиомы, используемые для решения этой проблемы, более сложны, чем простые проверки значений NULL, и рассматриваются далее в этой серии материалов.

Назначение, добавление и удаление целевых объектов вызова

Сведения о том, как определяется тип делегата и как объявляются и вызываются экземпляры делегата.

Разработчикам, которые хотят использовать метод `List.Sort()`, нужно определить метод, сигнатура которого совпадает с определением типа делегата, и назначить его делегату, используемому методом sort. Это назначение добавляет метод в список вызовов данного делегата объекта.

Предположим, требуется отсортировать список строк по их длине. Функция сравнения может выглядеть следующим образом:

```
private static int CompareLength(string left, string right) =>
    left.Length.CompareTo(right.Length);
```

Метод объявляется как закрытый метод. Все правильно. Вам может быть не нужно, чтобы этот метод был частью общедоступного интерфейса. Этот метод, присоединенный к делегату, по-прежнему можно использовать в качестве метода сравнения. В вызывающем коде этот метод будет присоединен к целевому списку объекта делегата и будет доступен через этот делегат.

Чтобы создать эту связь, передайте метод в метод `List.Sort()`:

```
phrases.Sort(CompareLength);
```

Обратите внимание, что имя метода используется без скобок. Использование метода как аргумента указывает компилятору преобразовать ссылку на метод в ссылку, которая может применяться как целевой объект вызова делегата, и присоединить этот метод в качестве целевого объекта вызова.

Вы также явно объявили переменную типа `Comparison<string>` и выполнили назначение:

```
Comparison<string> comparer = CompareLength;
phrases.Sort(comparer);
```

Если в качестве объекта делегата используется небольшой метод, для назначения обычно применяется синтаксис лямбда-выражения:

```
Comparison<string> comparer = (left, right) => left.Length.CompareTo(right.Length);
phrases.Sort(comparer);
```

Использование лямбда-выражений для целевых объектов делегатов рассматривается более подробно в следующем разделе.

В примере Sort() к делегату обычно подключается один целевой метод. Однако объекты делегатов поддерживают списки вызовов, где к объекту делегата присоединено несколько целевых методов.

Классы Delegate и MulticastDelegate

Описанная выше поддержка языка предоставляет функции и поддержку, которые обычно необходимы для работы с делегатами. Эти возможности основаны на двух классах в платформе .NET Core: Delegate и MulticastDelegate.

Класс `System.Delegate` и его прямой вложенный класс `System.MulticastDelegate` обеспечивают поддержку платформы для создания делегатов, регистрации методов в качестве целевых объектов делегатов и вызова всех методов, которые зарегистрированы как целевые объекты делегатов.

Что интересно, сами классы `System.Delegate` и `System.MulticastDelegate` не являются типами делегатов. Они являются основой для всех конкретных типов делегатов. Тот же процесс конструкции языка требует, что нельзя объявить класс, который является производным от `Delegate` или `MulticastDelegate`. Это запрещено правилами языка C#.

Вместо этого компилятор C# создает экземпляры класса, производного от `MulticastDelegate`, при использовании ключевого слова языка C# для объявления типов делегатов.

Такая схема впервые появилась в первом выпуске C# и .NET. Одной из целей команды разработки было обеспечение соблюдения типобезопасности при использовании делегатов. Это значило, что делегаты вызывались с помощью правильного типа и числа аргументов. И что любой возвращаемый тип был правильно указан во время компиляции. Делегаты входили в выпуск 1.0 .NET, который действовал до универсальных типов.

Наилучший способ обеспечения безопасности типа заключался в том, что компилятор должен был создать конкретные классы делегатов, представляющих используемую сигнатуру метода.

Несмотря на то, что невозможно создать производные классы напрямую, вы будете использовать методы, определенные в этих классах. Разберем наиболее распространенные методы, которые будут использоваться при работе с делегатами.

Первый и самый важный момент состоит в том, что каждый делегат является производным от `MulticastDelegate`. Многоадресный делегат означает, что при вызове с помощью делегата может быть вызвано несколько целевых объектов метода. В исходной структуре планировалось проводить различие между делегатами, где можно было прикреплять и вызывать только один целевой метод, и делегатами, где можно было прикреплять и вызывать несколько целевых методов. На практике это различие оказалась менее полезным, чем предполагалось изначально. Уже были созданы два разных класса, которые находились в платформе с момента первого общедоступного выпуска.

Методы, которые чаще всего будут использоваться с делегатами, — `Invoke()` и `BeginInvoke()` /

`EndInvoke()`. `Invoke()` будет вызывать все методы, которые были прикреплены к определенному экземпляру делегата. Как было показано выше, для вызова делегатов обычно используется синтаксис вызова метода в переменной делегата. Далее в этой серии материалов вы узнаете о шаблонах, которые работают непосредственно с этими методами.

Ознакомившись с синтаксисом языка и классами, поддерживающими делегаты, давайте рассмотрим способы использования, создания и вызова строго типизированных делегатов.

Вперед

Строго типизированные делегаты

18.12.2021 • 2 minutes to read

Назад

Из предыдущей статьи вы узнали, как создавать определенные типы делегатов с помощью ключевого слова `delegate`.

Абстрактный класс делегата предоставляет инфраструктуру для слабой взаимозависимости (loose coupling) и вызовов. Конкретные типы делегата становятся гораздо полезнее, поскольку включают и обеспечивают безопасность типов для методов, добавляемых в список вызовов для объекта делегата. Компилятор создает эти методы, если вы используете ключевое слово `delegate` и определяете конкретный тип делегата.

В результате новые типы делегатов создаются всякий раз, когда возникает необходимость в новой сигнатуре метода. Со временем эта работа может стать слишком громоздкой. Для каждого нового компонента требуются новые типы делегатов.

К счастью, это необязательно. Платформа .NET Core содержит несколько типов, которые можно использовать всякий раз, когда вам нужно делегировать типы. Эти определения универсальны, поэтому всякий раз, когда вам нужно объявить новый метод, можно объявить настройку.

Первый из этих типов — это тип Action и несколько вариантов:

```
public delegate void Action();
public delegate void Action<in T>(T arg);
public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
// Other variations removed for brevity.
```

Модификатор `in` для аргумента универсального типа рассматривается в статье о ковариации.

Существуют варианты делегата `Action`, содержащие до 16 аргументов, такие как Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16>. Очень важно, чтобы для каждого из аргументов делегата в этих определениях использовались универсальные аргументы — это дает максимальную гибкость. Аргументы метода могут быть одного типа, однако это необязательно.

В качестве типа делегата используйте один из типов `Action` с типом возвращаемого значения void.

Платформа включает также несколько типов универсальных делегатов, которые можно использовать в качестве типов делегатов, возвращающих значения:

```
public delegate TResult Func<out TResult>();
public delegate TResult Func<in T1, out TResult>(T1 arg);
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
// Other variations removed for brevity
```

Модификатор `out` для аргумента универсального типа результата рассматривается в статье о ковариации.

Существуют варианты делегата `Func`, содержащие до 16 входных аргументов, такие как Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,TResult>. Как правило, тип результата должен быть последним параметром типа во всех объявлениях `Func`.

В качестве типа делегата, возвращающего значение, используйте один из типов `Func`.

Также есть специализированный тип Predicate<T>, предназначенный для делегата, возвращающего тест одного значения:

```
public delegate bool Predicate<in T>(T obj);
```

Можно заметить, что для каждого типа `Predicate` существует структурно эквивалентный тип `Func`, например:

```
Func<string, bool> TestForString;
Predicate<string> AnotherTestForString;
```

Можно подумать, что два эти типа эквивалентны. Это не так. Эти две переменные не заменяют друг друга. Переменную одного типа нельзя назначить другому типу. В системе типов C# используются имена определенных типов, а не структуры.

Все эти определения типов делегатов в библиотеке .NET Core должны означать, что новый тип делегата для каждой создаваемой функции, которой требуются делегаты, создавать не нужно. Эти универсальные определения должны предоставлять типы делегатов, необходимые в большинстве ситуаций. Можно просто создать экземпляр одного из этих типов с необходимыми параметрами типа. Если алгоритмы можно сделать универсальными, эти делегаты можно использовать как универсальные типы.

Это позволит сэкономить время и свести к минимуму число новых типов, которые нужно создать для работы с делегатами.

В следующей статье представлено несколько общих шаблонов для работы с делегатами на практике.

Вперед

Общие шаблоны делегатов

18.12.2021 • 7 minutes to read

Назад

Делегаты предоставляют механизм, который обеспечивает проектирование программного обеспечения с минимальной взаимозависимостью между компонентами.

Отличным примером такого проектирования является LINQ. В модели выражений запросов LINQ делегаты применяются для обеспечения всех возможностей. Рассмотрим простой пример.

```
var smallNumbers = numbers.Where(n => n < 10);
```

В нем из последовательности чисел отфильтровываются только числа со значением меньше 10. Метод `Where` использует делегат, который определяет, какие элементы последовательности проходят через фильтр. При создании запроса LINQ вы предоставляете реализацию делегата для этой цели.

Прототип метода Where имеет следующий вид:

```
public static IEnumerable<TSource> Where<TSource> (this IEnumerable<TSource> source, Func<TSource, bool>
predicate);
```

Этот пример актуален для всех методов, которые относятся к LINQ. Все они используют делегаты для управления определенными запросами. Этот конструктивный шаблон API очень эффективен, что делает его важным для изучения и понимания.

Из этого примера видно, что делегаты почти не требуют взаимозависимости между компонентами. Не нужно создавать класс, производный от некоторого базового класса. Не нужно реализовывать определенный интерфейс. Единственным требованием является предоставление реализации одного метода, на основе которого решается поставленная задача.

Создание собственных компонентов с помощью делегатов

Давайте продолжим пример, создав компонент с помощью модели, основанной на делегатах.

Определим компонент, который можно использовать для сообщений журнала в большой системе. Компоненты библиотеки можно использовать во множестве разных сред на различных платформах. В компонентах, управляющих журналами, много общих черт. Они должны принимать сообщения от любого компонента системы. Эти сообщения имеют разные приоритеты, которыми управляет основной компонент. В окончательной архивной форме сообщений должны быть метки времени. В более сложных сценариях может потребоваться фильтровать сообщения по исходному компоненту.

Часто будет меняться один аспект: куда записываются сообщения. В некоторых средах они могут записываться в консоль ошибок. В других — в файл. К иным вариантам относятся хранилище базы данных, журналы событий ОС или иные хранилища документов.

В некоторых ситуациях могут использоваться сочетания назначений вывода. Например, сообщения могут записываться в консоль и в файл.

Модель на основе делегатов обеспечивает высокую гибкость и упрощает поддержку механизмов хранения, добавляемых в будущем.

В рамках этой модели основным компонентом журнала может быть невиртуальный и даже запечатанный класс. Вы можете подключать любой набор делегатов для записи сообщений на различные носители данных. Встроенная поддержка делегатов многоадресной рассылки позволяет легко реализовывать сценарии, в которых сообщения должны записываться в несколько расположений (в файл и консоль).

Первая реализация

Начнем с малого: начальная реализация будет принимать новые сообщения и записывать их с помощью любого подключенного делегата. Можно начать с одного делегата, который записывает сообщения в консоль.

```
public static class Logger
{
    public static Action<string> WriteMessage;

    public static void LogMessage(string msg)
    {
        WriteMessage(msg);
    }
}
```

Приведенный выше статический класс содержит только самое необходимое для работы. Нам нужно создать единственную реализацию метода, который записывает сообщения в консоль.

```
public static class LoggingMethods
{
    public static void LogToConsole(string message)
    {
        Console.Error.WriteLine(message);
    }
}
```

Наконец, необходимо подключить делегат к делегату WriteMessage, объявленному в средстве ведения журнала.

```
Logger.WriteMessage += LoggingMethods.LogToConsole;
```

Рекомендации

Наш пример пока очень прост, но все же он демонстрирует некоторые важные моменты, касающиеся проектирования с помощью делегатов.

Использование типов делегатов, определенных на платформе .NET Core, упрощает работу с делегатами для пользователей. Вам не нужно определять новые типы, а разработчикам, использующим вашу библиотеку, не нужно изучать новые специальные типы делегатов.

Применяются минимально необходимые, но при этом максимально гибкие интерфейсы: чтобы создать компонент для вывода данных журнала, необходимо написать всего один метод. Это может быть статический метод или метод экземпляра. Он может иметь любой уровень доступа.

Форматирование вывода

Давайте немного улучшим первую версию, а затем приступим к созданию других механизмов ведения журнала.

Затем добавим в метод `LogMessage()` несколько аргументов, чтобы класс журнала создавал более

структурированные сообщения.

```
public enum Severity
{
    Verbose,
    Trace,
    Information,
    Warning,
    Error,
    Critical
}
```

```
public static class Logger
{
    public static Action<string> WriteMessage;

    public static void LogMessage(Severity s, string component, string msg)
    {
        var outputMsg = $"{DateTime.Now}\t{s}\t{component}\t{msg}";
        WriteMessage(outputMsg);
    }
}
```

Далее используем аргумент `Severity` для фильтрации сообщений, отправляемых в место вывода журнала.

```
public static class Logger
{
    public static Action<string> WriteMessage;

    public static Severity LogLevel {get;set;} = Severity.Warning;

    public static void LogMessage(Severity s, string component, string msg)
    {
        if (s < LogLevel)
            return;

        var outputMsg = $"{DateTime.Now}\t{s}\t{component}\t{msg}";
        WriteMessage(outputMsg);
    }
}
```

Рекомендации

Вы добавили новые функции в инфраструктуру ведения журналов. Так как компонент logger очень слабо связан с любым из механизмов вывода, эти функции можно добавлять без влияния на код, в котором реализуется делегат logger.

По мере расширения кода вы увидите дополнительные примеры того, как такая слабая взаимосвязь обеспечивает большую гибкость в плане обновления компонентов сайта без внесения изменений в другие его части. На самом деле в более крупном приложении классы вывода данных журнала могут находиться в другой сборке и даже не требовать повторной сборки.

Создание второго модуля вывода

Работа над компонентом журнала продвигается. Давайте добавим еще один модуль вывода, который записывает сообщения в файл. Для этого потребуется немного больше усилий. Это будет класс, который инкапсулирует файловые операции и обеспечивает закрытие файла после каждой операции записи. Благодаря этому все данные будут окончательно записываться на диск после создания каждого

сообщения.

Вот это средство ведения журнала на основе файла:

```
public class FileLogger
{
    private readonly string logPath;
    public FileLogger(string path)
    {
        logPath = path;
        Logger.WriteMessage += LogMessage;
    }

    public void DetachLog() => Logger.WriteMessage -= LogMessage;
    // make sure this can't throw.
    private void LogMessage(string msg)
    {
        try
        {
            using (var log = File.AppendText(logPath))
            {
                log.WriteLine(msg);
                log.Flush();
            }
        }
        catch (Exception)
        {
            // Hmm. We caught an exception while
            // logging. We can't really log the
            // problem (since it's the log that's failing).
            // So, while normally, catching an exception
            // and doing nothing isn't wise, it's really the
            // only reasonable option here.
        }
    }
}
```

После создания этого класса можно создать его экземпляр, и он подключит свой метод LogMessage к компоненту Logger:

```
var file = new FileLogger("log.txt");
```

Эти два метода не являются взаимоисключающими. Вы можете подключить оба метода ведения журнала, чтобы сообщения создавались как в консоли, так и в файле.

```
var fileOutput = new FileLogger("log.txt");
Logger.WriteMessage += LoggingMethods.LogToConsole; // LoggingMethods is the static class we utilized
earlier
```

Позднее вы можете удалить один из делегатов даже в том же приложении без каких-либо проблем для системы.

```
Logger.WriteMessage -= LoggingMethods.LogToConsole;
```

Рекомендации

Итак, вы добавили второй обработчик вывода для подсистемы ведения журнала. Ему требуется немного больший объем инфраструктуры для правильной поддержки файловой системы. Делегат представляет собой метод экземпляра. Кроме того, это закрытый метод. В более высоком уровне доступности нет

необходимости, так как инфраструктура делегатов может подключать делегаты.

Во-вторых, модель на основе делегатов предоставляет несколько методов вывода без дополнительного кода. Вам не нужно создавать дополнительную инфраструктуру для поддержки нескольких методов вывода. Они просто добавляются в список вызова.

Обратите особое внимание на код в методе вывода данных журнала в файл. Он написан так, что не создает никаких исключений. Хотя это не является строго обязательным, зачастую это более удобно. Если какой-либо из методов делегата создает исключение, остальные делегаты в списке вызова не будут вызваны.

Наконец, отметим, что компонент записи данных журнала в файл должен управлять своими ресурсами, открывая и закрывая файл для каждого сообщения журнала. Вы можете оставить файл открытым и реализовать интерфейс `IDisposable` для закрытия файла по завершении операции. У каждого метода свои плюсы и минусы. В обоих случаях несколько усиливается взаимозависимость между классами.

Для поддержки каждого из сценариев в код класса `Logger` не нужно вносить никаких изменений.

Обработка NULL-делегатов

Наконец, давайте изменим метод LogMessage так, чтобы он был эффективен в случаях, когда механизм вывода не выбран. Текущая реализация вызывает исключение `NullReferenceException`, когда к делегату `WriteMessage` не подключен список вызова. Однако предпочтительнее может быть автоматическое продолжение выполнения в случае, если методы не подключены. Такое поведение легко реализовать с помощью условного оператора null в сочетании с методом `Delegate.Invoke()`.

```
public static void LogMessage(string msg)
{
    WriteMessage?.Invoke(msg);
}
```

Условный оператор null (`?.`) замыкается, если левый операнд (в данном случае `WriteMessage`) имеет значение null, что означает, что попытки записать сообщение не предпринимаются.

Метод `Invoke()` не указан в документации по `System.Delegate` или `System.MulticastDelegate`. Компилятор создает типобезопасный метод `Invoke` для любого объявленного типа делегата. В этом примере это означает, что метод `Invoke` принимает один аргумент `string` и имеет тип возвращаемого значения void.

Сводка рекомендаций

Вы ознакомились с простейшим компонентом журнала, который можно расширять с помощью других модулей записи и иных функций. Благодаря использованию делегатов при проектировании эти разные компоненты оказываются слабо взаимосвязанными. Преимуществ несколько. Очень легко создавать новые механизмы вывода и подключать их к системе. Этим механизмам требуется только один метод, который записывает сообщение журнала. Такая модель устойчива при добавлении новых возможностей. Обязательным требованием для каждого модуля записи является реализация одного метода. Это может быть статический метод или метод экземпляра. Он может быть открытым, закрытым или иметь любой иной допустимый уровень доступа.

В класс Logger можно вносить любое количество улучшений и изменений без серьезной модификации. Открытый интерфейс API, как и любой другой класс, нельзя модифицировать без риска внесения существенных изменений. Но так как взаимосвязь между средством ведения журнала и модулями вывода осуществляется только посредством делегата, другие типы (например, интерфейсы или базовые классы) не затрагиваются. Взаимосвязь минимальна.

Вперед

Общие сведения о событиях

18.12.2021 • 2 minutes to read

Назад

События, так же как и делегаты, представляют собой механизм *позднего связывания*. На самом деле события основаны на тех же средствах языка, которые обеспечивают поддержку делегатов.

С помощью событий объект может сообщить всем компонентам системы, которым это необходимо, о том, что что-то произошло. Любой другой компонент может подписаться на событие, чтобы получать уведомления о его наступлении.

Возможно, вы уже пользовались событиями при программировании. Во многих графических системах есть модель событий, которая позволяет сообщать о действиях пользователей. Такие события сообщают о перемещениях мыши, нажатиях кнопок и иных подобных действиях. Это наиболее распространенный, но, безусловно, не единственный вариант использования событий.

Вы можете определить события, которые должны вызываться для классов. Важным моментом при работе с событиями является то, что для определенного события может быть не зарегистрирован ни один объект. Код необходимо писать так, чтобы он не вызывал событий, если прослушиватели не настроены.

При подписке на событие также создается взаимосвязь между двумя объектами (источником события и приемником событий). Если приемник событий больше не должен получать события, необходимо отменить его подписку на источник события.

Цели при проектировании поддержки событий

Ниже перечислены цели модели событий, реализуемой в языке.

- Между источником события и приемником событий должна быть минимальная взаимосвязь. Эти два компонента могут создаваться разными организациями и даже обновляться по совершенно разным графикам.
- Подписка на событие и отмена подписки на него должны производиться максимально просто.
- Источники событий должны поддерживать несколько подписчиков на события. Кроме того, должен поддерживаться сценарий, когда подписчики на события не подключены.

Как можно увидеть, цели в отношении событий очень похожи на цели в отношении делегатов. Вот почему языковая поддержка событий основана на механизмах поддержки делегатов.

Языковая поддержка событий

Синтаксис определения событий, а также подписки и отмены подписки на них является расширением синтаксиса для делегатов.

Для определения события используется ключевое слово `event`.

```
public event EventHandler<FileListArgs> Progress;
```

Тип события (в этом примере `EventHandler<FileListArgs>`) должен быть типом делегата. При объявлении события должен соблюдаться ряд соглашений. Как правило, тип делегата события имеет возвращаемый тип void. Объявление события должно представлять собой глагол или глагольное словосочетание. Если

событие сообщает о том, что уже произошло, используйте прошедшее время. Для сообщения о том, что должно произойти, используйте глагол в настоящем времени (например, `Closing`). Настоящее время часто указывает на то, что класс поддерживает какую-либо настройку. Одна из самых распространенных ситуаций — поддержка отмены. Например, событие `Closing` может иметь аргумент, который указывает на то, должна ли продолжаться операция закрытия. В других ситуациях вызывающим объектам может предоставляться возможность изменения поведения путем изменения свойств аргументов события. Событие может вызываться для указания действия, которое алгоритму предлагается выполнить в следующую очередь. Обработчик событий может предписать выполнение другого действия, изменив свойства аргумента события.

Если нужно инициировать событие, то следует вызвать соответствующий обработчик событий с помощью синтаксиса вызова делегатов:

```
Progress?.Invoke(this, new FileListArgs(file));
```

Как описано в разделе, посвященном делегатам, оператор ?. позволяет легко предотвратить попытки вызова события, если на него нет подписчиков.

Подписка на событие производится с помощью оператора `+=`:

```
EventHandler<FileListArgs> onProgress = (sender, eventArgs) =>
    Console.WriteLine(eventArgs.FoundFile);

fileLister.Progress += onProgress;
```

Имя метода обработчика обычно содержит имя события с префиксом On, как показано выше.

Для отмены подписки на событие служит оператор `-=`:

```
fileLister.Progress -= onProgress;
```

Важно, чтобы для выражения, представляющего обработчик событий, была объявлена локальная переменная. Благодаря этому при отмене подписки обработчик удаляется. Если вместо этого использовалось тело лямбда-выражения, то производится попытка удалить обработчик, который не был подключен, что не приводит ни к какому результату.

В следующей статье вы узнаете больше о типичных шаблонах событий и ознакомитесь с вариантами этого примера.

Вперед

Стандартные шаблоны событий .NET

18.12.2021 • 7 minutes to read

Назад

События .NET обычно следуют нескольким известным шаблонам. Стандартизация на основе этих шаблонов означает, что разработчики могут использовать знание таких стандартных шаблонов, применяя их к любой программе событий .NET.

Мы разберем эти стандартные шаблоны, чтобы снабдить вас знаниями, необходимыми для создания источников стандартных событий, подписки и обработки стандартных событий в коде.

Сигнатуры делегатов событий

Стандартной сигнатурой делегата события .NET является:

```
void OnEventRaised(object sender, EventArgs args);
```

Тип возвращаемого значения — void. События основаны на делегатах и являются делегатами многоадресной рассылки. Это обеспечивает поддержку нескольких подписчиков для любого источника событий. Одно значение, возвращаемое из метода, не масштабируется на несколько подписчиков событий. Какое возвращаемое значение доступно источнику события после возникновения события? Далее в этой статье будет показано, как создавать протоколы событий, которые поддерживают подписчики на события, передающие данные в источник событий.

Список аргументов содержит два аргумента: отправителя и аргументы события. Тип времени компиляции `sender` — `System.Object`, даже если вам, вероятно, известен более производный тип, который всегда является правильным. По соглашению используйте `object`.

Второй аргумент обычно являлся типом, производным от `System.EventArgs`. (В следующем разделе вы увидите, что это соглашение больше не является обязательным.) Даже если тип события не требует дополнительных аргументов, необходимо предоставить оба аргумента. Существует специальное значение `EventArgs.Empty`, которое следует использовать для обозначения того, что событие не содержит никаких дополнительных сведений.

Создадим класс, который перечисляет соответствующие шаблону файлы в каталоге или любом из его подкаталогов. Этот компонент создает событие для каждого найденного файла, который соответствует шаблону.

Использование модели событий обеспечивает некоторые преимущества разработки. Можно создать несколько прослушивателей событий, которые выполняют разные действия при нахождении искомого файла. Сочетание разных прослушивателей позволяет создавать более надежные алгоритмы.

Ниже показано объявление аргумента исходного события для поиска искомого файла:

```
public class FileFoundArgs : EventArgs
{
    public string FoundFile { get; }

    public FileFoundArgs(string fileName)
    {
        FoundFile = fileName;
    }
}
```

Несмотря на то, что этот тип выглядит как небольшой тип, содержащий только данные, вы должны выполнить соглашение и назначить его ссылочным типом (`class`). Это означает, что объект аргумента будет передаваться по ссылке, а любые обновления данных будут доступны всем подписчикам. Первая версия является неизменяемым объектом. Рекомендуется сделать свойства в типе аргумента события неизменяемыми. Таким образом, один подписчик не сможет изменить значения до того, как их увидит другой подписчик. (Существуют исключения, как можно будет увидеть ниже.)

Затем нужно создать объявление события в классе FileSearcher. Использование типа `EventHandler<T>` означает, что вам не требуется создавать еще одно определение типа. Вы просто используете универсальную специализацию.

Заполним класс FileSearcher для поиска файлов, соответствующих шаблону, и вызова правильного события при обнаружении совпадения.

```
public class FileSearcher
{
    public event EventHandler<FileFoundArgs> FileFound;

    public void Search(string directory, string searchPattern)
    {
        foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
        {
            FileFound?.Invoke(this, new FileFoundArgs(file));
        }
    }
}
```

Определение и вызов событий, подобных полям

Самый простой способ добавить событие в класс — объявить это событие как открытое поле, как показано в предыдущем примере.

```
public event EventHandler<FileFoundArgs> FileFound;
```

В таком коде объявляется открытое поле, что не рекомендуется в объектно-ориентированном программировании, поскольку необходимо обеспечить защиту доступа к данным с помощью свойств и методов. Хотя это выглядит нарушением рекомендаций, код, созданный компилятором, создает программы-оболочки, чтобы доступ к объектам событий мог осуществляться только безопасным образом. Единственные операции, доступные для событий, подобных полям, — обработчик add:

```
EventHandler<FileFoundArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    filesFound++;
};

fileLister.FileFound += onFileFound;
```

и обработчик remove:

```
fileLister.FileFound -= onFileFound;
```

Обратите внимание, что для обработчика используется локальная переменная. Если вы используете тело лямбда-выражения, удаление не будет работать корректно. Будет существовать другой экземпляр делегата, не выполняющий никаких действий.

Код вне класса не может вызывать события, а также выполнять другие операции.

Получение возвращаемых значений от подписчиков на событие

Простая версия работает нормально. Давайте добавим еще одну возможность — отмену.

При вызове события нахождения прослушиватели должны иметь возможность остановить дальнейшую обработку, если этот файл является последним искомым.

Обработчики событий не возвращают значение, поэтому вам нужно выполнить это другим способом. Стандартный шаблон события использует объект EventArgs для включения полей, которые подписчики на события могут использовать для передачи сообщения об отмене.

Для этого случая предусмотрено два разных шаблона, которые можно использовать в зависимости от семантики контракта "Отмена". В обоих случаях в EventArguments добавляется логическое поле для события найденного файла.

Один шаблон позволяет любому одному подписчику отменить операцию. Для этого шаблона новое поле инициализируется значением `false`. Любой подписчик можно изменить его на `true`. После того как все подписчики увидят событие, компонент FileSearcher проверяет логическое значение и выполняет действие.

Второй шаблон отменяет операцию, только если все подписчики подтвердили ее отмену. В этом шаблоне новое поле инициализируется для указания того, что операцию следует отменить, и любой подписчик может изменить его, чтобы указать, что следует продолжить операцию. После того как все подписчики увидят событие, компонент FileSearcher проверяет логическое значение и выполняет действие. В этом шаблоне есть еще один дополнительный шаг: компонент должен знать, все ли подписчики видели событие. Если подписчики отсутствуют, поле неверно сообщит об отмене.

Реализуем первую версию для этого примера. Добавьте логическое поле с именем `CancelRequested` в тип `FileFoundArgs`:

```
public class FileFoundArgs : EventArgs
{
    public string FoundFile { get; }
    public bool CancelRequested { get; set;}

    public FileFoundArgs(string fileName)
    {
        FoundFile = fileName;
    }
}
```

Это новое поле автоматически инициализируется значением `false`, которое используется по умолчанию для логических полей, чтобы не происходило случайной отмены. Единственным другим изменением в компоненте является установка флага после вызова события для просмотра, если любой из подписчиков запросил отмену:

```
public void List(string directory, string searchPattern)
{
    foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
    {
        var args = new FileFoundArgs(file);
        FileFound?.Invoke(this, args);
        if (args.CancelRequested)
            break;
    }
}
```

Одно из преимуществ этого шаблона заключается в том, что это не является критическим изменением. Ни один из подписчиков не запрашивал отмену ранее и не запрашивает в настоящий момент. Код подписчиков не требует обновления, если не требуется поддержка нового протокола отмены. Они очень слабо связаны.

Изменим подписчик, чтобы он запрашивал отмену, когда обнаруживает первый исполняемый файл:

```
EventHandler<FileFoundArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    eventArgs.CancelRequested = true;
};
```

Добавление другого объявления события

Добавим еще одну возможность и продемонстрируем другие выражения языка для событий. Добавим перегрузку метода `Search`, который проходит через все подкаталоги в поиске файлов.

Эта операция может выполняться длительное время в каталоге с большим числом вложенных каталогов. Добавим событие, которое вызывается в начале каждого нового поиска в каталоге. Это позволяет подписчикам отслеживать ход выполнения и сообщать о нем пользователю. Все примеры, которые мы создали до сих пор, являются открытыми. Сделаем это событие внутренним. Это означает, что типы, используемые для аргументов, также можно сделать внутренними.

Вы начнете с создания нового производного класса EventArgs для передачи сведений о новом каталоге и ходе выполнения.

```
internal class SearchDirectoryArgs : EventArgs
{
    internal string CurrentSearchDirectory { get; }
    internal int TotalDirs { get; }
    internal int CompletedDirs { get; }

    internal SearchDirectoryArgs(string dir, int totalDirs, int completedDirs)
    {
        CurrentSearchDirectory = dir;
        TotalDirs = totalDirs;
        CompletedDirs = completedDirs;
    }
}
```

Опять же, вы можете следовать рекомендациям по созданию неизменяемого ссылочного типа для аргументов событий.

Теперь определим событие. На этот раз будет использоваться другой синтаксис. Помимо синтаксиса полей можно явно создать свойство с помощью обработчиков add и remove. В этом примере вы не будете добавлять код в эти обработчики, здесь просто демонстрируется их создание.

```
internal event EventHandler<SearchDirectoryArgs> DirectoryChanged
{
    add { directoryChanged += value; }
    remove { directoryChanged -= value; }
}
private EventHandler<SearchDirectoryArgs> directoryChanged;
```

Созданный здесь код очень похож на тот код, который компилятор создает для определения полей событий, как было показано ранее. Для создания события используется синтаксис, очень похожий на используемый для свойств. Обратите внимание, что обработчики имеют разные имена: `add` и `remove`. Они вызываются для подписки на событие или отмены подписки на событие. Учтите, что вы также должны объявить закрытое резервное поле для хранения переменной событий. Оно инициализируется значением NULL.

Теперь добавим перегрузку метода `Search`, который обходит подкаталоги и вызывает оба события. Для этого проще всего использовать аргумент по умолчанию для задания поиска по всем каталогам:

```
public void Search(string directory, string searchPattern, bool searchSubDirs = false)
{
    if (searchSubDirs)
    {
        var allDirectories = Directory.GetDirectories(directory, "*.*", SearchOption.AllDirectories);
        var completedDirs = 0;
        var totalDirs = allDirectories.Length + 1;
        foreach (var dir in allDirectories)
        {
            directoryChanged?.Invoke(this,
                new SearchDirectoryArgs(dir, totalDirs, completedDirs++));
            // Search 'dir' and its subdirectories for files that match the search pattern:
            SearchDirectory(dir, searchPattern);
        }
        // Include the Current Directory:
        directoryChanged?.Invoke(this,
            new SearchDirectoryArgs(directory, totalDirs, completedDirs++));
        SearchDirectory(directory, searchPattern);
    }
    else
    {
        SearchDirectory(directory, searchPattern);
    }
}

private void SearchDirectory(string directory, string searchPattern)
{
    foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
    {
        var args = new FileFoundArgs(file);
        FileFound?.Invoke(this, args);
        if (args.CancelRequested)
            break;
    }
}
```

На этом этапе можно запустить приложение, вызывающее перегруженный метод для поиска всех вложенных каталогов. Для нового события `ChangeDirectory` нет подписчиков, однако благодаря использованию идиомы `?.Invoke()` мы можем гарантировать правильную работу метода.

Добавим обработчик для написания строки, показывающей ход выполнения в окне консоли.

```
fileLister.DirectoryChanged += (sender, eventArgs) =>
{
    Console.Write($"Entering '{eventArgs.CurrentSearchDirectory}'.");
    Console.WriteLine($" {eventArgs.CompletedDirs} of {eventArgs.TotalDirs} completed...");
};
```

Мы познакомились с шаблонами, которые используются во всей экосистеме .NET. Научившись использовать эти шаблоны и соглашения, вы сможете быстро создавать код C# и .NET на основе идиом.

Далее мы рассмотрим некоторые изменения в этих шаблонах в самой последней версии .NET.

Вперед

Обновленный шаблон событий .NET Core

18.12.2021 • 3 minutes to read

Назад

В предыдущей статье рассматривались наиболее распространенные шаблоны событий. .NET Core имеет менее жесткий шаблон. В этой версии определение `EventHandler<TEventArgs>` больше не имеет ограничения, указывающего на то, что `TEventArgs` должен быть классом, производным от `System.EventArgs`.

В результате повышается гибкость разработки и обеспечивается обратная совместимость. Начнем с гибких возможностей. Класс System.EventArgs представляет один метод: `MemberwiseClone()`, который создает неполную копию объекта. Чтобы реализовать свою функциональность для любого класса, производного от `EventArgs`, этот метод должен использовать отражение. Функциональность проще создать в определенном производном классе. Это фактически означает, что наследование от System.EventArgs является ограничением, которое регламентирует разработки, но не предоставляет никаких дополнительных преимуществ. На самом деле, можно изменить определения `FileFoundArgs` и `SearchDirectoryArgs` так, чтобы они не были производными от `EventArgs`. Программа будет работать точно так же.

Вместо `SearchDirectoryArgs` можно также использовать структуру, внеся дополнительное изменение.

```
internal struct SearchDirectoryArgs
{
    internal string CurrentSearchDirectory { get; }
    internal int TotalDirs { get; }
    internal int CompletedDirs { get; }

    internal SearchDirectoryArgs(string dir, int totalDirs, int completedDirs) : this()
    {
        CurrentSearchDirectory = dir;
        TotalDirs = totalDirs;
        CompletedDirs = completedDirs;
    }
}
```

Дополнительное изменение заключается в вызове конструктора без параметров перед входом в конструктор, который инициализирует все поля. Без этого согласно правилам языка C# будет выведено сообщение о получении доступа к свойствам до их назначения.

Не следует изменять `FileFoundArgs` с класса (ссылочный тип) на структуру (тип значения). Это связано с тем, что протокол для обработки отмены требует передачи аргументов события по ссылке. Если вы выполнили то же изменение, класс поиска файла никогда не сможет отслеживать изменения, внесенные подписчиками событий. Для каждого подписчика будет использоваться новая копия, которая будет отличаться от той, которую обнаружил объект поиска файла.

Теперь давайте рассмотрим обратную совместимость этого изменения. Удаление ограничения не влияет на существующий код. Все имеющиеся типы аргументов событий по-прежнему являются производными от `System.EventArgs`. Обратная совместимость является одной из основных причин, по которой они будут и далее являться производными от `System.EventArgs`. Все существующие подписчики на события будут подписчиками на событие, следующее классическому шаблону.

Согласно аналогичной логике, теперь тип события аргумента не будет иметь подписчиков в

существующих базах кода. Новые типы событий, которые не являются производными от `System.EventArgs`, не нарушат функциональность этих баз кода.

События с асинхронными подписчиками

Вам осталось изучить последний шаблон: как правильно написать подписчики событий, которые вызывают асинхронный код. Это описано в статье, посвященной async и await. Асинхронные методы могут иметь возвращаемый тип void, однако это крайне не рекомендуется. Когда код подписчика событий вызывает асинхронный метод, вам придется создать метод `async void`. Это необходимо для сигнатуры обработчика событий.

Вы должны найти правильное решение. Каким-то образом необходимо создать безопасный метод `async void`. Ниже приведены основные сведения о шаблоне, которым необходимо следовать.

```
worker.StartWorking += async (sender, eventArgs) =>
{
    try
    {
        await DoWorkAsync();
    }
    catch (Exception e)
    {
        //Some form of logging.
        Console.WriteLine($"Async task failure: {e.ToString()}");
        // Consider gracefully, and quickly exiting.
    }
};
```

Во-первых, обратите внимание на то, что обработчик помечен как асинхронный обработчик. Поскольку он назначается типу делегата обработчика событий, он будет иметь возвращаемый тип void. Это означает, что необходимо следовать шаблону, приведенному в обработчике, и не допускать создания исключений вне контекста асинхронного обработчика. Поскольку он не возвращает задачу, ничто не сообщает об ошибке при переходе в состояние сбоя. Поскольку метод является асинхронным, он просто не может создать исключение. (Вызывающий метод продолжал выполнение, так как это `async`.) Фактическое поведение во время выполнения будет определяться по-разному для разных сред. Оно может завершить поток или процесс, владеющий потоком, или оставить процесс в неопределенном состоянии. Все эти возможные результаты очень нежелательны.

Вот почему необходимо заключить оператор await для асинхронных задач в собственный блок try. Если он приводит к сбою задачи, можно записать ошибку в журнал. Если это ошибка, из-за которой невозможно восстановить приложение, можно быстро и правильно выйти из программы.

Это были основные обновления шаблона событий .NET. Затем вы увидите множество примеров предыдущих версий в библиотеках, с которыми вы работаете. Однако также необходимо иметь представление и о последних шаблонах.

В следующей статье этой серии материалов вы узнаете об использовании `delegates` и `events` в своих проектах. Это относительно схожие понятия, и сведения в этой статье помогут вам принять оптимальное решение.

Вперед

Различия между делегатами и событиями

18.12.2021 • 2 minutes to read

Назад

Разработчики, не имеющие опыта работы с платформой .NET Core, часто не могут решить, что следует выбрать: структуру на основе `delegates` или на основе `events`. Часто выбор делегатов или событий довольно сложен, так как эти две возможности языка довольно похожи. Более того, события основаны на тех же средствах языка, которые обеспечивают поддержку делегатов.

И те и другие обеспечивают сценарии позднего связывания, в которых взаимодействие компонента осуществляется путем вызова метода, известного только во время выполнения. И те и другие поддерживают методы с одним или несколькими подписчиками. Иногда это называют поддержкой одноадресности и многоадресности. Синтаксис добавления и удаления обработчиков в обоих случаях похож. Наконец, при вызове событий и делегатов используется абсолютно одинаковый синтаксис вызова методов. Более того, поддерживается одинаковый синтаксис метода `Invoke()` для использования с оператором `?.`.

Учитывая такое сходство, легко могут возникнуть проблемы с выбором подходящего механизма в той или иной ситуации.

Прослушивание событий необязательно

Самым важным фактором при выборе подходящего механизма является обязательность наличия подключенного подписчика. Если ваш код должен вызывать код, предоставленный подписчиком, следует использовать структуру на основе делегатов, когда вам нужно реализовать обратный вызов. Если код может выполнить все задачи, не вызывая подписчики, следует использовать структуру на основе событий.

Обратите внимание на примеры, рассматривавшиеся в этом разделе. Код, который вы создавали с помощью `List.Sort()`, должен содержать функцию сравнения для правильной сортировки элементов. Запросам LINQ необходимо предоставить делегаты для определения элементов, которые следует вернуть. В обоих случаях использовалась структура на основе делегатов.

Рассмотрим событие `Progress`. Оно сообщает о ходе выполнения задачи. Задача продолжает выполняться вне зависимости от того, есть ли прослушиватели. Еще один пример — `FileSearcher`. Нахождение всех искомых файлов производится, даже если не подключены обработчики событий. Элементы управления UX продолжают работать правильно, даже если нет подписчиков, прослушивающих события. В обоих примерах используются структуры на основе событий.

Для возвращаемых значений требуются делегаты

Еще одним аспектом является прототип метода, который требуется для метода делегата. Как вы уже видели, все делегаты, используемые для событий, имеют тип возвращаемого значения void. Вы уже также знаете, что есть идиомы для создания обработчиков событий, которые передают информацию обратно источникам событий, изменяя свойства объекта аргумента события. Хотя такие идиомы работают, они не так естественны, как возвращение значения из метода.

Обратите внимание, что часто присутствуют оба исследуемых аспекта. Если метод делегата возвращает значение, скорее всего, это повлияет на алгоритм.

Для событий используется закрытый вызов

Классы, отличные от тех, в которых содержится событие, могут только добавлять и удалять прослушиватели событий. Вызывать событие может только класс, содержащий событие. События обычно являются открытыми членами класса. Для сравнения, делегаты часто передаются как параметры и хранятся как закрытые члены класса (если хранятся вообще).

Прослушиватели событий часто имеют более длительный срок существования

То, что прослушиватели событий имеют более продолжительное время существования, — не столь важный фактор. Однако структура на основе событий может оказаться более естественной, когда источник событий инициирует события в течение длительного периода времени. Примеры структуры на основе событий для элементов управления UX можно найти во многих системах. После подписания на событие источник события может вызывать события в течение всего времени существования программы. (Когда события больше не нужны, вы можете отменить подписку на них.)

Сравните это с часто встречающимися структурами на основе делегатов, когда делегат применяется в качестве аргумента метода и не используется после того, как метод возвращает управление.

Тщательно оценивайте ситуацию

Приведенные выше указания не являются строгими правилами. Они представляют собой лишь рекомендации, которые могут помочь вам выбрать правильный вариант в конкретном случае. Так как эти механизмы схожи, вы можете смоделировать и тот и другой, чтобы проверить, какой из них окажется естественнее. Оба они хорошо подходят для сценариев с поздним связыванием. Используйте тот механизм, который лучше соответствует вашим требованиям.

Синтаксис LINQ

18.12.2021 • 3 minutes to read

Аббревиатура LINQ обозначает целый набор технологий, создающих и использующих возможности интеграции запросов непосредственно в язык C#. Традиционно запросы к данным выражаются в виде простых строк без проверки типов при компиляции или поддержки IntelliSense. Кроме того, разработчику приходится изучать различные языки запросов для каждого типа источников данных: баз данных SQL, XML-документов, различных веб-служб и т. д. Технологии LINQ превращают запросы в удобную языковую конструкцию, которая применяется аналогично классам, методам и событиям.

Для разработчика, который создает запросы, наиболее очевидной частью LINQ является интегрированное выражение запроса. Выражения запроса используют декларативный *синтаксис запроса*. С помощью синтаксиса запроса можно выполнять фильтрацию, упорядочение и группирование данных из источника данных, обходясь минимальным объемом программного кода. Одни и те же базовые выражения запроса позволяют одинаково легко получать и преобразовывать данные из баз данных SQL, наборов данных ADO .NET, XML-документов, XML-потоков и коллекций .NET.

В следующем примере показан полный пример использования запроса. Полная операция сначала создает источник данных, затем определяет выражение запроса и выполняет этот запрос в инструкции `foreach`.

```
class LINQQueryExpressions
{
    static void Main()
    {

        // Specify the data source.
        int[] scores = new int[] { 97, 92, 81, 60 };

        // Define the query expression.
        IEnumerable<int> scoreQuery =
            from score in scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in scoreQuery)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 97 92 81
```

Описание выражения запроса

- Выражение запроса можно использовать для получения и преобразования данных из любого источника данных, поддерживающего LINQ. Например, можно одним запросом получить данные из базы данных SQL и создать на их основе выходной XML-поток.
- Выражение запроса очень легко понять, поскольку в нем используется много знакомых конструкций языка C#.
- Все переменные в выражениях запросов строго типизированы, хотя во многих случаях вам не нужно указывать тип явным образом, поскольку компилятор определит его автоматически.

Дополнительные сведения см. в статье Отношения между типами в операциях запросов LINQ (C#).

- Запрос не будет выполняться, пока вы не начнете обращаться к переменной запроса, например, с помощью инструкции `foreach`. Дополнительные сведения см. в статье Введение в запросы LINQ (C#).
- Во время компиляции выражения запроса преобразуются в вызовы метода стандартного оператора запроса. Для преобразования используются правила, заданные в спецификации C#. Любой запрос, который может быть выражен с помощью синтаксиса запросов, также может быть выражен с помощью синтаксиса методов. Однако в большинстве случаев синтаксис запроса будет понятнее и лаконичнее. Дополнительные сведения см. в статьях C# language specification (Спецификация языка C#) и Общие сведения о стандартных операторах запроса (C#).
- Мы рекомендуем при написании запросов LINQ использовать синтаксис запросов везде, где это возможно, а синтаксис метода — только если это совершенно необходимо. Между этими формами синтаксиса нет никакой разницы в семантике или производительности. Выражения запросов обычно более удобочитаемыми, чем аналогичные выражения с использованием синтаксиса метода.
- Некоторых операции запросов, например Count или Max, не имеют эквивалентных предложений выражения для запросов и должны выражаться как вызовы методов. Синтаксис запросов и синтаксис методов можно сочетать друг с другом различными способами. Дополнительные сведения см. в статье Синтаксис запросов и синтаксис методов в LINQ (C#).
- Выражения запросов могут компилироваться в деревья выражений или в делегаты, в зависимости от типа, к которому применяется конкретный запрос. Запросы IEnumerable<T> компилируются в делегаты. Запросы IQueryable и IQueryable<T> компилируются в деревья выражений. Дополнительные сведения см. в статье Деревья выражений.

Следующие шаги

Чтобы получить дополнительные сведения о LINQ, сначала ознакомьтесь с некоторыми основным понятиями в статье Основы выражения запроса, а затем переходите к документации по интересующей вас технологии LINQ.

- XML-документы: LINQ to XML
- Платформа ADO.NET Entity Framework: LINQ to Entities
- Коллекции, файлы, строки и другие сущности .NET: LINQ to Objects (C#)

Чтобы глубже разобраться в базовой концепции LINQ изучите статью о LINQ в C#.

Чтобы быстрее приступить к работе с LINQ в C#, переходите к руководству Работа с LINQ.

Основы выражения запроса

18.12.2021 • 10 minutes to read

В этой статье представлены основные понятия, связанные с выражениями запроса на языке C#.

Что такое запрос и для чего он нужен

Запрос — это набор инструкций, которые описывают, какие данные необходимо извлечь из указанного источника (или источников) данных, а также описывают форму и организацию извлекаемых данных. Запрос отличается от полученного с его помощью результата.

Обычно исходные данные логически организованы как последовательность элементов одного вида. Например, таблица базы данных SQL содержит последовательность строк. В файле XML содержится "последовательность" элементов XML (они организованы иерархически в древовидную структуру). Коллекция в памяти содержит последовательность объектов.

С точки зрения приложения определенные тип и структура оригинальных исходных данных не важны. Приложение всегда видит исходные данные в виде коллекции IEnumerable<T> или IQueryable<T>. Например, в LINQ to XML исходные данные становятся видимыми как `IEnumerable` <XElement>.

При такой исходной последовательности, запрос может выполнять одно из трех возможных действий.

- Извлечение подмножества элементов для получения новой последовательности без изменения отдельных элементов. Затем запрос может отсортировать или сгруппировать возвращаемую последовательность различными способами, как показано в следующем примере (предположим, что `scores` является `int[]`):

  ```
  IEnumerable<int> highScoresQuery =
      from score in scores
      where score > 80
      orderby score descending
      select score;
  ```

- Извлечение последовательности элементов, как и в предыдущем примере, но с преобразованием элементов в новый тип объекта. Например, запрос может извлекать только фамилии из определенных записей клиентов в источнике данных. Запрос также может извлекать полную запись и использовать ее для создания другого типа объекта в памяти или даже данных XML перед созданием заключительной последовательности результатов. В следующем примере показана трансформация `int` в `string`. Обратите внимание на новый тип `highScoresQuery`.

  ```
  IEnumerable<string> highScoresQuery2 =
      from score in scores
      where score > 80
      orderby score descending
      select $"The score is {score}";
  ```

- Извлечение одноэлементного значения исходных данных, таких как:

  - Количество элементов, соответствующих определенному условию.
  - Элемент с наибольшим или наименьшим значением.
  - Первый элемент, соответствующий условию, или сумма определенных значений в указанном

наборе элементов. Например, следующий запрос возвращает количество оценок выше 80 из целочисленного массива `scores`:

```
int highScoreCount =
    (from score in scores
     where score > 80
     select score)
     .Count();
```

В предыдущем примере обратите внимание на использование скобок вокруг выражения запроса перед вызовом метода `Count`. Его также можно выразить, используя новую переменную для сохранения конкретного результата. Этот метод является более удобочитаемым, так как переменная, в которой хранится запрос, хранится отдельно от запроса, в котором хранится результат.

```
IEnumerable<int> highScoresQuery3 =
    from score in scores
    where score > 80
    select score;

int scoreCount = highScoresQuery3.Count();
```

В предыдущем примере запрос выполняется в вызове `Count`, так как `Count` должен выполнить итерацию результатов, чтобы определить количество элементов, возвращенных методом `highScoresQuery`.

Что такое выражение запроса

Выражение запроса — запрос, выраженный с помощью синтаксиса запроса. Выражение запроса является конструкцией языка первого класса. Оно похоже на любое другое выражение и может использоваться в любом контексте, в котором выражение C# является допустимым. Выражение запроса состоит из набора предложений, написанных в декларативном синтаксисе, аналогичном SQL или XQuery. Каждое предложение, в свою очередь, содержит одно или несколько выражений C#, которые могут являться выражениями запроса или содержать выражение запроса.

Выражение запроса должно начинаться предложением from и заканчиваться предложением select или group. Между первым предложением `from` и последним предложением `select` или `group` может содержаться одно или несколько необязательных предложений: where, orderby, join, let и даже дополнительных предложений from. Можно также использовать ключевое слово into, чтобы результат предложения `join` или `group` мог служить источником дополнительных предложений запроса в том же выражении запроса.

Переменная запроса

В LINQ переменная запроса — это любая переменная, сохраняющая *запрос* вместо *результатов* запроса. Говоря точнее, переменная запроса всегда является перечислимым типом и производит последовательность элементов, когда она используется в итерации оператора `foreach` или прямом вызове ее метода `IEnumerator.MoveNext`.

В следующем примере кода показано простое выражение запроса с одним источником данных, одним предложением фильтрации, одним предложением упорядочения и без трансформации исходных элементов. Предложение `select` завершает запрос.

```
static void Main()
{
    // Data source.
    int[] scores = { 90, 71, 82, 93, 75, 82 };

    // Query Expression.
    IEnumerable<int> scoreQuery = //query variable
        from score in scores //required
        where score > 80 // optional
        orderby score descending // optional
        select score; //must end with select or group

    // Execute the query to produce the results
    foreach (int testScore in scoreQuery)
    {
        Console.WriteLine(testScore);
    }
}
// Outputs: 93 90 82 82
```

В предыдущем примере `scoreQuery` — *переменная запроса*, которую иногда называют просто *запросом*. В переменной запроса не хранятся фактические данные результата, которые получаются с помощью цикла `foreach`. Когда выполняется оператор `foreach`, результаты запроса не возвращаются с помощью переменной запроса `scoreQuery`. В этом случае они возвращаются с помощью переменной итерации `testScore`. Итерация переменной `scoreQuery` может выполняться во втором цикле `foreach`. Результаты будет теми же, если ни они, ни источник данных не изменяются.

В переменной запроса может храниться запрос, выраженный с помощью синтаксиса запроса или метода запроса, или их комбинации. В следующих примерах `queryMajorCities` и `queryMajorCities2` являются переменными запроса.

```
//Query syntax
IEnumerable<City> queryMajorCities =
    from city in cities
    where city.Population > 100000
    select city;

// Method-based syntax
IEnumerable<City> queryMajorCities2 = cities.Where(c => c.Population > 100000);
```

С другой стороны, в следующих примерах показаны переменные, которые не являются переменными запроса даже несмотря на то, что все они инициализируются запросом. Они не являются переменными запроса, так как в них хранятся результаты.

```
int highestScore =
    (from score in scores
     select score)
    .Max();

// or split the expression
IEnumerable<int> scoreQuery =
    from score in scores
    select score;

int highScore = scoreQuery.Max();
// the following returns the same result
int highScore = scores.Max();

List<City> largeCitiesList =
    (from country in countries
     from city in country.Cities
     where city.Population > 10000
     select city)
       .ToList();

// or split the expression
IEnumerable<City> largeCitiesQuery =
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city;

List<City> largeCitiesList2 = largeCitiesQuery.ToList();
```

Дополнительные сведения о различных способах выражения запросов см. в разделе Синтаксис запросов и синтаксис методов в LINQ.

Явная и неявная типизация переменных запроса

В этой документации обычно явно указывается тип переменной запроса для того, чтобы продемонстрировать типичное отношение между переменной запроса и предложением select. Однако можно также использовать ключевое слов var, чтобы указать компилятору вывести тип переменной запроса (или любой другой локальной переменной) во время компиляции. Например, ранее приведенный в данном разделе пример запроса также может быть выражен путем неявной типизации:

```
// Use of var is optional here and in all queries.
// queryCities is an IEnumerable<City> just as
// when it is explicitly typed.
var queryCities =
    from city in cities
    where city.Population > 100000
    select city;
```

Дополнительные сведения см. в разделах Неявно типизированные локальные переменные и Связи типов в операциях запроса LINQ.

Начало выражения запроса

Выражение запроса должно начинаться с предложения `from`. Оно задает источник данных вместе с переменной диапазона. Переменная диапазона предоставляет каждый последующий элемент в исходной последовательности во время ее обзора. Переменная диапазона строго типизируется на основе типа элементов в источнике данных. В следующем примере переменная диапазона типизируется как `countries`, так как `Country` является массивом объектов `Country`. Так как переменная диапазона строго типизируется, для доступа к любым доступным элементам типа можно использовать оператор-точку.

```
IEnumerable<Country> countryAreaQuery =
    from country in countries
    where country.Area > 500000 //sq km
    select country;
```

Переменная диапазона находится в области до тех пор, пока запрос не завершится с помощью точки с запятой или предложения continuation.

Выражение запроса может содержать несколько предложений `from`. Используйте дополнительные предложения `from`, если каждый элемент в исходной последовательности является коллекцией или содержит коллекцию. Например, предположим, что имеется коллекция объектов `Country`, каждый из которых содержит коллекцию объектов `City` с именем `Cities`. Для выполнения запросов к объектам `City` в каждой коллекции `Country` используйте два предложения `from`, как показано ниже.

```
IEnumerable<City> cityQuery =
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city;
```

Дополнительные сведения см. в разделе Предложение from.

Окончание выражения запроса

Выражение запроса должно завершаться предложением `group` или `select`.

Предложение group

Используйте предложение `group` для получения последовательности групп, упорядоченных по указанному ключу. Ключом могут быть данные любого типа. Например, следующий запрос создает последовательность групп, содержащую один или несколько объектов `Country`, ключ для которых имеет тип `char` со значением, соответствующим первым буквам названий стран.

```
var queryCountryGroups =
    from country in countries
    group country by country.Name[0];
```

Дополнительные сведения о группировании см. в разделе Предложение group.

Предложение select

Используйте предложение `select` для получения всех других типов последовательностей. Простое предложение `select` просто создает последовательность с тем же типом объектов, что и у объектов, которые содержатся в источнике данных. В этом примере источник данных содержит объекты типа `Country`. Предложение `orderby` просто сортирует элементы в новом порядке, а предложение `select` создает последовательность переупорядоченных объектов `Country`.

```
IEnumerable<Country> sortedQuery =
    from country in countries
    orderby country.Area
    select country;
```

Предложение `select` может использоваться для преобразования исходных данных в последовательности новых типов. Такое преобразование также называется *проекцией*. В следующем примере предложение `select` создает *проекцию* последовательности анонимных типов, которая содержит только подмножество полей оригинального элемента. Обратите внимание, что новые объекты инициализируются с помощью инициализатора объекта.

```
// Here var is required because the query
// produces an anonymous type.
var queryNameAndPop =
    from country in countries
    select new { Name = country.Name, Pop = country.Population };
```

Дополнительные сведения обо всех методах использования предложения `select` для преобразования исходных данных см. в разделе Предложение select.

Продолжения с использованием ключевого слова *into*

Ключевое слово `into` можно использовать в предложении `select` или `group` для создания временного идентификатора, в котором хранится запрос. Это действие рекомендуется выполнять, если требуется выполнить в запросе дополнительные операции запроса после операции группирования или выбора. В следующем примере объекты `countries` группируются в соответствии с численностью населения в диапазоны по 10 миллионов. После создания этих групп дополнительные предложения отфильтровывают некоторые группы, а затем сортируют группы в порядке возрастания. Чтобы выполнить эти дополнительные операции, требуется продолжение, предоставляемое с помощью `countryGroup`.

```
// percentileQuery is an IEnumerable<IGrouping<int, Country>>
var percentileQuery =
    from country in countries
    let percentile = (int) country.Population / 10_000_000
    group country by percentile into countryGroup
    where countryGroup.Key >= 20
    orderby countryGroup.Key
    select countryGroup;

// grouping is an IGrouping<int, Country>
foreach (var grouping in percentileQuery)
{
    Console.WriteLine(grouping.Key);
    foreach (var country in grouping)
        Console.WriteLine(country.Name + ":" + country.Population);
}
```

Дополнительные сведения см. в разделе into.

Фильтрация, упорядочение и присоединение

Между открывающим предложением `from` и завершающим предложением `select` или `group` могут размещаться все остальные необязательные предложения (`where` , `join` , `orderby` , `from` , `let`). Любое необязательное предложение может использоваться в теле запроса несколько раз или отсутствовать вообще.

Предложение where

Используйте предложение `where` для фильтрации элементов из источника данных по одному или нескольким выражениям предиката. Предложение `where` в следующем примере имеет один предикат с двумя условиями.

```
IEnumerable<City> queryCityPop =
    from city in cities
    where city.Population < 200000 && city.Population > 100000
    select city;
```

Дополнительные сведения см. в разделе Предложение where.

Предложение orderby

Используйте `orderby` предложение для сортировки результатов по возрастанию или убыванию. Также можно задать порядок дополнительной сортировки. В следующем примере выполняется основная сортировка объектов `country` по свойству `Area`. Затем выполняется дополнительная сортировка по свойству `Population`.

```
IEnumerable<Country> querySortedCountries =
    from country in countries
    orderby country.Area, country.Population descending
    select country;
```

Ключевое слово `ascending` является необязательным, так как сортировка по умолчанию происходит по возрастанию, если не задан порядок сортировки. Дополнительные сведения см. в разделе Предложение orderby.

Предложение join

Используйте предложение `join` для связи или объединения элементов из одного источника данных с элементами из другого источника данных на основе сравнения на равенство определенных ключей в каждом элементе. В LINQ операции объединения выполняются в последовательностях объектов, элементы которых относятся к разным типам. После объединения двух последовательностей необходимо использовать оператор `select` или `group`, чтобы указать элемент для сохранения в выходной последовательности. Также можно использовать анонимный тип, чтобы объединить свойства каждого набора связанных элементов в новый тип для выходной последовательности. В следующем примере связываются объекты `prod`, свойство `Category` которых соответствует одной из категорий в массиве строк `categories`. Продукты, `Category` для которых не соответствует ни одной строке в `categories`, отфильтровываются. Инструкция `select` проецирует новый тип, свойства которого берутся из `cat` и `prod`.

```
var categoryQuery =
    from cat in categories
    join prod in products on cat equals prod.Category
    select new { Category = cat, Name = prod.Name };
```

Также можно выполнить групповое соединение путем сохранения результатов операции `join` во временную переменную, используя ключевое слово into. Дополнительные сведения см. в разделе Предложение join.

Предложение let

Используйте предложение `let` для сохранения результата выражения, например вызов метода, в новую переменную диапазона. В следующем примере в переменную диапазона `firstName` сохраняется первый элемент массива строк, возвращенного с помощью `Split`.

```
string[] names = { "Svetlana Omelchenko", "Claire O'Donnell", "Sven Mortensen", "Cesar Garcia" };
IEnumerable<string> queryFirstNames =
    from name in names
    let firstName = name.Split(' ')[0]
    select firstName;

foreach (string s in queryFirstNames)
    Console.Write(s + " ");
//Output: Svetlana Claire Sven Cesar
```

Дополнительные сведения см. в разделе Предложение let.

Вложенные запросы в выражении запроса

Предложение запроса может само содержать выражение запроса, которое иногда называют *вложенным*

запросом. Каждый вложенный запрос начинается с собственным предложением `from`, которое может указывать на источник данных, отличный от источника данных первого предложения `from`. Например, в следующем запросе показано выражение запроса, которое используется в операторе "select" для извлечения результатов операции группирования.

```
var queryGroupMax =
    from student in students
    group student by student.GradeLevel into studentGroup
    select new
    {
        Level = studentGroup.Key,
        HighestScore =
            (from student2 in studentGroup
             select student2.Scores.Average())
             .Max()
    };
```

Дополнительные сведения см. в руководстве по выполнению вложенного запроса в операции группирования.

См. также

- Руководство по программированию на C#
- LINQ
- Ключевые слова запросов (LINQ)
- Общие сведения о стандартных операторах запросов

LINQ в C#

18.12.2021 • 2 minutes to read

Этот раздел включает ссылки на разделы, содержащие более подробные сведения о LINQ.

Содержание раздела

Введение в запросы LINQ

Описание трех составляющих основной операции запроса LINQ, которые являются общими для всех языков и источников данных.

LINQ и универсальные типы

Краткое введение в универсальные типы и принципы их использования в запросах LINQ.

Преобразование данных с помощью LINQ

Описание различных способов преобразования данных, полученных в результате запросов.

Связи типов в операциях запросов LINQ

Сведения о сохранении и преобразовании типов в трех составляющих операции запроса LINQ.

Синтаксис запросов и синтаксис методов в LINQ

Сравнение синтаксиса методов и синтаксиса запросов как двух способов выражения запроса LINQ.

Возможности C#, поддерживающие LINQ

Описание языковых конструкций в C#, которые поддерживают LINQ.

Связанные разделы

Выражения запросов LINQ

Общие сведения о запросах в LINQ и ссылки на дополнительные ресурсы.

Общие сведения о стандартных операторах запросов

Представление стандартных методов, используемых в запросах LINQ.

Создание запросов LINQ на языке C#

18.12.2021 • 3 minutes to read

В этой статье рассматриваются три способа создания запросов LINQ в C#:

1. Используя синтаксис запроса.
2. Используя синтаксис метода.
3. Используя комбинацию синтаксиса запроса и синтаксиса метода.

Следующие примеры демонстрируют некоторые простые запросы LINQ, составленные с использованием каждого из указанных выше методов. Как правило, по возможности следует использовать метод (1), а (2) или (3) при необходимости.

> **NOTE**
> Эти запросы обращаются к простым коллекциям в памяти, однако базовый синтаксис при этом точно такой же, как в запросах LINQ to Entities и LINQ to XML.

Пример синтаксиса запросов

Для создания запросов рекомендуется использовать *синтаксис запроса*, позволяющий создавать *выражения запросов*. В следующем примере показаны три выражения запросов. Первое выражение запроса показывает, как фильтровать или ограничивать результаты, применяя условия в предложении `where`. Оно возвращает все элементы исходной последовательности, значения которых больше 7 или меньше 3. Второе выражение показывает, как сортировать возвращаемые результаты. Третье выражение показывает, как группировать результаты по ключу. Этот запрос возвращает две группы на основе первой буквы слова.

```
// Query #1.
List<int> numbers = new List<int>() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

// The query variable can also be implicitly typed by using var
IEnumerable<int> filteringQuery =
    from num in numbers
    where num < 3 || num > 7
    select num;

// Query #2.
IEnumerable<int> orderingQuery =
    from num in numbers
    where num < 3 || num > 7
    orderby num ascending
    select num;

// Query #3.
string[] groupingQuery = { "carrots", "cabbage", "broccoli", "beans", "barley" };
IEnumerable<IGrouping<char, string>> queryFoodGroups =
    from item in groupingQuery
    group item by item[0];
```

Обратите внимание, что запросы имеют тип IEnumerable<T>. Все эти запросы можно написать с помощью `var`, как показано в следующем примере:

`var query = from num in numbers...`

В каждом из предыдущих примеров запросы фактически не выполняются, пока не произойдет итерация переменной запроса в операторе `foreach` или другом операторе. Дополнительные сведения см. в разделе Введение в запросы LINQ.

Пример синтаксиса метода

Некоторые операции запросов должны быть выражены как вызов метода. Как правило, такие методы возвращают одноэлементные числовые значения, например Sum, Max, Min, Average и другие. Эти методы необходимо вызывать в запросе последними, поскольку представляют только одно значение и не могут служить источником для дополнительной операции запроса. В следующем примере показан вызов метода в выражении запроса:

```
List<int> numbers1 = new List<int>() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
List<int> numbers2 = new List<int>() { 15, 14, 11, 13, 19, 18, 16, 17, 12, 10 };
// Query #4.
double average = numbers1.Average();

// Query #5.
IEnumerable<int> concatenationQuery = numbers1.Concat(numbers2);
```

Если метод имеет параметры Action или Func, они предоставляются в виде лямбда-выражения, как показано в следующем примере:

```
// Query #6.
IEnumerable<int> largeNumbersQuery = numbers2.Where(c => c > 15);
```

Из числа предыдущих запросов незамедлительно выполняется только запрос 4. Это связано с тем, что возвращается одно значение, а не универсальная коллекция IEnumerable<T>. Сам метод для вычисления значения должен использовать `foreach`.

Каждый из предыдущих запросов может быть написан с использованием неявной типизации по переменной var, как показано в следующем примере:

```
// var is used for convenience in these queries
var average = numbers1.Average();
var concatenationQuery = numbers1.Concat(numbers2);
var largeNumbersQuery = numbers2.Where(c => c > 15);
```

Пример смешанного синтаксиса запроса и метода

В этом примере показано, как применять синтаксис метода к результатам предложения запроса. Просто заключите выражение запроса в круглые скобки, а затем примените оператор точки и вызовите метод. В следующем примере запрос 7 возвращает количество чисел со значением от 3 до 7. При этом в целом сохранять результат вызова метода лучше во вторую переменную. В этом случае меньше вероятность спутать запрос с его результатами.

```
// Query #7.

// Using a query expression with method syntax
int numCount1 =
    (from num in numbers1
     where num < 3 || num > 7
     select num).Count();

// Better: Create a new variable to store
// the method call result
IEnumerable<int> numbersQuery =
    from num in numbers1
    where num < 3 || num > 7
    select num;

int numCount2 = numbersQuery.Count();
```

Поскольку запрос 7 возвращает одно значение, а не коллекцию, он выполняется незамедлительно.

Предыдущий запрос можно написать, используя неявную типизацию с переменной `var`, как показано ниже:

```
var numCount = (from num in numbers...
```

При записи в синтаксисе метода он будет выглядеть следующим образом:

```
var numCount = numbers.Where(n => n < 3 || n > 7).Count();
```

При записи с явной типизацией он будет выглядеть следующим образом:

```
int numCount = numbers.Where(n => n < 3 || n > 7).Count();
```

См. также

- Пошаговое руководство: Написание запросов на C#
- LINQ
- предложение where

Запрос коллекции объектов

18.12.2021 • 2 minutes to read

В этом примере показано, как выполнить простой запрос к списку объектов `Student`. Каждый объект `Student` содержит некоторые основные сведения об учащемся, а также список, отражающий баллы, которые он набрал по результатам четырех экзаменов.

Это приложение служит основой для многих других примеров в этом разделе, где используется тот же источник данных `students`.

Пример

Следующий запрос возвращает список учащихся, набравших 90 баллов или больше на первом экзамене.

```
public class Student
{
    #region data
    public enum GradeLevel { FirstYear = 1, SecondYear, ThirdYear, FourthYear };

    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Id { get; set; }
    public GradeLevel Year;
    public List<int> ExamScores;

    protected static List<Student> students = new List<Student>
    {
        new Student {FirstName = "Terry", LastName = "Adams", Id = 120,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int> { 99, 82, 81, 79}},
        new Student {FirstName = "Fadi", LastName = "Fakhouri", Id = 116,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int> { 99, 86, 90, 94}},
        new Student {FirstName = "Hanying", LastName = "Feng", Id = 117,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int> { 93, 92, 80, 87}},
        new Student {FirstName = "Cesar", LastName = "Garcia", Id = 114,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int> { 97, 89, 85, 82}},
        new Student {FirstName = "Debra", LastName = "Garcia", Id = 115,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int> { 35, 72, 91, 70}},
        new Student {FirstName = "Hugo", LastName = "Garcia", Id = 118,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int> { 92, 90, 83, 78}},
        new Student {FirstName = "Sven", LastName = "Mortensen", Id = 113,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int> { 88, 94, 65, 91}},
        new Student {FirstName = "Claire", LastName = "O'Donnell", Id = 112,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int> { 75, 84, 91, 39}},
        new Student {FirstName = "Svetlana", LastName = "Omelchenko", Id = 111,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int> { 97, 92, 81, 60}},
        new Student {FirstName = "Lance", LastName = "Tucker", Id = 119,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int> { 68, 79, 88, 92}},
        new Student {FirstName = "Michael", LastName = "Tucker", Id = 122,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int> { 94, 92, 91, 91}},
```

```
        new Student {FirstName = "Eugene", LastName = "Zabokritski", Id = 121,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int> { 96, 85, 91, 60}}
    };
    #endregion

    // Helper method, used in GroupByRange.
    protected static int GetPercentile(Student s)
    {
        double avg = s.ExamScores.Average();
        return avg > 0 ? (int)avg / 10 : 0;
    }

    public static void QueryHighScores(int exam, int score)
    {
        var highScores = from student in students
                         where student.ExamScores[exam] > score
                         select new {Name = student.FirstName, Score = student.ExamScores[exam]};

        foreach (var item in highScores)
        {
            Console.WriteLine($"{item.Name,-15}{item.Score}");
        }
    }
}

public class Program
{
    public static void Main()
    {
        Student.QueryHighScores(1, 90);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
```

Этот запрос намеренно сделан простым, чтобы с ним можно было экспериментировать. Например, можно добавить в предложение `where` дополнительные условия или отсортировать результаты с помощью предложения `orderby`.

См. также

- LINQ
- Интерполяция строк

Практическое руководство. Возврат запроса из метода (руководство по программированию на C#)

18.12.2021 • 2 minutes to read

В этом примере показан способ возврата запроса из метода в качестве возвращаемого значения и параметра `out`.

Объекты запроса являются составляемыми, что означает возможность возврата запроса из метода. Объекты, представляющие запросы, не сохраняют результирующую коллекцию, а сохраняют действия, необходимые для получения результатов. Преимущество возвращения объектов запроса заключается в том, что их можно комбинировать или изменять. Поэтому любое возвращаемое значение или параметр `out` метода, который возвращает запрос, должны также иметь этот тип. Если метод материализует запрос в конкретный тип List<T> или Array, считается, что он должен возвращать результаты запроса, а не сам запрос. Переменную запроса, возвращаемую из метода, можно формировать или изменять.

Пример

В следующем примере первый метод возвращает запрос в качестве возвращаемого значения, а второй метод возвращает запрос в качестве параметра `out`. Обратите внимание, что в обоих случаях возвращается запрос, а не его результаты.

```
class MQ
{
    // QueryMethhod1 returns a query as its value.
    IEnumerable<string> QueryMethod1(ref int[] ints)
    {
        var intsToStrings = from i in ints
                            where i > 4
                            select i.ToString();
        return intsToStrings;
    }

    // QueryMethod2 returns a query as the value of parameter returnQ.
    void QueryMethod2(ref int[] ints, out IEnumerable<string> returnQ)
    {
        var intsToStrings = from i in ints
                            where i < 4
                            select i.ToString();
        returnQ = intsToStrings;
    }

    static void Main()
    {
        MQ app = new MQ();

        int[] nums = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

        // QueryMethod1 returns a query as the value of the method.
        var myQuery1 = app.QueryMethod1(ref nums);

        // Query myQuery1 is executed in the following foreach loop.
        Console.WriteLine("Results of executing myQuery1:");
        // Rest the mouse pointer over myQuery1 to see its type.
        foreach (string s in myQuery1)
```

```
        {
            Console.WriteLine(s);
        }

        // You also can execute the query returned from QueryMethod1
        // directly, without using myQuery1.
        Console.WriteLine("\nResults of executing myQuery1 directly:");
        // Rest the mouse pointer over the call to QueryMethod1 to see its
        // return type.
        foreach (string s in app.QueryMethod1(ref nums))
        {
            Console.WriteLine(s);
        }

        IEnumerable<string> myQuery2;
        // QueryMethod2 returns a query as the value of its out parameter.
        app.QueryMethod2(ref nums, out myQuery2);

        // Execute the returned query.
        Console.WriteLine("\nResults of executing myQuery2:");
        foreach (string s in myQuery2)
        {
            Console.WriteLine(s);
        }

        // You can modify a query by using query composition. A saved query
        // is nested inside a new query definition that revises the results
        // of the first query.
        myQuery1 = from item in myQuery1
                   orderby item descending
                   select item;

        // Execute the modified query.
        Console.WriteLine("\nResults of executing modified myQuery1:");
        foreach (string s in myQuery1)
        {
            Console.WriteLine(s);
        }

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

См. также

- LINQ

Сохранение результатов запроса в памяти

18.12.2021 • 2 minutes to read

Запрос, по сути, является набором инструкций, на основании которых осуществляется извлечение и организация данных. Запросы выполняются медленно по мере выполнения запроса к каждому последующему элементу в результирующем наборе. При использовании `foreach` для итерации результатов элементы возвращаются как те, к которым был получен доступ. Чтобы оценить запрос и сохранить его результаты без выполнения цикла `foreach`, просто вызовите один из следующих методов для переменной запроса:

- ToList
- ToArray
- ToDictionary
- ToLookup

Рекомендуется при сохранении результатов запроса назначить возвращенный объект коллекции новой переменной, как показано в следующем примере.

Пример

```
class StoreQueryResults
{
    static List<int> numbers = new List<int>() { 1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };
    static void Main()
    {

        IEnumerable<int> queryFactorsOfFour =
            from num in numbers
            where num % 4 == 0
            select num;

        // Store the results in a new variable
        // without executing a foreach loop.
        List<int> factorsofFourList = queryFactorsOfFour.ToList();

        // Iterate the list just to prove it holds data.
        Console.WriteLine(factorsofFourList[2]);
        factorsofFourList[2] = 0;
        Console.WriteLine(factorsofFourList[2]);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key");
        Console.ReadKey();
    }
}
```

См. также

- LINQ

Группировка результатов запросов

18.12.2021 • 7 minutes to read

Группирование — одна из самых эффективных функций LINQ. В приведенных ниже примерах демонстрируются различные способы группирования данных:

- по отдельному свойству;
- по первой букве строкового свойства;
- по расчетному числовому диапазону;
- по логическому предикату или другому выражению;
- по составному ключу.

Кроме того, два последних запроса передают свои результаты в новый анонимный тип, содержащий только имя и фамилию студента. Дополнительные сведения см. в разделе Предложение group.

Пример вспомогательного класса и источника данных

Во всех примерах в этом разделе используются указанные ниже вспомогательные классы и источники данных.

```
public class StudentClass
{
    #region data
    protected enum GradeLevel { FirstYear = 1, SecondYear, ThirdYear, FourthYear };
    protected class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int ID { get; set; }
        public GradeLevel Year;
        public List<int> ExamScores;
    }

    protected static List<Student> students = new List<Student>
    {
        new Student {FirstName = "Terry", LastName = "Adams", ID = 120,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 99, 82, 81, 79}},
        new Student {FirstName = "Fadi", LastName = "Fakhouri", ID = 116,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 99, 86, 90, 94}},
        new Student {FirstName = "Hanying", LastName = "Feng", ID = 117,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 93, 92, 80, 87}},
        new Student {FirstName = "Cesar", LastName = "Garcia", ID = 114,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 97, 89, 85, 82}},
        new Student {FirstName = "Debra", LastName = "Garcia", ID = 115,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 35, 72, 91, 70}},
        new Student {FirstName = "Hugo", LastName = "Garcia", ID = 118,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 92, 90, 83, 78}},
        new Student {FirstName = "Sven", LastName = "Mortensen", ID = 113,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 88, 94, 65, 91}},
```

```
        new Student {FirstName = "Claire", LastName = "O'Donnell", ID = 112,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 75, 84, 91, 39}},
        new Student {FirstName = "Svetlana", LastName = "Omelchenko", ID = 111,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 97, 92, 81, 60}},
        new Student {FirstName = "Lance", LastName = "Tucker", ID = 119,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 68, 79, 88, 92}},
        new Student {FirstName = "Michael", LastName = "Tucker", ID = 122,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 94, 92, 91, 91}},
        new Student {FirstName = "Eugene", LastName = "Zabokritski", ID = 121,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 96, 85, 91, 60}}
    };
    #endregion

    //Helper method, used in GroupByRange.
    protected static int GetPercentile(Student s)
    {
        double avg = s.ExamScores.Average();
        return avg > 0 ? (int)avg / 10 : 0;
    }

    public void QueryHighScores(int exam, int score)
    {
        var highScores = from student in students
                         where student.ExamScores[exam] > score
                         select new {Name = student.FirstName, Score = student.ExamScores[exam]};

        foreach (var item in highScores)
        {
            Console.WriteLine($"{item.Name,-15}{item.Score}");
        }
    }
}

public class Program
{
    public static void Main()
    {
        StudentClass sc = new StudentClass();
        sc.QueryHighScores(1, 90);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
```

Пример группировки по отдельному свойству

В этом примере показана группировка элементов источника с помощью отдельного свойства элемента в качестве ключа группы. В данном случае в качестве ключа используется `string`, фамилия учащегося. Для ключа можно также использовать подстроку. При операции группирования используется компаратор проверки на равенство, используемый по умолчанию для данного типа.

Вставьте приведенный ниже метод в класс `StudentClass`. Измените оператор вызова в методе `Main` на `sc.GroupBySingleProperty()`.

```
public void GroupBySingleProperty()
{
    Console.WriteLine("Group by a single property in an object:");

    // Variable queryLastNames is an IEnumerable<IGrouping<string,
    // DataClass.Student>>.
    var queryLastNames =
        from student in students
        group student by student.LastName into newGroup
        orderby newGroup.Key
        select newGroup;

    foreach (var nameGroup in queryLastNames)
    {
        Console.WriteLine($"Key: {nameGroup.Key}");
        foreach (var student in nameGroup)
        {
            Console.WriteLine($"\t{student.LastName}, {student.FirstName}");
        }
    }
}
/* Output:
    Group by a single property in an object:
    Key: Adams
            Adams, Terry
    Key: Fakhouri
            Fakhouri, Fadi
    Key: Feng
            Feng, Hanying
    Key: Garcia
            Garcia, Cesar
            Garcia, Debra
            Garcia, Hugo
    Key: Mortensen
            Mortensen, Sven
    Key: O'Donnell
            O'Donnell, Claire
    Key: Omelchenko
            Omelchenko, Svetlana
    Key: Tucker
            Tucker, Lance
            Tucker, Michael
    Key: Zabokritski
            Zabokritski, Eugene
*/
```

Пример группировки по значению

В этом примере показана группировка элементов источника, когда в качестве ключа группы используется не свойство объекта. В данном случае в качестве ключа используется первая буква фамилии учащегося.

Вставьте приведенный ниже метод в класс `StudentClass`. Измените оператор вызова в методе `Main` на `sc.GroupBySubstring()`.

```
public void GroupBySubstring()
{
    Console.WriteLine("\r\nGroup by something other than a property of the object:");

    var queryFirstLetters =
        from student in students
        group student by student.LastName[0];

    foreach (var studentGroup in queryFirstLetters)
    {
        Console.WriteLine($"Key: {studentGroup.Key}");
        // Nested foreach is required to access group items.
        foreach (var student in studentGroup)
        {
            Console.WriteLine($"\t{student.LastName}, {student.FirstName}");
        }
    }
}
/* Output:
    Group by something other than a property of the object:
    Key: A
            Adams, Terry
    Key: F
            Fakhouri, Fadi
            Feng, Hanying
    Key: G
            Garcia, Cesar
            Garcia, Debra
            Garcia, Hugo
    Key: M
            Mortensen, Sven
    Key: O
            O'Donnell, Claire
            Omelchenko, Svetlana
    Key: T
            Tucker, Lance
            Tucker, Michael
    Key: Z
            Zabokritski, Eugene
*/
```

Пример группировки по диапазону

В этом примере показана группировка элементов источника путем использования числового диапазона в качестве ключа группы. Затем запрос передает результаты в анонимный тип, содержащий только имя и фамилию, а также диапазон процентилей, к которому принадлежит студент. Использование анонимного типа объясняется тем, что для отображения результатов не требуется использовать полный объект `Student`. `GetPercentile` — это вспомогательная функция, которая вычисляет процент на основе средних результатов учащегося. Метод возвращает целое число в диапазоне от 0 до 10.

```
//Helper method, used in GroupByRange.
protected static int GetPercentile(Student s)
{
    double avg = s.ExamScores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}
```

Вставьте приведенный ниже метод в класс `StudentClass`. Измените оператор вызова в методе `Main` на `sc.GroupByRange()`.

```
public void GroupByRange()
{
    Console.WriteLine("\r\nGroup by numeric range and project into a new anonymous type:");

    var queryNumericRange =
        from student in students
        let percentile = GetPercentile(student)
        group new { student.FirstName, student.LastName } by percentile into percentGroup
        orderby percentGroup.Key
        select percentGroup;

    // Nested foreach required to iterate over groups and group items.
    foreach (var studentGroup in queryNumericRange)
    {
        Console.WriteLine($"Key: {studentGroup.Key * 10}");
        foreach (var item in studentGroup)
        {
            Console.WriteLine($"\t{item.LastName}, {item.FirstName}");
        }
    }
}
/* Output:
    Group by numeric range and project into a new anonymous type:
    Key: 60
            Garcia, Debra
    Key: 70
            O'Donnell, Claire
    Key: 80
            Adams, Terry
            Feng, Hanying
            Garcia, Cesar
            Garcia, Hugo
            Mortensen, Sven
            Omelchenko, Svetlana
            Tucker, Lance
            Zabokritski, Eugene
    Key: 90
            Fakhouri, Fadi
            Tucker, Michael
*/
```

Пример группировки по сравнению

В этом примере показана группировка элементов источника с помощью выражения логического сравнения. В этом случае логическое выражение проверяет, превосходит ли среднее значение экзаменационного результата учащегося 75 баллов. Как и в предыдущих примерах, результаты передаются в анонимный тип, так как весь элемент источника не требуется. Обратите внимание на то, что свойства в анонимном типе становятся свойствами в члене `Key`, а при выполнении запроса доступ к ним можно получить по имени.

Вставьте приведенный ниже метод в класс `StudentClass`. Измените оператор вызова в методе `Main` на `sc.GroupByBoolean()`.

```
public void GroupByBoolean()
{
    Console.WriteLine("\r\nGroup by a Boolean into two groups with string keys");
    Console.WriteLine("\"True\" and \"False\" and project into a new anonymous type:");
    var queryGroupByAverages = from student in students
                               group new { student.FirstName, student.LastName }
                                    by student.ExamScores.Average() > 75 into studentGroup
                               select studentGroup;

    foreach (var studentGroup in queryGroupByAverages)
    {
        Console.WriteLine($"Key: {studentGroup.Key}");
        foreach (var student in studentGroup)
            Console.WriteLine($"\t{student.FirstName} {student.LastName}");
    }
}
/* Output:
    Group by a Boolean into two groups with string keys
    "True" and "False" and project into a new anonymous type:
    Key: True
            Terry Adams
            Fadi Fakhouri
            Hanying Feng
            Cesar Garcia
            Hugo Garcia
            Sven Mortensen
            Svetlana Omelchenko
            Lance Tucker
            Michael Tucker
            Eugene Zabokritski
    Key: False
            Debra Garcia
            Claire O'Donnell
*/
```

Группировка по анонимному типу

В этом примере показано, как использовать анонимный тип для инкапсуляции ключа, содержащего несколько значений. В данном случае в качестве первого значения ключа используется первая буква фамилии учащегося. Второе значение ключа является логическим и указывает, набрал ли учащийся более 85 баллов на первом экзамене. Группы можно сортировать по любому из свойств в ключе.

Вставьте приведенный ниже метод в класс `StudentClass`. Измените оператор вызова в методе `Main` на `sc.GroupByCompositeKey()`.

```
public void GroupByCompositeKey()
{
    var queryHighScoreGroups =
        from student in students
        group student by new { FirstLetter = student.LastName[0],
            Score = student.ExamScores[0] > 85 } into studentGroup
        orderby studentGroup.Key.FirstLetter
        select studentGroup;

    Console.WriteLine("\r\nGroup and order by a compound key:");
    foreach (var scoreGroup in queryHighScoreGroups)
    {
        string s = scoreGroup.Key.Score == true ? "more than" : "less than";
        Console.WriteLine($"Name starts with {scoreGroup.Key.FirstLetter} who scored {s} 85");
        foreach (var item in scoreGroup)
        {
            Console.WriteLine($"\t{item.FirstName} {item.LastName}");
        }
    }
}

/* Output:
    Group and order by a compound key:
    Name starts with A who scored more than 85
            Terry Adams
    Name starts with F who scored more than 85
            Fadi Fakhouri
            Hanying Feng
    Name starts with G who scored more than 85
            Cesar Garcia
            Hugo Garcia
    Name starts with G who scored less than 85
            Debra Garcia
    Name starts with M who scored more than 85
            Sven Mortensen
    Name starts with O who scored less than 85
            Claire O'Donnell
    Name starts with O who scored more than 85
            Svetlana Omelchenko
    Name starts with T who scored less than 85
            Lance Tucker
    Name starts with T who scored more than 85
            Michael Tucker
    Name starts with Z who scored more than 85
            Eugene Zabokritski
*/
```

См. также

- GroupBy
- IGrouping<TKey,TElement>
- LINQ
- предложение group
- Анонимные типы
- Вложенный запрос в операции группирования
- Создание вложенной группы
- Группировка данных

Создание вложенной группы

18.12.2021 • 2 minutes to read

В следующем примере демонстрируется создание вложенных групп в выражении запроса LINQ. Каждая группа, созданная в соответствии с годом обучения или курсом учащегося, затем подразделяется на группы по именам учащихся.

Пример

> **NOTE**
>
> Этот пример содержит ссылки на объекты, определенные в примере кода в разделе Запрос коллекции объектов.

```
public void QueryNestedGroups()
{
    var queryNestedGroups =
        from student in students
        group student by student.Year into newGroup1
        from newGroup2 in
            (from student in newGroup1
             group student by student.LastName)
        group newGroup2 by newGroup1.Key;

    // Three nested foreach loops are required to iterate
    // over all elements of a grouped group. Hover the mouse
    // cursor over the iteration variables to see their actual type.
    foreach (var outerGroup in queryNestedGroups)
    {
        Console.WriteLine($"DataClass.Student Level = {outerGroup.Key}");
        foreach (var innerGroup in outerGroup)
        {
            Console.WriteLine($"\tNames that begin with: {innerGroup.Key}");
            foreach (var innerGroupElement in innerGroup)
            {
                Console.WriteLine($"\t\t{innerGroupElement.LastName} {innerGroupElement.FirstName}");
            }
        }
    }
}
/*
 Output:
DataClass.Student Level = SecondYear
        Names that begin with: Adams
                Adams Terry
        Names that begin with: Garcia
                Garcia Hugo
        Names that begin with: Omelchenko
                Omelchenko Svetlana
DataClass.Student Level = ThirdYear
        Names that begin with: Fakhouri
                Fakhouri Fadi
        Names that begin with: Garcia
                Garcia Debra
        Names that begin with: Tucker
                Tucker Lance
DataClass.Student Level = FirstYear
        Names that begin with: Feng
                Feng Hanying
        Names that begin with: Mortensen
                Mortensen Sven
        Names that begin with: Tucker
                Tucker Michael
DataClass.Student Level = FourthYear
        Names that begin with: Garcia
                Garcia Cesar
        Names that begin with: O'Donnell
                O'Donnell Claire
        Names that begin with: Zabokritski
                Zabokritski Eugene
 */
```

Обратите внимание, что для выполнения итерации по внутренним элементам вложенной группы необходимо три вложенных цикла `foreach`.

См. также

- LINQ

Вложенный запрос в операции группирования

18.12.2021 • 2 minutes to read

В этой статье показаны два разных способа создания запроса, упорядочивающего исходные данные в группы и затем выполняющего вложенный запрос для каждой группы по отдельности. Основное действие в каждом примере заключается в группировании исходных элементов с помощью *продолжения* с именем `newGroup` с последующим созданием вложенного запроса для `newGroup`. Этот вложенный запрос выполняется для каждой новой группы, создаваемой внешним запросом. В этом конкретном примере следует обратить внимание на то, что в конечном итоге получается не группа, а простая последовательность анонимных типов.

Дополнительные сведения о способах группирования см. в разделе Предложение group.

Дополнительные сведения о продолжениях см. в разделе into. В приведенном ниже примере в качестве источника данных используется структура данных в памяти, но те же принципы действуют для любого типа источника данных LINQ.

Пример

> **NOTE**
>
> Этот пример содержит ссылки на объекты, определенные в примере кода в разделе Запрос коллекции объектов.

```
public void QueryMax()
{
    var queryGroupMax =
        from student in students
        group student by student.Year into studentGroup
        select new
        {
            Level = studentGroup.Key,
            HighestScore =
            (from student2 in studentGroup
             select student2.ExamScores.Average()).Max()
        };

    int count = queryGroupMax.Count();
    Console.WriteLine($"Number of groups = {count}");

    foreach (var item in queryGroupMax)
    {
        Console.WriteLine($"  {item.Level} Highest Score={item.HighestScore}");
    }
}
```

Запрос, который приведен в фрагменте кода выше, можно записать, используя синтаксис метода. В следующем фрагменте кода приведен семантически эквивалентный запрос, написанный с использованием синтаксиса метода.

```
public void QueryMaxUsingMethodSyntax()
{
    var queryGroupMax = students
        .GroupBy(student => student.Year)
        .Select(studentGroup => new
        {
            Level = studentGroup.Key,
            HighestScore = studentGroup.Select(student2 => student2.ExamScores.Average()).Max()
        });

    int count = queryGroupMax.Count();
    Console.WriteLine($"Number of groups = {count}");

    foreach (var item in queryGroupMax)
    {
        Console.WriteLine($"  {item.Level} Highest Score={item.HighestScore}");
    }
}
```

См. также

- LINQ

Группирование результатов по смежным ключам

18.12.2021 • 7 minutes to read

В приведенном ниже примере показано, как сгруппировать элементы в блоки, представляющие последовательности смежных ключей. Например, предположим, что имеется следующая последовательность пар "ключ-значение":

КЛЮЧ	ЗНАЧЕНИЕ
Объект	Мы
Объект	думаем,
Объект	что
B	LINQ
C	is
Объект	действительно
B	известным
B	!

Следующие группы будут созданы в таком порядке:

1. Мы думаем, что
2. LINQ
3. is
4. действительно
5. известным, !

Решение реализуется как метод расширения, который является потокобезопасным и возвращает результаты потоковым образом. Иными словами, он создает группы по мере прохода по исходной последовательности. В отличие от операторов `group` и `orderby`, он может начать возвращать группы вызывающему объекту до того, как будет прочитана вся последовательность.

Потокобезопасность обеспечивается созданием копии каждой группы или блока по мере итерации исходной последовательности, как указывается в комментариях исходного кода. Если исходная последовательность имеет большой набор смежных элементов, среда CLR может создать исключение OutOfMemoryException.

Пример

В приведенном ниже примере показаны метод расширения и использующий его клиентский код:

```
using System;
using System.Collections.Generic;
```

```
using System.Collections.Generic;
using System.Linq;

namespace ChunkIt
{
    // Static class to contain the extension methods.
    public static class MyExtensions
    {
        public static IEnumerable<IGrouping<TKey, TSource>> ChunkBy<TSource, TKey>(this IEnumerable<TSource>
source, Func<TSource, TKey> keySelector)
        {
            return source.ChunkBy(keySelector, EqualityComparer<TKey>.Default);
        }

        public static IEnumerable<IGrouping<TKey, TSource>> ChunkBy<TSource, TKey>(this IEnumerable<TSource>
source, Func<TSource, TKey> keySelector, IEqualityComparer<TKey> comparer)
        {
            // Flag to signal end of source sequence.
            const bool noMoreSourceElements = true;

            // Auto-generated iterator for the source array.
            var enumerator = source.GetEnumerator();

            // Move to the first element in the source sequence.
            if (!enumerator.MoveNext()) yield break;

            // Iterate through source sequence and create a copy of each Chunk.
            // On each pass, the iterator advances to the first element of the next "Chunk"
            // in the source sequence. This loop corresponds to the outer foreach loop that
            // executes the query.
            Chunk<TKey, TSource> current = null;
            while (true)
            {
                // Get the key for the current Chunk. The source iterator will churn through
                // the source sequence until it finds an element with a key that doesn't match.
                var key = keySelector(enumerator.Current);

                // Make a new Chunk (group) object that initially has one GroupItem, which is a copy of the
current source element.
                current = new Chunk<TKey, TSource>(key, enumerator, value => comparer.Equals(key,
keySelector(value)));

                // Return the Chunk. A Chunk is an IGrouping<TKey,TSource>, which is the return value of the
ChunkBy method.
                // At this point the Chunk only has the first element in its source sequence. The remaining
elements will be
                // returned only when the client code foreach's over this chunk. See Chunk.GetEnumerator for
more info.
                yield return current;

                // Check to see whether (a) the chunk has made a copy of all its source elements or
                // (b) the iterator has reached the end of the source sequence. If the caller uses an inner
                // foreach loop to iterate the chunk items, and that loop ran to completion,
                // then the Chunk.GetEnumerator method will already have made
                // copies of all chunk items before we get here. If the Chunk.GetEnumerator loop did not
                // enumerate all elements in the chunk, we need to do it here to avoid corrupting the
iterator
                // for clients that may be calling us on a separate thread.
                if (current.CopyAllChunkElements() == noMoreSourceElements)
                {
                    yield break;
                }
            }
        }

        // A Chunk is a contiguous group of one or more source elements that have the same key. A Chunk
        // has a key and a list of ChunkItem objects, which are copies of the elements in the source
sequence.
        class Chunk<TKey, TSource> : IGrouping<TKey, TSource>
```

```
        {
            // INVARIANT: DoneCopyingChunk == true ||
            //   (predicate != null && predicate(enumerator.Current) && current.Value == enumerator.Current)

            // A Chunk has a linked list of ChunkItems, which represent the elements in the current chunk.
Each ChunkItem
            // has a reference to the next ChunkItem in the list.
            class ChunkItem
            {
                public ChunkItem(TSource value)
                {
                    Value = value;
                }
                public readonly TSource Value;
                public ChunkItem Next = null;
            }

            // The value that is used to determine matching elements
            private readonly TKey key;

            // Stores a reference to the enumerator for the source sequence
            private IEnumerator<TSource> enumerator;

            // A reference to the predicate that is used to compare keys.
            private Func<TSource, bool> predicate;

            // Stores the contents of the first source element that
            // belongs with this chunk.
            private readonly ChunkItem head;

            // End of the list. It is repositioned each time a new
            // ChunkItem is added.
            private ChunkItem tail;

            // Flag to indicate the source iterator has reached the end of the source sequence.
            internal bool isLastSourceElement = false;

            // Private object for thread syncronization
            private object m_Lock;

            // REQUIRES: enumerator != null && predicate != null
            public Chunk(TKey key, IEnumerator<TSource> enumerator, Func<TSource, bool> predicate)
            {
                this.key = key;
                this.enumerator = enumerator;
                this.predicate = predicate;

                // A Chunk always contains at least one element.
                head = new ChunkItem(enumerator.Current);

                // The end and beginning are the same until the list contains > 1 elements.
                tail = head;

                m_Lock = new object();
            }

            // Indicates that all chunk elements have been copied to the list of ChunkItems,
            // and the source enumerator is either at the end, or else on an element with a new key.
            // the tail of the linked list is set to null in the CopyNextChunkElement method if the
            // key of the next element does not match the current chunk's key, or there are no more elements
in the source.
            private bool DoneCopyingChunk => tail == null;

            // Adds one ChunkItem to the current group
            // REQUIRES: !DoneCopyingChunk && lock(this)
            private void CopyNextChunkElement()
            {
                // Try to advance the iterator on the source sequence.
                // If MoveNext returns false we are at the end, and isLastSourceElement is set to true
```

```
                isLastSourceElement = !enumerator.MoveNext();

                // If we are (a) at the end of the source, or (b) at the end of the current chunk
                // then null out the enumerator and predicate for reuse with the next chunk.
                if (isLastSourceElement || !predicate(enumerator.Current))
                {
                    enumerator = null;
                    predicate = null;
                }
                else
                {
                    tail.Next = new ChunkItem(enumerator.Current);
                }

                // tail will be null if we are at the end of the chunk elements
                // This check is made in DoneCopyingChunk.
                tail = tail.Next;
            }

            // Called after the end of the last chunk was reached. It first checks whether
            // there are more elements in the source sequence. If there are, it
            // Returns true if enumerator for this chunk was exhausted.
            internal bool CopyAllChunkElements()
            {
                while (true)
                {
                    lock (m_Lock)
                    {
                        if (DoneCopyingChunk)
                        {
                            // If isLastSourceElement is false,
                            // it signals to the outer iterator
                            // to continue iterating.
                            return isLastSourceElement;
                        }
                        else
                        {
                            CopyNextChunkElement();
                        }
                    }
                }
            }

            public TKey Key => key;

            // Invoked by the inner foreach loop. This method stays just one step ahead
            // of the client requests. It adds the next element of the chunk only after
            // the clients requests the last element in the list so far.
            public IEnumerator<TSource> GetEnumerator()
            {
                //Specify the initial element to enumerate.
                ChunkItem current = head;

                // There should always be at least one ChunkItem in a Chunk.
                while (current != null)
                {
                    // Yield the current item in the list.
                    yield return current.Value;

                    // Copy the next item from the source sequence,
                    // if we are at the end of our local list.
                    lock (m_Lock)
                    {
                        if (current == tail)
                        {
                            CopyNextChunkElement();
                        }
                    }
```

```
                    // Move to the next ChunkItem in the list.
                    current = current.Next;
                }
            }

            System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() =>
GetEnumerator();
        }
    }

    // A simple named type is used for easier viewing in the debugger. Anonymous types
    // work just as well with the ChunkBy operator.
    public class KeyValPair
    {
        public string Key { get; set; }
        public string Value { get; set; }
    }

    class Program
    {
        // The source sequence.
        public static IEnumerable<KeyValPair> list;

        // Query variable declared as class member to be available
        // on different threads.
        static IEnumerable<IGrouping<string, KeyValPair>> query;

        static void Main(string[] args)
        {
            // Initialize the source sequence with an array initializer.
            list = new[]
            {
                new KeyValPair{ Key = "A", Value = "We" },
                new KeyValPair{ Key = "A", Value = "think" },
                new KeyValPair{ Key = "A", Value = "that" },
                new KeyValPair{ Key = "B", Value = "Linq" },
                new KeyValPair{ Key = "C", Value = "is" },
                new KeyValPair{ Key = "A", Value = "really" },
                new KeyValPair{ Key = "B", Value = "cool" },
                new KeyValPair{ Key = "B", Value = "!" }
            };

            // Create the query by using our user-defined query operator.
            query = list.ChunkBy(p => p.Key);

            // ChunkBy returns IGrouping objects, therefore a nested
            // foreach loop is required to access the elements in each "chunk".
            foreach (var item in query)
            {
                Console.WriteLine($"Group key = {item.Key}");
                foreach (var inner in item)
                {
                    Console.WriteLine($"\t{inner.Value}");
                }
            }

            Console.WriteLine("Press any key to exit");
            Console.ReadKey();
        }
    }
}
```

Чтобы использовать метод расширения в своем проекте, скопируйте статический класс `MyExtensions` в новый или существующий файл исходного кода и, если необходимо, добавьте директиву `using` для пространства имен, в котором он находится.

См. также

- LINQ

Динамическое определение фильтров предикатов во время выполнения

18.12.2021 • 2 minutes to read

В некоторых случаях количество предикатов, которые нужно применить к исходным элементам в предложении `where`, неизвестно вплоть до времени выполнения. Одним из способов динамического определения сразу нескольких фильтров предикатов служит метод Contains, как показано в следующем примере. Пример конструируется двумя способами. Во-первых, проект запускается в результате применения фильтра к предоставленным в программе значениям. Затем проект запускается снова, когда применяются данные, введенные в среде выполнения.

Фильтрование с помощью метода Contains

1. Откройте новое консольное приложение и присвойте ему имя `PredicateFilters`.
2. Скопируйте класс `StudentClass` из окна Запрос коллекции объектов и вставьте его в пространство имен `PredicateFilters` под классом `Program`. `StudentClass` предоставляет список объектов `Student`.
3. Закомментируйте метод `Main` в `StudentClass`.
4. Замените класс `Program` на следующий код:

```
class DynamicPredicates : StudentClass
{
    static void Main(string[] args)
    {
        string[] ids = { "111", "114", "112" };

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    static void QueryByID(string[] ids)
    {
        var queryNames =
            from student in students
            let i = student.ID.ToString()
            where ids.Contains(i)
            select new { student.LastName, student.ID };

        foreach (var name in queryNames)
        {
            Console.WriteLine($"{name.LastName}: {name.ID}");
        }
    }
}
```

5. Добавьте в метод `Main` в классе `DynamicPredicates` следующую строку под объявлением `ids`.

```
QueryById(ids);
```

6. Запустите проект.

7. В консольном окне отобразятся следующие выходные данные:

   Гарсия: 114

   О'Доннелл: 112

   Омельченко: 111

8. Следующим шагом станет повторный запуск проекта, на этот раз с данными, введенными в среде выполнения, а не с массивом `ids`. В методе `Main` измените `QueryByID(ids)` на `QueryByID(args)`.

9. Запустите проект с аргументами командной строки `122 117 120 115`. Когда проект будет запущен, эти значения станут элементами `args` — параметра метода `Main`.

10. В консольном окне отобразятся следующие выходные данные:

    Адамс: 120

    Фенг: 117

    Гарсия: 115

    Такер: 122

Фильтрование с помощью оператора switch

1. Оператор `switch` позволяет выбрать один из предустановленных альтернативных запросов. В следующем примере `studentQuery` использует другое предложение `where`, в зависимости от того, какой уровень оценок или год указан в среде выполнения.

2. Скопируйте следующий метод и вставьте его в класс `DynamicPredicates`.

```
// To run this sample, first specify an integer value of 1 to 4 for the command
// line. This number will be converted to a GradeLevel value that specifies which
// set of students to query.
// Call the method: QueryByYear(args[0]);

static void QueryByYear(string level)
{
    GradeLevel year = (GradeLevel)Convert.ToInt32(level);
    IEnumerable<Student> studentQuery = null;
    switch (year)
    {
        case GradeLevel.FirstYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.FirstYear
                           select student;
            break;
        case GradeLevel.SecondYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.SecondYear
                           select student;
            break;
        case GradeLevel.ThirdYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.ThirdYear
                           select student;
            break;
        case GradeLevel.FourthYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.FourthYear
                           select student;
            break;

        default:
            break;
    }
    Console.WriteLine($"The following students are at level {year}");
    foreach (Student name in studentQuery)
    {
        Console.WriteLine($"{name.LastName}: {name.ID}");
    }
}
```

3. В методе `Main` замените вызов `QueryByID` на следующий вызов, который отправляет первый элемент из массива `args` как аргумент: `QueryByYear(args[0])`.

4. Запустите проект с помощью аргумента командной строки, выраженного целочисленным значением от 1 до 4.

См. также

- LINQ
- предложение where

Выполнение внутренних соединений

18.12.2021 • 10 minutes to read

В терминах реляционных баз данных *внутреннее соединение* формирует результирующий набор, в котором каждый элемент первой коллекции отображается по одному разу для каждого соответствующего элемента во второй коллекции. Если для элемента в первой коллекции нет соответствующих элементов, он не отображается в результирующем наборе. Метод Join, который вызывается предложением `join` в C#, реализует внутреннее соединение.

В этой статье показано выполнение четырех видов внутреннего соединения:

- Простое внутреннее соединение, сопоставляющее элементы из двух источников данных на основе простого ключа.
- Внутреннее соединение, сопоставляющее элементы из двух источников данных на основе *составного* ключа. Составной ключ — это ключ, который состоит из нескольких значений, позволяющих сопоставлять элементы на основе сразу нескольких свойств.
- *Множественное соединение*, при котором несколько последовательных операций соединения добавляются друг к другу.
- Внутреннее соединение, реализуемое с помощью группового соединения.

Пример соединения по простому ключу

В следующем примере создаются две коллекции, содержащие объекты двух определяемых пользователем типов, `Person` и `Pet`. Запрос включает предложение `join` в C# для сопоставления объектов `Person` с объектами `Pet`, где `Owner` — это `Person`. Предложение `select` в C# определяет, как будут выглядеть результирующие объекты. В этом примере результирующие объекты — это анонимные типы, состоящие из имени владельца и его домашнего животного.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// Simple inner join.
/// </summary>
public static void InnerJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };
    Person rui = new Person { FirstName = "Rui", LastName = "Raposo" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = rui };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene, rui };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    // Create a collection of person-pet pairs. Each element in the collection
    // is an anonymous type containing both the person's name and their pet's name.
    var query = from person in people
                join pet in pets on person equals pet.Owner
                select new { OwnerName = person.FirstName, PetName = pet.Name };

    foreach (var ownerAndPet in query)
    {
        Console.WriteLine($"\"{ownerAndPet.PetName}\" is owned by {ownerAndPet.OwnerName}");
    }
}

// This code produces the following output:
//
// "Daisy" is owned by Magnus
// "Barley" is owned by Terry
// "Boots" is owned by Terry
// "Whiskers" is owned by Charlotte
// "Blue Moon" is owned by Rui
```

Обратите внимание на то, что объект `Person`, параметр `LastName` которого имеет значение "Huff", не отображается в результирующем наборе, поскольку нет объекта `Pet`, параметр `Pet.Owner` которого совпадает с этим объектом `Person`.

Пример соединения по составному ключу

Вместо сопоставления элементов по одному единственному свойству можно воспользоваться составным ключом и сравнить элементы по нескольким свойствам. Для этого задайте функцию селектора ключа для каждой коллекции, чтобы вернуть анонимный тип, состоящий из свойств, которые вы хотите сравнить. Если свойства помечаются, они должны иметь одинаковую метку в анонимном типе каждого

ключа. Кроме того, свойства должны отображаться в одинаковом порядке.

В следующем примере на основе списка объектов `Employee` и списка объектов `Student` определяется, какие работники также являются студентами. Оба этих типа имеют свойства `FirstName` и `LastName` типа String. Функции, создающие ключи соединения из каждого элемента в списке, возвращают анонимный тип, состоящий из свойств `FirstName` и `LastName` каждого элемента. Операция соединения сравнивает эти составные ключи на предмет равенства и возвращает пары объектов из каждого списка, в которых имя и фамилия совпадают.

```
class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int EmployeeID { get; set; }
}

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int StudentID { get; set; }
}

/// <summary>
/// Performs a join operation using a composite key.
/// </summary>
public static void CompositeKeyJoinExample()
{
    // Create a list of employees.
    List<Employee> employees = new List<Employee> {
        new Employee { FirstName = "Terry", LastName = "Adams", EmployeeID = 522459 },
         new Employee { FirstName = "Charlotte", LastName = "Weiss", EmployeeID = 204467 },
         new Employee { FirstName = "Magnus", LastName = "Hedland", EmployeeID = 866200 },
         new Employee { FirstName = "Vernette", LastName = "Price", EmployeeID = 437139 } };

    // Create a list of students.
    List<Student> students = new List<Student> {
        new Student { FirstName = "Vernette", LastName = "Price", StudentID = 9562 },
        new Student { FirstName = "Terry", LastName = "Earls", StudentID = 9870 },
        new Student { FirstName = "Terry", LastName = "Adams", StudentID = 9913 } };

    // Join the two data sources based on a composite key consisting of first and last name,
    // to determine which employees are also students.
    IEnumerable<string> query = from employee in employees
                                join student in students
                                on new { employee.FirstName, employee.LastName }
                                equals new { student.FirstName, student.LastName }
                                select employee.FirstName + " " + employee.LastName;

    Console.WriteLine("The following people are both employees and students:");
    foreach (string name in query)
        Console.WriteLine(name);
}

// This code produces the following output:
//
// The following people are both employees and students:
// Terry Adams
// Vernette Price
```

Пример множественного соединения

Для выполнения множественного соединения можно соединять друг с другом любое количество

операций соединения. Каждое предложение `join` в C# сопоставляет указанный источник данных с результатами предыдущего соединения.

В следующем примере создаются три коллекции: список объектов `Person`, список объектов `Cat` и список объектов `Dog`.

Первое предложение `join` в C# сопоставляет людей и кошек, сравнивая объекты `Person``Cat.Owner`. Оно возвращает последовательность анонимных типов, содержащих объект `Person` и `Cat.Name`.

Второе предложение `join` в C# сопоставляет анонимные типы, возвращаемые первым соединением, с объектами `Dog` в предоставленном списке собак на основе ключа, состоящего из свойства `Owner` типа `Person` и первой буквы имени животного. Оно возвращает последовательность анонимных типов, содержащих свойства `Cat.Name` и `Dog.Name` из каждой совпадающей пары. Поскольку это внутреннее соединение, возвращаются только объекты из первого источника данных, имеющие соответствие во втором источнике данных.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

class Cat : Pet
{ }

class Dog : Pet
{ }

public static void MultipleJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };
    Person rui = new Person { FirstName = "Rui", LastName = "Raposo" };
    Person phyllis = new Person { FirstName = "Phyllis", LastName = "Harris" };

    Cat barley = new Cat { Name = "Barley", Owner = terry };
    Cat boots = new Cat { Name = "Boots", Owner = terry };
    Cat whiskers = new Cat { Name = "Whiskers", Owner = charlotte };
    Cat bluemoon = new Cat { Name = "Blue Moon", Owner = rui };
    Cat daisy = new Cat { Name = "Daisy", Owner = magnus };

    Dog fourwheeldrive = new Dog { Name = "Four Wheel Drive", Owner = phyllis };
    Dog duke = new Dog { Name = "Duke", Owner = magnus };
    Dog denim = new Dog { Name = "Denim", Owner = terry };
    Dog wiley = new Dog { Name = "Wiley", Owner = charlotte };
    Dog snoopy = new Dog { Name = "Snoopy", Owner = rui };
    Dog snickers = new Dog { Name = "Snickers", Owner = arlene };

    // Create three lists.
    List<Person> people =
        new List<Person> { magnus, terry, charlotte, arlene, rui, phyllis };
    List<Cat> cats =
        new List<Cat> { barley, boots, whiskers, bluemoon, daisy };
    List<Dog> dogs =
        new List<Dog> { fourwheeldrive, duke, denim, wiley, snoopy, snickers };

    // The first join matches Person and Cat.Owner from the list of people and
```

```
    // cats, based on a common Person. The second join matches dogs whose names start
    // with the same letter as the cats that have the same owner.
    var query = from person in people
                join cat in cats on person equals cat.Owner
                join dog in dogs on
                new { Owner = person, Letter = cat.Name.Substring(0, 1) }
                equals new { dog.Owner, Letter = dog.Name.Substring(0, 1) }
                select new { CatName = cat.Name, DogName = dog.Name };

    foreach (var obj in query)
    {
        Console.WriteLine(
            $"The cat \"{obj.CatName}\" shares a house, and the first letter of their name, with \"
{obj.DogName}\".");
    }
}

// This code produces the following output:
//
// The cat "Daisy" shares a house, and the first letter of their name, with "Duke".
// The cat "Whiskers" shares a house, and the first letter of their name, with "Wiley".
```

Пример внутреннего соединения с использованием группового соединения

В следующем примере показано, как реализовать внутреннее соединение с помощью группового соединения.

В `query1` список объектов `Person` группируется со списком объектов `Pet` на основе сравнения свойства объекта `Person` со свойством `Pet.Owner`. Групповое соединение создает коллекцию промежуточных групп, где каждая группа состоит из объекта `Person` и последовательности соответствующих ему объектов `Pet`.

Если в запрос добавляется второе предложение `from`, этот ряд последовательностей объединяется (или сводится) в одну более длинную последовательность. Тип элементов конечной последовательности определяется предложением `select`. В данном примере этот тип является анонимным и состоит из свойств `Person.FirstName` и `Pet.Name` для каждой совпадающей пары.

Результат `query1` эквивалентен результирующему набору, который можно было бы получить, выполнив внутреннее соединение с помощью предложения `join` без предложения `into`. Переменная `query2` демонстрирует этот эквивалентный запрос.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// Performs an inner join by using GroupJoin().
/// </summary>
public static void InnerGroupJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
```

```
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    var query1 = from person in people
                 join pet in pets on person equals pet.Owner into gj
                 from subpet in gj
                 select new { OwnerName = person.FirstName, PetName = subpet.Name };

    Console.WriteLine("Inner join using GroupJoin():");
    foreach (var v in query1)
    {
        Console.WriteLine($"{v.OwnerName} - {v.PetName}");
    }

    var query2 = from person in people
                 join pet in pets on person equals pet.Owner
                 select new { OwnerName = person.FirstName, PetName = pet.Name };

    Console.WriteLine("\nThe equivalent operation using Join():");
    foreach (var v in query2)
        Console.WriteLine($"{v.OwnerName} - {v.PetName}");
}

// This code produces the following output:
//
// Inner join using GroupJoin():
// Magnus - Daisy
// Terry - Barley
// Terry - Boots
// Terry - Blue Moon
// Charlotte - Whiskers
//
// The equivalent operation using Join():
// Magnus - Daisy
// Terry - Barley
// Terry - Boots
// Terry - Blue Moon
// Charlotte - Whiskers
```

См. также

- Join
- GroupJoin
- Выполнение групповых соединений
- Выполнение левых внешних соединений
- Анонимные типы

Выполнение групповых соединений

18.12.2021 • 4 minutes to read

Групповое соединение можно использовать для создания иерархических структур данных. Каждый элемент из первой коллекции группируется с набором соответствующих элементов из второй коллекции.

Например, класс или таблица реляционной базы данных под названием `Student` может содержать два поля: `Id` и `Name`. Второй класс или таблица реляционной базы данных под названием `Course` может содержать два поля: `StudentId` и `CourseTitle`. Групповое соединение этих двух источников данных, основанное на сравнении `Student.Id` и `Course.StudentId`, приведет к группировке каждого объекта `Student` с коллекцией объектов `Course` (которая может быть пустой).

> **NOTE**
> Каждый элемент первой коллекции отображается в результирующем наборе группового соединения вне зависимости от того, найдены ли соответствующие ему элементы во второй коллекции. В случае, если такие элементы не найдены, последовательность соответствующих элементов будет пуста. Селектор результата имеет доступ к каждому элементу первой коллекции. При этом следует учесть отличие от селектора результата в негрупповом соединении: в этом случае селектор не имеет доступа к элементам из первой коллекции, для которых нет совпадений во второй коллекции.

> **WARNING**
> Метод Enumerable.GroupJoin не имеет прямого эквивалента в традиционных терминах реляционных баз данных. Однако он реализует расширенный набор внутренних соединений и левых внешних соединений. Обе эти операции можно записать с точки зрения группового соединения. Дополнительные сведения см. в статье Операции соединения и в разделе об операторе GroupJoin.

В первом примере этой статьи показано, как выполнить групповое соединение. Во втором примере — как использовать групповое соединение для создания XML-элементов.

Пример группового соединения

В приведенном ниже примере выполняется групповое соединение объектов типа `Person` и `Pet` на основе сравнения `Person` свойства `Pet.Owner`. В отличие от негруппового соединения, в котором создается пара элементов для каждого точного совпадения, групповое соединение создает один результирующий объект для каждого элемента первой коллекции, то есть объекта `Person` в данном примере. Соответствующие элементы из второй коллекции (в данном примере — объекты `Pet`) группируются в коллекцию. Наконец, результирующая функция селектора создает для каждого совпадения анонимный тип, состоящий из `Person.FirstName` и коллекции объектов `Pet`.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// This example performs a grouped join.
/// </summary>
public static void GroupJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    // Create a list where each element is an anonymous type
    // that contains the person's first name and a collection of
    // pets that are owned by them.
    var query = from person in people
                join pet in pets on person equals pet.Owner into gj
                select new { OwnerName = person.FirstName, Pets = gj };

    foreach (var v in query)
    {
        // Output the owner's name.
        Console.WriteLine($"{v.OwnerName}:");
        // Output each of the owner's pet's names.
        foreach (Pet pet in v.Pets)
            Console.WriteLine($"  {pet.Name}");
    }
}

// This code produces the following output:
//
// Magnus:
//   Daisy
// Terry:
//   Barley
//   Boots
//   Blue Moon
// Charlotte:
//   Whiskers
// Arlene:
```

Пример группового соединения для создания XML

Групповые соединения идеально подходят для создания XML с помощью LINQ to XML. Приведенный ниже пример аналогичен предыдущему за исключением того, что вместо создания анонимных типов

результирующая функция селектора создает XML-элементы, представляющие соединенные объекты.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// This example creates XML output from a grouped join.
/// </summary>
public static void GroupJoinXMLExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    // Create XML to display the hierarchical organization of people and their pets.
    XElement ownersAndPets = new XElement("PetOwners",
        from person in people
        join pet in pets on person equals pet.Owner into gj
        select new XElement("Person",
            new XAttribute("FirstName", person.FirstName),
            new XAttribute("LastName", person.LastName),
            from subpet in gj
            select new XElement("Pet", subpet.Name)));

    Console.WriteLine(ownersAndPets);
}

// This code produces the following output:
//
// <PetOwners>
//   <Person FirstName="Magnus" LastName="Hedlund">
//     <Pet>Daisy</Pet>
//   </Person>
//   <Person FirstName="Terry" LastName="Adams">
//     <Pet>Barley</Pet>
//     <Pet>Boots</Pet>
//     <Pet>Blue Moon</Pet>
//   </Person>
//   <Person FirstName="Charlotte" LastName="Weiss">
//     <Pet>Whiskers</Pet>
//   </Person>
//   <Person FirstName="Arlene" LastName="Huff" />
// </PetOwners>
```

См. также

- Join
- GroupJoin
- Выполнение внутренних соединений
- Выполнение левых внешних соединений
- Анонимные типы

Выполнение левых внешних соединений

18.12.2021 • 2 minutes to read

Левое внешнее соединение — это соединение, при котором каждый элемент первой коллекции возвращается независимо от наличия взаимосвязанных элементов во второй коллекции. LINQ можно использовать для левого внешнего соединения, вызвав метод DefaultIfEmpty на основании результатов группового соединения.

Пример

В приведенном ниже примере показано, как использовать метод DefaultIfEmpty применительно к результатам группового соединения для выполнения левого внешнего соединения.

Первым шагом выполнения левого внешнего соединения двух коллекций является выполнение внутреннего соединения с помощью группового соединения. (Описание этого процесса см. в статье Выполнение внутренних соединений.) В этом примере список объектов `Person` соединен посредством внутреннего соединения со списком объектов `Pet` на основе объекта `Person`, соответствующего `Pet.Owner`.

Вторым шагом является включение каждого элемента первой (левой) коллекции в набор результатов, даже если элемент не имеет совпадений в правой коллекции. Эта процедура выполняется путем вызова метода DefaultIfEmpty для каждой последовательности совпадающих элементов из группового соединения. В этом примере DefaultIfEmpty вызывается для каждой последовательности совпадающих объектов `Pet`. Метод возвращает коллекцию, содержащую одно значение по умолчанию, если последовательность соответствующих объектов `Pet` пуста для любого объекта `Person`, обеспечивая представление каждого объекта `Person` в коллекции результатов.

> **NOTE**
> Значение по умолчанию для ссылочного типа — `null`, поэтому в примере проверяется пустая ссылка перед доступом к каждому элементу в каждой коллекции `Pet`.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

public static void LeftOuterJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    var query = from person in people
                join pet in pets on person equals pet.Owner into gj
                from subpet in gj.DefaultIfEmpty()
                select new { person.FirstName, PetName = subpet?.Name ?? String.Empty };

    foreach (var v in query)
    {
        Console.WriteLine($"{v.FirstName+":",-15}{v.PetName}");
    }
}

// This code produces the following output:
//
// Magnus:        Daisy
// Terry:         Barley
// Terry:         Boots
// Terry:         Blue Moon
// Charlotte:     Whiskers
// Arlene:
```

См. также

- Join
- GroupJoin
- Выполнение внутренних соединений
- Выполнение групповых соединений
- Анонимные типы

Упорядочение результатов предложения соединения

18.12.2021 • 2 minutes to read

В этом примере показано, как упорядочить результаты операции соединения. Обратите внимание на то, что упорядочение выполняется после соединения. Несмотря на то, что предложение `orderby` с одним или несколькими исходными последовательностями использовать до соединения можно, обычно это не рекомендуется. Некоторые поставщики LINQ могут не сохранять этот порядок после соединения.

Пример

Этот запрос создает группу соединения, а затем сортирует группы по элементу категории, который остается в области действия. В инициализаторе анонимного типа подзапрос упорядочивает все соответствующие элементы из последовательности продуктов.

```
    class HowToOrderJoins
    {
        #region Data
        class Product
        {
            public string Name { get; set; }
            public int CategoryID { get; set; }
        }

        class Category
        {
            public string Name { get; set; }
            public int ID { get; set; }
        }

        // Specify the first data source.
        List<Category> categories = new List<Category>()
 {
    new Category(){Name="Beverages", ID=001},
    new Category(){ Name="Condiments", ID=002},
    new Category(){ Name="Vegetables", ID=003},
    new Category() {  Name="Grains", ID=004},
    new Category() {  Name="Fruit", ID=005}
 };

        // Specify the second data source.
        List<Product> products = new List<Product>()
{
   new Product{Name="Cola",  CategoryID=001},
   new Product{Name="Tea",  CategoryID=001},
   new Product{Name="Mustard", CategoryID=002},
   new Product{Name="Pickles", CategoryID=002},
   new Product{Name="Carrots", CategoryID=003},
   new Product{Name="Bok Choy", CategoryID=003},
   new Product{Name="Peaches", CategoryID=005},
   new Product{Name="Melons", CategoryID=005},
 };
        #endregion
        static void Main()
        {
            HowToOrderJoins app = new HowToOrderJoins();
            app.OrderJoin1();
```

```
            // Keep console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }

        void OrderJoin1()
        {
            var groupJoinQuery2 =
                from category in categories
                join prod in products on category.ID equals prod.CategoryID into prodGroup
                orderby category.Name
                select new
                {
                    Category = category.Name,
                    Products = from prod2 in prodGroup
                               orderby prod2.Name
                               select prod2
                };

            foreach (var productGroup in groupJoinQuery2)
            {
                Console.WriteLine(productGroup.Category);
                foreach (var prodItem in productGroup.Products)
                {
                    Console.WriteLine($"  {prodItem.Name,-10} {prodItem.CategoryID}");
                }
            }
        }
        /* Output:
            Beverages
              Cola       1
              Tea        1
            Condiments
              Mustard    2
              Pickles    2
            Fruit
              Melons     5
              Peaches    5
            Grains
            Vegetables
              Bok Choy   3
              Carrots    3
         */
    }
```

См. также

- LINQ
- предложение orderby
- предложение join

Соединение с помощью составных ключей

18.12.2021 • 2 minutes to read

В этом примере показано, как выполнить операции соединения, в которых требуется использовать более одного ключа для определения соответствия. Для этих целей используется составной ключ. Составной ключ создается как анонимный тип или именованный тип со значениями, которые нужно сравнить. Если переменная запроса будет передаваться за пределы метода, необходимо использовать именованный тип, который переопределяет Equals и GetHashCode для ключа. Имена свойств и порядок, в котором они возникают, должны совпадать в каждом ключе.

Пример

В следующем примере демонстрируется использование составного ключа для объединения данных из трех таблиц:

```
var query = from o in db.Orders
    from p in db.Products
    join d in db.OrderDetails
        on new {o.OrderID, p.ProductID} equals new {d.OrderID, d.ProductID} into details
        from d in details
        select new {o.OrderID, p.ProductID, d.UnitPrice};
```

Определение типа в составных ключах зависит от имен свойств в ключах и порядка, в котором они возникают. Если свойства в исходных последовательностях имеют другие имена, в ключах им необходимо присвоить новые имена. Например, если в таблицах `Orders` и `OrderDetails` используются разные имена столбцов, можно создать составные ключи, назначив одинаковые имена в анонимных типах:

```
join...on new {Name = o.CustomerName, ID = o.CustID} equals
    new {Name = d.CustName, ID = d.CustID }
```

Составные ключи можно также использовать в предложении `group`.

См. также

- LINQ
- предложение join
- предложение group

Выполнение пользовательских операций соединения

18.12.2021 • 4 minutes to read

В этом примере показано, как выполнить операции соединения, которые нельзя осуществить с помощью предложения `join`. В выражении запроса предложение `join` ограничено уравнивающими соединениями и оптимизировано для них; они являются самым общим типом операций соединения. При выполнении уравнивающего соединения использование предложения `join` почти всегда обеспечивает наилучшую производительность.

Однако предложение `join` нельзя использовать в указанных ниже случаях.

- Соединение объявлено в выражении неравенства (не уравнивающее соединение).
- Соединение объявлено в нескольких выражениях равенства или неравенства.
- Необходимость создания временной переменной диапазона для правосторонней (внутренней) последовательности перед операцией соединения.

Чтобы независимо представить каждый источник данных при выполнении соединений, не являющихся уравнивающими, можно использовать несколько предложений `from`. Затем к переменной диапазона для каждого источника можно применить выражение предиката в предложении `where`. Выражение также может принимать форму вызова метода.

> **NOTE**
> Не путайте этот вид пользовательской операции соединения с использованием нескольких предложений `from` для доступа к внутренним коллекциям. Дополнительные сведения см. в разделе Предложение join.

Пример 1

Первый метод в приведенном ниже примере показывает простое перекрестное соединение. Перекрестные соединения следует использовать с осторожностью, так как они могут возвращать очень большие наборы результатов. Однако такие соединения могут быть полезны при создании исходных последовательностей, относительно которых выполняются дополнительные запросы.

Результатом второго метода является последовательность всех продуктов, идентификатор категории которых находится в списке категорий с правой стороны. Учтите необходимость использования предложения `let` и метода `Contains` для создания временного массива. Также можно создать массив перед запросом и удалить первое предложение `from`.

```
class CustomJoins
{

    #region Data

    class Product
    {
        public string Name { get; set; }
        public int CategoryID { get; set; }
    }

    class Category
```

```
    class Category
    {
        public string Name { get; set; }
        public int ID { get; set; }
    }

    // Specify the first data source.
    List<Category> categories = new List<Category>()
    {
        new Category(){Name="Beverages", ID=001},
        new Category(){ Name="Condiments", ID=002},
        new Category(){ Name="Vegetables", ID=003},
    };

    // Specify the second data source.
    List<Product> products = new List<Product>()
    {
        new Product{Name="Tea",  CategoryID=001},
        new Product{Name="Mustard", CategoryID=002},
        new Product{Name="Pickles", CategoryID=002},
        new Product{Name="Carrots", CategoryID=003},
        new Product{Name="Bok Choy", CategoryID=003},
        new Product{Name="Peaches", CategoryID=005},
        new Product{Name="Melons", CategoryID=005},
        new Product{Name="Ice Cream", CategoryID=007},
        new Product{Name="Mackerel", CategoryID=012},
    };
    #endregion

    static void Main()
    {
        CustomJoins app = new CustomJoins();
        app.CrossJoin();
        app.NonEquijoin();

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    void CrossJoin()
    {
        var crossJoinQuery =
            from c in categories
            from p in products
            select new { c.ID, p.Name };

        Console.WriteLine("Cross Join Query:");
        foreach (var v in crossJoinQuery)
        {
            Console.WriteLine($"{v.ID,-5}{v.Name}");
        }
    }

    void NonEquijoin()
    {
        var nonEquijoinQuery =
            from p in products
            let catIds = from c in categories
                         select c.ID
            where catIds.Contains(p.CategoryID) == true
            select new { Product = p.Name, CategoryID = p.CategoryID };

        Console.WriteLine("Non-equijoin query:");
        foreach (var v in nonEquijoinQuery)
        {
            Console.WriteLine($"{v.CategoryID,-5}{v.Product}");
        }
    }
}
```

```
    /* Output:
Cross Join Query:
1    Tea
1    Mustard
1    Pickles
1    Carrots
1    Bok Choy
1    Peaches
1    Melons
1    Ice Cream
1    Mackerel
2    Tea
2    Mustard
2    Pickles
2    Carrots
2    Bok Choy
2    Peaches
2    Melons
2    Ice Cream
2    Mackerel
3    Tea
3    Mustard
3    Pickles
3    Carrots
3    Bok Choy
3    Peaches
3    Melons
3    Ice Cream
3    Mackerel
Non-equijoin query:
1    Tea
2    Mustard
2    Pickles
3    Carrots
3    Bok Choy
Press any key to exit.
     */
```

Пример 2

В приведенном ниже примере запрос должен соединять две последовательности на основе сопоставления ключей, которые в случае с внутренней (правосторонней) последовательностью не могут быть получены до предложения соединения. Если соединение было выполнено с предложением `join`, для каждого элемента требуется вызвать метод `Split`. Использование нескольких предложений `from` позволяет запросу избежать издержек, связанных с повторным вызовом метода. Однако поскольку предложение `join` оптимизировано, в этом случае его использование может быть эффективнее нескольких предложений `from`. Результаты будут зависеть в основном от затрат на вызов метода.

```
class MergeTwoCSVFiles
{
    static void Main()
    {
        // See section Compiling the Code for information about the data files.
        string[] names = System.IO.File.ReadAllLines(@"../../../names.csv");
        string[] scores = System.IO.File.ReadAllLines(@"../../../scores.csv");

        // Merge the data sources using a named type.
        // You could use var instead of an explicit type for the query.
        IEnumerable<Student> queryNamesScores =
            // Split each line in the data files into an array of strings.
            from name in names
            let x = name.Split(',')
            from score in scores
            let s = score.Split(',')
```

```
            // Look for matching IDs from the two data files.
            where x[2] == s[0]
            // If the IDs match, build a Student object.
            select new Student()
            {
                FirstName = x[0],
                LastName = x[1],
                ID = Convert.ToInt32(x[2]),
                ExamScores = (from scoreAsText in s.Skip(1)
                              select Convert.ToInt32(scoreAsText)).
                              ToList()
            };

        // Optional. Store the newly created student objects in memory
        // for faster access in future queries
        List<Student> students = queryNamesScores.ToList();

        foreach (var student in students)
        {
            Console.WriteLine($"The average score of {student.FirstName} {student.LastName} is
{student.ExamScores.Average()}.");
        }

        //Keep console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int ID { get; set; }
    public List<int> ExamScores { get; set; }
}

/* Output:
    The average score of Omelchenko Svetlana is 82.5.
    The average score of O'Donnell Claire is 72.25.
    The average score of Mortensen Sven is 84.5.
    The average score of Garcia Cesar is 88.25.
    The average score of Garcia Debra is 67.
    The average score of Fakhouri Fadi is 92.25.
    The average score of Feng Hanying is 88.
    The average score of Garcia Hugo is 85.75.
    The average score of Tucker Lance is 81.75.
    The average score of Adams Terry is 85.25.
    The average score of Zabokritski Eugene is 83.
    The average score of Tucker Michael is 92.
 */
```

См. также

- LINQ
- предложение join
- Упорядочение результатов предложения соединения

Обработка значений NULL в выражениях запросов

18.12.2021 • 2 minutes to read

В этом примере показано, как обрабатывать возможные нулевые значения в исходных коллекциях. Коллекция объектов, например IEnumerable<T>, может содержать элементы со значением NULL. Если исходная коллекция равна `null` или содержит элемент со значением `null`, а ваш запрос не может обрабатывать значения `null`, то при выполнении запроса будет выдано исключение NullReferenceException.

Чтобы избежать исключения в связи с пустой ссылкой, можно составить защитный код, как показано в следующем примере:

```
var query1 =
    from c in categories
    where c != null
    join p in products on c.ID equals
        p?.CategoryID
    select new { Category = c.Name, Name = p.Name };
```

В предыдущем примере предложение `where` отфильтровывает все пустые элементы в последовательности категорий. Этот метод не связан с проверкой на наличие пустых значений в предложении join. Условное выражение со значением NULL в этом примере работает, поскольку `Products.CategoryID` имеет тип `int?`, что является сокращением `Nullable<int>`.

Если в предложении join только один из ключей сравнения имеет тип, допускающий значение NULL, остальные типы в выражении запроса можно привести к типу, допускающему значение NULL. В следующем примере предполагается, что `EmployeeID` — это столбец, содержащий значения типа `int?`:

```
void TestMethod(Northwind db)
{
    var query =
        from o in db.Orders
        join e in db.Employees
            on o.EmployeeID equals (int?)e.EmployeeID
        select new { o.OrderID, e.FirstName };
}
```

В каждом из примеров в запросе используется ключевое слово `equals`. В версии C# 9 добавлено сопоставление шаблонов, включающее шаблоны для `is null` и `is not null`. Эти шаблоны не рекомендуется использовать в запросах LINQ, так как поставщики запросов могут неверно интерпретировать новый синтаксис C#. Поставщик запросов — это библиотека, которая преобразует выражения запросов C# в собственный формат данных, например Entity Framework Core. Поставщики запросов реализуют интерфейс System.Linq.IQueryProvider, чтобы создавать источники данных, реализующие интерфейс System.Linq.IQueryable<T>.

См. также

- Nullable<T>
- LINQ

- Типы значений, допускающие значение NULL

Обработка исключений в выражениях запросов

18.12.2021 • 2 minutes to read

В контексте выражения запроса можно вызвать любой метод. Тем не менее рекомендуется избегать вызова в выражении запроса любого метода, который может создавать побочные эффекты, например изменение содержимого источника данных или создание исключения. В этом примере показано, как избежать создания исключений при вызове методов в выражениях запросов без нарушения общих рекомендаций .NET относительно обработки исключений. Эти рекомендации гласят, что можно перехватывать определенные исключения, если вы понимаете, почему они создаются в данном контексте. Дополнительные сведения см. в разделе Лучшие методики обработки исключений.

В последнем примере показано, как обрабатывать случаи, когда необходимо создать исключение во время выполнения запроса.

Пример 1

В следующем примере показано, как переместить код обработки исключений за пределы выражения запроса. Это возможно, только если метод не зависит от переменных, являющихся локальными для запроса.

```
class ExceptionsOutsideQuery
{
    static void Main()
    {
        // DO THIS with a datasource that might
        // throw an exception. It is easier to deal with
        // outside of the query expression.
        IEnumerable<int> dataSource;
        try
        {
            dataSource = GetData();
        }
        catch (InvalidOperationException)
        {
            // Handle (or don't handle) the exception
            // in the way that is appropriate for your application.
            Console.WriteLine("Invalid operation");
            goto Exit;
        }

        // If we get here, it is safe to proceed.
        var query = from i in dataSource
                    select i * i;

        foreach (var i in query)
            Console.WriteLine(i.ToString());

        //Keep the console window open in debug mode
        Exit:
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // A data source that is very likely to throw an exception!
    static IEnumerable<int> GetData()
    {
        throw new InvalidOperationException();
    }
}
```

Пример 2

В некоторых случаях лучшей реакцией на исключение, которое вызывается в запросе, будет немедленное прекращение выполнения запроса. В следующем примере демонстрируется обработка исключений, которые могут быть созданы из тела запроса. Предположим, что `SomeMethodThatMightThrow` потенциально может вызвать исключение, которое требует остановки выполнения запроса.

Обратите внимание, что блок `try` включает цикл `foreach`, а не сам запрос. Это вызвано тем, что цикл `foreach` является точкой, в которой запрос фактически выполняется. Дополнительные сведения см. в статье Введение в запросы LINQ (C#).

```
class QueryThatThrows
{
    static void Main()
    {
        // Data source.
        string[] files = { "fileA.txt", "fileB.txt", "fileC.txt" };

        // Demonstration query that throws.
        var exceptionDemoQuery =
            from file in files
            let n = SomeMethodThatMightThrow(file)
            select n;

        // Runtime exceptions are thrown when query is executed.
        // Therefore they must be handled in the foreach loop.
        try
        {
            foreach (var item in exceptionDemoQuery)
            {
                Console.WriteLine($"Processing {item}");
            }
        }

        // Catch whatever exception you expect to raise
        // and/or do any necessary cleanup in a finally block
        catch (InvalidOperationException e)
        {
            Console.WriteLine(e.Message);
        }

        //Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // Not very useful as a general purpose method.
    static string SomeMethodThatMightThrow(string s)
    {
        if (s[4] == 'C')
            throw new InvalidOperationException();
        return @"C:\newFolder\" + s;
    }
}
/* Output:
    Processing C:\newFolder\fileA.txt
    Processing C:\newFolder\fileB.txt
    Operation is not valid due to the current state of the object.
 */
```

См. также

- LINQ

Написание безопасного и эффективного кода C#

18.12.2021 • 15 minutes to read

C# позволяет создавать проверяемый безопасный код с более высокой производительностью. Если вы будете внимательно применять эти методы, у вас будет меньше сценариев, требующих небезопасного кода. Эти функции упрощают использование ссылок на типы значений в качестве аргументов метода и возвращаемых значений метода. При безопасном выполнении эти методики сводят к минимуму копирование типов значений. Используя типы значений, можно свести к минимуму число распределений и сборок мусора.

В большей части примера кода в этой статье демонстрируются функции, добавленные в C# 7.2. Чтобы использовать эти функции, убедитесь, что проект не настроен на использование более ранней версии. Дополнительные сведения см. в разделе Настройка языковой версии.

Преимущество использования типов значений заключается в том, что они часто позволяют избежать выделения памяти в кучах. Недостаток состоит в том, что они копируются по значению. Этот компромисс усложняет оптимизацию алгоритмов, работающих с большими объемами данных. Языковые функции, описываемые в этой статье, предоставляют механизмы, которые обеспечивают безопасный эффективный код с использованием ссылок на типы значений. При рациональном использовании этих функций можно свести к минимуму число операций выделения и копирования.

Некоторые рекомендации в этой статье относятся к методикам программирования, которые рекомендуется использовать всегда, а не только для повышения производительности. Используйте ключевое слово `readonly`, если оно точно отражает намерение проекта:

- Объявите неизменяемые структуры как `readonly`.
- Объявите элементы `readonly` для изменяемых структур.

В статье также объясняются некоторые низкоуровневые оптимизации, которые рекомендуется использовать при запуске профилировщика и выявлении узких мест.

- Использование модификатора параметра `in`.
- Использование инструкций `ref readonly return`.
- Использование типов `ref struct`.
- Использование типов `nint` и `nuint`.

Эти методы поддерживают компромисс между двумя целями:

- Сокращение количества выделений в куче.

  Переменные, являющиеся ссылочными типами, содержат ссылку на расположение в памяти и выделяются в управляемой куче. Ссылка копируется только когда ссылочный тип передается в качестве аргумента в метод или возвращается из метода. Каждый новый объект требует новое распределение и впоследствии должен быть освобожден. Сборка мусора занимает некоторое время.

- Сокращение числа копирований значений.

  Переменные, которые являются типами значений, непосредственно содержат их значения, и значение обычно копируется при передаче в метод или возвращается из метода. Это поведение включает в себя копирование значения `this` при вызове членов типа значения. Операция копирования занимает некоторое время, в зависимости от размера типа.

В этой статье используется следующий пример концепции трехмерной структуры для объяснения этих рекомендаций:

```
public struct Point3D
{
    public double X;
    public double Y;
    public double Z;
}
```

Различные примеры использования других реализаций этой концепции.

Объявите неизменяемые структуры как `readonly`

Объявите `readonly struct`, чтобы указать, что тип **неизменяемый**. Модификатор `readonly` сообщает компилятору, что ваша цель — создать неизменяемый тип. Компилятор указывает это решение со следующими правилами:

- Все элементы поля должны быть доступны только для чтения.
- Все свойства должны быть только для чтения, включая автоматически реализуемые свойства.

Этих двух правил достаточно, чтобы убедиться, что ни один из элементов `readonly struct` не изменяет состояние этой структуры. Объект `struct` является неизменяемым. Структура `Point3D` может быть определена как неизменяемая, как показано в следующем примере:

```
readonly public struct ReadonlyPoint3D
{
    public ReadonlyPoint3D(double x, double y, double z)
    {
        this.X = x;
        this.Y = y;
        this.Z = z;
    }

    public double X { get; }
    public double Y { get; }
    public double Z { get; }
}
```

Следуйте этим рекомендациям, когда планируете создать неизменяемый тип значения. Улучшения производительности являются дополнительным преимуществом. Ключевые слова `readonly struct` четко выражают намерение проекта.

Объявите элементы `readonly` для изменяемых структур.

В C# 8.0 и более поздних версиях, когда тип структуры является изменяемым, объявите элементы, которые не изменяют состояние как элементы `readonly`.

Рассмотрим другое приложение, для которого требуется структура объемных точек, однако оно должно поддерживать изменяемость. Следующая версия структуры объемных точек добавляет модификатор `readonly` только к тем элементам, которые не изменяют структуру. Используйте этот пример, если проект должен поддерживать модификации структуры некоторыми членами, однако вам все равно нужны преимущества реализации через `readonly` для некоторых членов:

```
public struct Point3D
{
    public Point3D(double x, double y, double z)
    {
        _x = x;
        _y = y;
        _z = z;
    }

    private double _x;
    public double X
    {
        readonly get => _x;
        set => _x = value;
    }

    private double _y;
    public double Y
    {
        readonly get => _y;
        set => _y = value;
    }

    private double _z;
    public double Z
    {
        readonly get => _z;
        set => _z = value;
    }

    public readonly double Distance => Math.Sqrt(X * X + Y * Y + Z * Z);

    public readonly override string ToString() => $"{X}, {Y}, {Z}";
}
```

В предыдущем примере показаны многие расположения, в которых можно применить модификатор `readonly`: методы, свойства и методы доступа к свойствам. При использовании автоматически реализуемых свойств компилятор добавляет модификатор `readonly` к методу доступа `get` для свойств, предназначенных для чтения и записи. Компилятор добавляет модификатор `readonly` к автоматически реализуемым объявлениям свойств для свойств только с методом доступа `get`.

Добавление модификатора `readonly` к членам, которые не изменяют состояние, предоставляет два связанных преимущества. Во первых, компилятор применяет свое намерение. Этот член не может изменять состояние структуры. Во-вторых, компилятор не создает защитные копии параметров `in` при доступе к члену `readonly`. Компилятор может безопасно выполнить эту оптимизацию, так как она гарантирует, что `struct` не изменяется членом `readonly`.

Использование инструкций `ref readonly return`

Используйте возвращаемое значение `ref readonly`, если выполняются оба следующих условия.

- Возвращаемое значение `struct` больше, чем IntPtr.Size.
- Время существования хранилища больше, чем значение, возвращаемое методом.

Вы можете возвращать значения по ссылке, когда возвращаемое значение не является локальным для возвращающего метода. Возврат по ссылке означает, что копируется только ссылка, не структура. В следующем примере свойство `Origin` не может использовать возврат `ref`, так как возвращаемое значение является локальной переменной:

```
public Point3D Origin => new Point3D(0,0,0);
```

Тем не менее, следующее определение свойства может возвращаться по ссылке, так как возвращаемое значение является статическим элементом:

```
public struct Point3D
{
    private static Point3D origin = new Point3D(0,0,0);

    // Dangerous! returning a mutable reference to internal storage
    public ref Point3D Origin => ref origin;

    // other members removed for space
}
```

Вы не хотите, чтобы вызывающие объекты изменяли источник, поэтому следует возвращать значение через `ref readonly`:

```
public struct Point3D
{
    private static Point3D origin = new Point3D(0,0,0);

    public static ref readonly Point3D Origin => ref origin;

    // other members removed for space
}
```

Возвращение `ref readonly` позволяет сохранить копирование больших структур и неизменность внутренних элементов данных.

Во время вызова вызывающие объекты выбирают использовать свойство `Origin` как `ref readonly` или как значение:

```
var originValue = Point3D.Origin;
ref readonly var originReference = ref Point3D.Origin;
```

При первом назначении в предыдущем примере кода создается и назначается копия константы `Origin`. При втором назначается ссылка. Обратите внимание, что модификатор `readonly` должен быть частью объявления переменной. Ссылку, на которую он ссылается, изменить невозможно. В противном случае возникнет ошибка времени компиляции.

В объявлении `originReference` требуется модификатор `readonly`.

Компилятор применяет правило, не позволяющее вызывающему объекту изменять ссылку. Попытки назначить значение напрямую вызывают ошибку времени компиляции. В других случаях компилятор выделяет защитную копию, если он не может безопасно использовать ссылку только для чтения. Правила статического анализа определяют, можно ли изменить структуру. Компилятор не создает защитную копию, если структура или ее член является `readonly struct`. Защитная копия не требуется для передачи структуры в качестве аргумента `in`.

Использование модификатора параметра `in`.

В следующих разделах объясняется, что делает модификатор `in`, как и когда его использовать для оптимизации производительности.

- Ключевые слова `out`, `ref` и `in`
- использование параметров `in` для больших структур
- Необязательное использование `in` на сайте вызова
- Избегание защитных копий

Ключевые слова `out`, `ref` и `in`

При передаче аргументов по ссылке можно использовать ключевое слово `in` как дополнение к ключевым словам `ref` и `out`. Ключевое слово `in` указывает, что аргумент передается по ссылке, но вызванный метод не изменяет это значение. Модификатор `in` может применяться к любому члену, который принимает параметры: к методам, делегатам, лямбда-выражениям, локальным функциям, индексаторам и операторам.

С добавлением ключевого слова `in` C# предоставляет полный словарь для выражения намерения проекта. Если в сигнатуре метода не указан ни один из следующих модификаторов, типы значений копируются при передаче в вызываемый метод. Каждый из этих модификаторов указывает, что переменная передается по ссылке, предотвращая копирование. Каждый модификатор выражает конкретное намерение.

- `out`. этот метод задает значение аргумента, используемого в качестве этого параметра.
- `ref`. Этот метод может изменять значение аргумента, используемого в качестве этого параметра.
- `in`. этот метод не изменяет значение аргумента, используемого в качестве этого параметра.

При добавлении модификатора `in` для передачи аргумента по ссылке вы объявляете о своем намерении передавать аргументы по ссылке, чтобы избежать ненужных операций копирования. Вы не собираетесь изменять объект, используемый в качестве этого аргумента.

Существуют и другие способы, которыми модификатор `in` дополняет `out` и `ref`. Невозможно создать перегрузки метода, которые отличаются только наличием `in`, `out` или `ref`. Эти новые правила расширяют то же поведение, которое всегда действовало для параметров `out` и `ref`. Как и модификаторы `out` и `ref`, типы значений не упаковываются, так как применяется модификатор `in`. Еще одно преимущество параметров `in` состоит в том, что вы можете использовать литеральные значения или константы для аргумента в параметре `in`.

Модификатор `in` также можно использовать со ссылочными типами или числовыми значениями. Однако преимущества в этих случаях минимальны (если они вообще есть).

Существует несколько способов, когда компилятор гарантирует, что аргумент `in` доступен только для чтения. Во-первых, вызванный метод не может быть напрямую назначен параметру `in`. Его невозможно напрямую назначить полю параметра `in`, когда это значение имеет тип `struct`. Кроме того, параметр `in` невозможно передать какому-либо методу, использующему модификатор `ref` или `out`. Эти правила применяются к любому полю параметра `in` при условии, что данное поле имеет тип `struct` и параметр имеет тип `struct`. На самом деле эти правила применяются к нескольким уровням доступа к членам при условии, что все уровни доступа к членам являются `structs`. Компилятор принудительно указывает, что типы `struct`, передаваемые в качестве аргументов `in`, и их члены `struct` являются переменными, доступными только для чтения, когда используются в качестве аргументов для других методов.

Использование параметров `in` для больших структур

Модификатор `in` можно применить к любому параметру `readonly struct`, но такой подход может повысить производительность только для типов значений, которые значительно больше IntPtr.Size. Для простых типов (таких как `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool` и `enum`) возможное повышение производительности минимально. Некоторые простые типы, например `decimal` размером 16 байт, имеют больший размер, чем 4-байтовые или 8-байтовые ссылки, но недостаточно большой для обеспечения значительной разницы в производительности в

большинстве сценариев. Производительность может снизиться при использовании передачи по ссылке для типов, меньше чем IntPtr.Size.

Ниже приведен пример метода, который вычисляет расстояние между двумя точками в трехмерном пространстве.

```
private static double CalculateDistance(in Point3D point1, in Point3D point2)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

Аргументами являются две структуры, каждая из которых содержит три типа double. double имеет размер 8 байт, поэтому каждый аргумент равен 24 байтам. Указывая модификатор `in`, вы, в зависимости от архитектуры компьютера, передаете этим аргументам 4- или 8-байтовую ссылку. Разница в размере невелика, но она может вырасти, когда приложение вызывает этот метод в непрерывном цикле с помощью множества различных значений.

Однако для оценки выигрыша в производительности следует измерять влияние любых низкоуровневых оптимизаций, таких как использование модификатора `in`. Например, можно подумать, что использование `in` в параметре Guid может оказаться полезным. Размер типа `Guid` равен 16 байтам, что вдвое больше размера 8-байтовой ссылки. Но такое небольшое различие, скорее всего, не приведет к серьезному выигрышу в производительности, если оно не находится в методе, который находится на критическом пути для вашего приложения.

Необязательное использование `in` на сайте вызова

В отличие от параметра `ref` или `out`, не нужно применять модификатор `in` на сайте вызова. В следующем коде показаны два примера вызова метода `CalculateDistance`. В первом используются две локальные переменные, передаваемые по ссылке. Второй содержит временную переменную, созданную в рамках вызова метода.

```
var distance = CalculateDistance(pt1, pt2);
var fromOrigin = CalculateDistance(pt1, new Point3D());
```

Пропуск модификатора `in` в месте вызова сообщает компилятору, что он может сделать копию аргумента по следующим причинам:

- Выполняется неявное преобразование, но не преобразование удостоверения из типа аргумента в тип параметра.
- Аргумент является выражением, но не имеет известную переменную хранения.
- Существует перегрузка, которая отличается наличием или отсутствием `in`. В этом случае перегрузка по значению подходит лучше.

Эти правила полезны и при обновлении существующего кода для использования аргументов со ссылками, доступными только для чтения. Внутри вызываемого метода можно вызвать любой метод экземпляра, который использует параметры передачи по значению. В этих экземплярах создается копия параметра `in`.

Поскольку компилятор может создавать временную переменную для любого параметра `in`, вы можете также указать значения по умолчанию для любого параметра `in`. Следующий код указывает начало координат (точку 0,0,0) в качестве значения по умолчанию для второй точки:

```
private static double CalculateDistance2(in Point3D point1, in Point3D point2 = default)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

Чтобы велеть компилятору передавать аргументы, доступные только для чтения, по ссылке, укажите модификатор `in` для аргументов в месте вызова, как показано в следующем коде:

```
distance = CalculateDistance(in pt1, in pt2);
distance = CalculateDistance(in pt1, new Point3D());
distance = CalculateDistance(pt1, in Point3D.Origin);
```

Это упрощает постепенное внедрение параметров `in` в больших базах кода, где возможен выигрыш по производительности. Сначала нужно добавить модификатор `in` в сигнатуры методов. Затем можно добавить модификатор `in` в местах вызовов и создать типы `readonly struct`, чтобы разрешить компилятору не создавать защитные копии параметров `in` в дополнительных расположениях.

Избегание защитных копий

Передавайте `struct` в качестве аргумента для параметра `in` только в том случае, если он объявлен с модификатором `readonly`, или метод обращается к только элементам `readonly` этой структуры. В противном случае компилятор должен создавать *защитные копии* во многих ситуациях, чтобы гарантировать, что аргументы не изменяются. Рассмотрим следующий пример, который вычисляет расстояние до точки в трехмерном пространстве от начала координат:

```
private static double CalculateDistance(in Point3D point1, in Point3D point2)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

Структура `Point3D` *не* предоставляется только для чтения. В тексте этого метода есть шесть разных вызовов доступа к свойству. На первый взгляд может показаться, что эти доступы безопасны. В конце концов, метод доступа `get` не должен изменять состояние объекта. Но нет правила языка, которое это обеспечивает. Это просто общее соглашение. Любой тип может реализовывать метод доступа `get`, который изменил внутреннее состояние.

Без языковой гарантии компилятору необходимо создавать временную копию аргумента перед вызовом любого члена, не помеченного модификатором `readonly`. Временное хранилище создается на стеке, значения аргумента копируются во временном хранилище, и значение копируется в стек для каждого доступа к членам как аргумент `this`. Во многих случаях эти копии достаточно снижают производительность, так что передача по значению работает быстрее, чем передача по ссылке только для чтения, если тип аргумента не `readonly struct` и метод вызывает члены, не помеченные как `readonly`. Если пометить все методы, которые не изменяют состояние структуры как `readonly`, компилятор сможет безопасно определить, что состояние структуры не изменено и защитная копия не требуется.

Если вычисление расстояния использует неизменяемую структуру, `ReadonlyPoint3D`, временные объекты не требуются:

```
private static double CalculateDistance3(in ReadonlyPoint3D point1, in ReadonlyPoint3D point2 = default)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

Компилятор создает более эффективный код, когда вызываются члены `readonly struct`. Ссылка `this`, а не копия приемника, всегда является параметром `in`, переданным по ссылке методу члена. Эта оптимизация позволяет избежать копирования при использовании `readonly struct` в качестве аргумента `in`.

Не передавайте тип значения, допускающий значения NULL, в качестве аргумента `in`. Тип Nullable<T> не объявлен как структура только для чтения. Это означает, что компилятор должен создавать защитные копии для любого аргумента типа, допускающего значение NULL и передаваемого в метод с помощью модификатора `in` в объявлении параметра.

Вы видите пример программы, который демонстрирует разницу в производительности с помощью BenchmarkDotNet в наших репозиториях примеров на сайте GitHub. Он сравнивает передачу изменяемых структур по значению и по ссылке с передачей неизменяемых структур по значению и по ссылке. Быстрее всего использовать неизменяемую структуру и передачу по ссылке.

Использование типов `ref struct`

Используйте `ref struct` или `readonly ref struct`, например Span<T> или ReadOnlySpan<T>, для работы с блоками памяти как последовательностью байтов. Объем памяти, используемой диапазоном только для чтения, будет ограничен одним кадром стека. Это ограничение позволяет компилятору кое-что оптимизировать. Главным стимулом для создания этой функции была структура Span<T> и связанные структуры. Вы получите повышение производительности благодаря этим усовершенствованиям, если будете использовать новые и обновленные интерфейсы API .NET, которые используют тип Span<T>.

Объявление структуры как `readonly ref` сочетает в себе преимущества и недостатки объявлений `ref struct` и `readonly struct`. Объем памяти, используемой диапазоном только для чтения, будет ограничен одним кадром стека, а объем памяти, используемой диапазоном только для чтения, невозможно изменить.

Похожие требования могут иметь место при работе с памятью, созданной с помощью `stackalloc`, или при использовании памяти из API взаимодействия. Для этих задач можно определить собственные типы `ref struct`.

Использование типов `nint` и `nuint`

Целочисленные типы собственного размера — это 32-разрядные целые числа в 32-разрядном процессе или 64-разрядные целые числа в 64-разрядном процессе. Используйте их для сценариев взаимодействия, с низкоуровневыми библиотеками и для оптимизации производительности в сценариях, где часто выполняются математические операции с целыми числами.

Выводы

Использование типов значений сводит к минимуму число операций распределения:

- Хранилище для типов значений выделяется в стеке для локальных переменных и аргументов метода.
- Хранилище для типов значений, которые являются членами других объектов, выделяется как часть

этого объекта, а не отдельное распределение.
- Хранилища возвращаемых значений типов значений выделяется в стеке.

Сравните это со ссылочными типами в таких же ситуациях:

- Хранилище для ссылочных типов выделяется в куче для локальных переменных и аргументов метода. Ссылка хранится в стеке.
- Хранилище для ссылочных типов, которые являются членами других объектов, выделяется в куче отдельно. Объект хранит ссылку.
- Хранилище возвращаемых значений ссылочного типа выделяется в куче. Ссылка на это хранилище хранится в стеке.

Чтобы свести распределения к минимуму, придется пойти на компромисс. Вы копируете больше памяти, если размер `struct` больше, чем размер ссылки. Ссылка обычно является 64- или 32-разрядной и зависит от ЦП целевого компьютера.

Эти компромиссы обычно имеют минимальное влияние на производительность. Однако для больших структур или больших коллекций влияние на производительность возрастает. Влияние может быть большим в плотных циклах и часто используемых путях для программ.

Эти усовершенствования языка C# предназначены для критических алгоритмов производительности, когда минимизация распределений памяти может иметь большое значение для достижения необходимой производительности. Может оказаться, что в создаваемом коде эти функции используются довольно редко. Тем не менее эти усовершенствования были реализованы в .NET. Поскольку с этими функциями работает все больше API-интерфейсов, повышение производительности приложений не останется незаметным.

См. также

- Модификатор параметров in (справочник по C#)
- Ключевое слово ref
- Возвращаемые значения ref и локальные переменные ref

Деревья выражений

18.12.2021 • 2 minutes to read

Если вы использовали LINQ, то у вас есть опыт работы с полнофункциональной библиотекой, в которой типы `Func` являются частью набора API. (Если вы не работали с LINQ, перед чтением этого раздела рекомендуем ознакомиться с руководством по LINQ и статьей о лямбда-выражениях.) *Деревья выражений* обеспечивают более широкие возможности взаимодействия с аргументами, являющимися функциями.

Аргументы функции применяются (как правило, с помощью лямбда-выражений) при создании запросов LINQ. В типичном запросе LINQ аргументы функции преобразуются в делегат, создаваемый компилятором.

Если требуются более широкие возможности взаимодействия, необходимо использовать *деревья выражений*. Деревья выражений представляют код в виде структуры, которую можно анализировать, изменять или выполнять. Это дает возможность управлять кодом во время выполнения. Вы можете написать код, который анализирует выполняющиеся алгоритмы или добавляет новые возможности. В более сложных сценариях вы можете изменять выполняющиеся алгоритмы и даже преобразовывать выражения C# в иную форму для выполнения в другой среде.

Вероятно, вы уже писали код, в котором используются деревья выражений. Интерфейсы API LINQ платформы Entity Framework принимают деревья выражений в качестве аргументов для модели выражений запросов LINQ. Это позволяет платформе Entity Framework преобразовывать запросы, написанные на C#, в код SQL, который выполняется в ядре СУБД. Другим примером является Moq — популярная платформа прототипирования для .NET.

В дальнейших разделах этого учебника описывается, что представляют собой деревья выражений, рассматриваются поддерживающие их классы платформы и демонстрируются способы работы с деревьями выражений. Вы узнаете, как считывать деревья выражений, как создавать их, а также как создавать модифицированные деревья выражений и выполнять код, представленный деревьями выражений. После ознакомления с учебником вы будете готовы к использованию этих структур для создания эффективных адаптивных алгоритмов.

1. Описание деревьев выражений

   Описание структуры и принципов использования *деревьев выражений*.

2. Типы платформ, поддерживающие деревья выражений

   Сведения о структурах и классах, которые служат для определения деревьев выражений и управления ими.

3. Выполнение выражений

   Сведения о том, как преобразовать дерево выражения, представленное как лямбда-выражение, в делегат и как выполнить полученный делегат.

4. Интерпретация выражений

   Сведения об обходе и анализе *деревьев выражений* для получения представления о том, какой код они представляют.

5. Построение выражений

   Сведения о построении узлов для дерева выражения и сборке деревьев выражений.

6. Преобразование выражений

   Сведения о создании измененной копии дерева выражения или преобразовании дерева выражения в другой формат.

7. Подведем итоги

   Сводные сведения о деревьях выражений.

Описание деревьев выражений

18.12.2021 • 3 minutes to read

Предыдущий раздел — Общие сведения

Дерево выражения — это структура данных, которая определяет код. Деревья выражений основаны на тех же структурах, которые компилятор использует для анализа кода и создания скомпилированных выходных данных. По мере изучения этого раздела вы заметите много общего между деревьями выражений и типами, используемыми в API Roslyn для создания анализаторов и исправлений кода. (Анализаторы и исправления кода представляют собой пакеты NuGet, которые выполняют статический анализ кода и могут предлагать возможные исправления для разработчика.) Эти понятия похожи, и конечный результат представляет собой структуру данных, которая позволяет осмысленно проанализировать исходный код. Однако деревья выражений основаны на совершенно другом наборе классов и интерфейсов API, нежели API Roslyn.

Рассмотрим простой пример. Вот строка кода:

```
var sum = 1 + 2;
```

Если анализировать ее как дерево выражения, то оно содержит несколько узлов. Внешний узел — это оператор объявления переменной с присваиванием (`var sum = 1 + 2;`). Этот внешний узел содержит несколько дочерних узлов: объявление переменной, оператор присваивания и выражение, представляющее часть справа от знака равенства. Это выражение далее делится на выражения, представляющие операцию сложения, а также ее левый и правый операнды.

Давайте немного более подробно рассмотрим выражения, составляющие часть справа от знака равенства. Выражение имеет вид `1 + 2`. Это двоичное выражение. Если точнее, это выражение двоичного сложения. Выражение двоичного сложения имеет два дочерних элемента, представляющих левый и правый узлы выражения сложения. В этом случае оба узла являются константными выражениями: левый операнд — это значение `1`, а правый — значение `2`.

Визуально весь оператор представляет собой дерево: вы можете начать с корневого узла и переходить к каждому узлу дерева, чтобы просмотреть код, составляющий оператор.

- Оператор объявления переменной с присваиванием (`var sum = 1 + 2;`)
  - Неявное объявление типа переменной (`var sum`)
    - Неявное ключевое слово var (`var`)
    - Объявление имени переменной (`sum`)
  - Оператор присваивания (`=`)
  - Выражение двоичного сложения (`1 + 2`)
    - Левый операнд (`1`)
    - Оператор сложения (`+`)
    - Правый операнд (`2`)

Это может показаться сложным, но это очень удобно. Тем же самым образом можно раскладывать на составные части гораздо более сложные выражения. Рассмотрим следующее выражение:

```
var finalAnswer = this.SecretSauceFunction(
    currentState.createInterimResult(), currentState.createSecondValue(1, 2),
    decisionServer.considerFinalOptions("hello")) +
    MoreSecretSauce('A', DateTime.Now, true);
```

Приведенное выше выражение также является объявлением переменной с присваиванием. В этом экземпляре правая часть присваивания представляет собой гораздо более сложное дерево. Мы не будем раскладывать это выражение на составные части, но рассмотрим возможные узлы. Имеются вызовы методов, использующие текущий объект в качестве получателя: в одном случае получатель указан явно `this`; в другом нет. Имеются вызовы методов, использующие другие объекты-получатели, а также константные аргументы различных типов. Наконец, имеется оператор двоичного сложения. В зависимости от типа возвращаемого значения `SecretSauceFunction()` или `MoreSecretSauce()` оператор двоичного сложения может быть вызовом метода, адресованным к переопределенному оператору сложения, который разрешается в вызов статического метода, адресованный к оператору двоичного сложения, определенному для класса.

Несмотря на кажущуюся сложность, приведенное выше выражение образует структуру дерева, по которой можно переходить так же легко, как и в первом примере. Вы можете обходить дочерние узлы для поиска листовых узлов в выражении. Родительские узлы содержат ссылки на свои дочерние узлы, и каждый узел имеет свойство, описывающее тип этого узла.

Структура дерева выражения очень согласована. Изучив основы, вы сможете разобраться даже в самом сложном коде, если он представлен деревом выражения. Стройность структуры данных объясняет, почему компилятор C# может анализировать самые сложные программы C# и создавать правильные выходные данные на основе исходного кода.

Когда вы разберетесь в структуре деревьев выражений, полученные знания позволят вам работать с множеством еще более сложных сценариев. Деревья выражений крайне эффективны.

Помимо преобразования алгоритмов для выполнения в других средах, деревья выражений можно использовать с целью упростить написание алгоритмов, проверяющих код перед его выполнением. Можно создать метод, аргументами которого являются выражения, а затем анализировать эти выражения перед выполнением кода. Дерево выражения — это полное представление кода: вы можете просмотреть значения любого подвыражения. Вы можете просмотреть имена методов и свойств. Вы можете узнать значение любого константного выражения. Кроме того, вы можете преобразовать дерево выражения в исполняемый делегат, а затем выполнить код.

Интерфейсы API для деревьев выражений позволяют создавать деревья, которые представляют практически любую допустимую конструкцию кода. Однако для обеспечения максимальной простоты некоторые идиомы C# нельзя создавать в дереве выражения. Одним из примеров являются асинхронные выражения (с ключевыми словами `async` и `await`). Если требуются асинхронные алгоритмы, то необходимо работать с объектами `Task` напрямую, а не полагаться на поддержку компилятора. Другой пример — создание циклов. Как правило, они создаются с помощью операторов `for`, `foreach`, `while` и `do`. Как вы увидите далее в этой серии, интерфейсы API для деревьев выражений поддерживают единственное выражение цикла, причем повторами управляют выражения `break` и `continue`.

Единственное, чего нельзя сделать, — это изменить дерево выражения. Деревья выражений являются неизменяемыми структурами данных. Если необходимо изменить дерево выражения, следует создать копию исходного дерева и внести в нее требуемые изменения.

Следующий раздел — Типы платформ, поддерживающие деревья выражений

Типы платформ, поддерживающие деревья выражений

18.12.2021 • 2 minutes to read

Предыдущий раздел: "Описание деревьев выражений"

Платформа .NET Core содержит множество классов для работы с деревьями выражений. Полный список можно просмотреть в разделе System.Linq.Expressions. Вместо того чтобы знакомиться с этим списком, давайте разберемся, как строятся классы платформы.

Согласно принципам языка выражение — это блок кода, который выполняет вычисления и возвращает значение. Выражения могут быть очень простыми: константное выражение `1` всегда возвращает значение 1. Они могут быть более сложными: выражение `(-B + Math.Sqrt(B*B - 4 * A * C)) / (2 * A)` возвращает один корень квадратного уравнения (если у уравнения есть решение).

Все начинается с System.Linq.Expression

Одна из трудностей при работе с деревьями выражений заключается в том, что выражения различных типов могут использоваться в программах в самых разных местах. Возьмем для примера выражение присваивания. В правой его части может быть константа, переменная, выражение вызова метода или иной элемент. Такая гибкость языка означает, что при обходе дерева выражения в его узлах вам могут встретиться различные типы выражений. Поэтому, если есть возможность использовать базовый тип выражения, это простейший способ работы. Однако иногда этого недостаточно. На этот случай базовый класс Expression содержит свойство `NodeType`. Оно возвращает `ExpressionType` — перечисление возможных типов выражений. Если вы знаете тип узла выражения, то можете привести его к этому типу и выполнять с ним определенные действия. Можно выполнить поиск узлов определенных типов, а затем работать со свойствами данного типа выражения.

Например, приведенный ниже код печатает имя переменной для выражения доступа к переменной. В нем используется метод, заключающийся в проверке типа узла, приведении к выражению доступа к переменной и последующей проверке свойств определенного типа выражения.

```
Expression<Func<int, int>> addFive = (num) => num + 5;

if (addFive.NodeType == ExpressionType.Lambda)
{
    var lambdaExp = (LambdaExpression)addFive;

    var parameter = lambdaExp.Parameters.First();

    Console.WriteLine(parameter.Name);
    Console.WriteLine(parameter.Type);
}
```

Создание деревьев выражений

Класс `System.Linq.Expression` также содержит множество статических методов для создания выражений. Эти методы создают узел выражения с помощью аргументов, предоставленных для его дочерних элементов. Таким образом, выражение строится на основе листовых узлов. Например, следующий код создает выражение сложения:

```
// Addition is an add expression for "1 + 2"
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var addition = Expression.Add(one, two);
```

На этом простом примере видно, что в создании деревьев выражений и работе с ними задействовано множество типов. Такая сложность необходима для предоставления широких возможностей словаря C#.

Перемещение по API

Существуют типы узлов выражений, которые сопоставляются почти со всеми элементами синтаксиса языка C#. Каждый тип имеет определенные методы для конкретного типа элементов языка. Держать в голове приходится слишком многое. Вместо того чтобы запоминать все это, при работе с деревьями выражений можно применять описанные ниже приемы.

1. Просмотрите элементы перечисления `ExpressionType`, чтобы определить возможные узлы для анализа. Это очень полезно, если нужно выполнить обход дерева и изучить его.
2. Просмотрите статические члены класса `Expression`, которые служат для создания выражения. С помощью этих методов можно создать выражение любого типа на основе набора дочерних узлов.
3. Обратите внимание на класс `ExpressionVisitor`, который служит для создания измененного дерева выражения.

В каждой из этих трех областей доступны и другие возможности. Вы обязательно найдете то, что вам нужно, если начнете с этих трех основных шагов.

Следующий раздел: "Выполнение деревьев выражений"

Выполнение деревьев выражений

18.12.2021 • 4 minutes to read

Предыдущий раздел: "Типы платформ, поддерживающие деревья выражений"

Дерево выражения — это структура данных, представляющая некоторый код. Это не компилируемый и исполняемый код. Чтобы выполнить код .NET, представленный деревом выражения, необходимо преобразовать его в исполняемые инструкции на промежуточном языке (IL).

Преобразование лямбда-выражений в функции

Любое лямбда-выражение или тип, производный от лямбда-выражения, можно преобразовать в исполняемый код IL. Другие типы выражений невозможно преобразовать в код напрямую. На практике это ограничение редко имеет значение. Лямбда-выражения — это единственный тип выражений, которые может потребоваться выполнить путем преобразования в исполняемый код IL. (Например, представьте, есть ли смысл выполнять `ConstantExpression` напрямую. Может ли это быть полезным?) Любое дерево выражения, представляющее собой `LambdaExpression` или тип, производный от `LambdaExpression`, можно преобразовать в код IL. Тип выражения `Expression<TDelegate>` — единственный конкретный пример в библиотеках .NET Core. Он служит для представления выражения, которое соответствует любому типу делегата. Так как этот тип сопоставлен с типом делегата, .NET может проверить выражение и создать код IL для делегата, который соответствует сигнатуре лямбда-выражения.

В большинстве случаев при этом создается простое сопоставление между выражением и соответствующим ему делегатом. Например, дерево выражения, представленное `Expression<Func<int>>`, преобразуется в делегат типа `Func<int>`. Для лямбда-выражения с любым типом возвращаемого значения и списком аргументов существует тип делегата, который является целевым типом для исполняемого кода, представленного этим лямбда-выражением.

Тип `LambdaExpression` содержит члены `Compile` и `CompileToMethod`, которые используются для преобразования дерева выражения в исполняемый код. Метод `Compile` создает делегат. Метод `CompileToMethod` обновляет объект `MethodBuilder` с помощью кода IL, который представляет скомпилированные выходные данные дерева выражения. Обратите внимание на то, что метод `CompileToMethod` доступен только на полнофункциональной классической платформе, но не на платформе .NET Core.

При необходимости можно также предоставить объект `DebugInfoGenerator`, который будет получать символьную отладочную информацию для создаваемого объекта делегата. Это позволяет преобразовать дерево выражения в объект делегата и иметь полную отладочную информацию о созданном делегате.

Выражение преобразовывается в делегат с помощью следующего кода:

```
Expression<Func<int>> add = () => 1 + 2;
var func = add.Compile(); // Create Delegate
var answer = func(); // Invoke Delegate
Console.WriteLine(answer);
```

Обратите внимание на то, что тип делегата основан на типе выражения. Если требуется использовать объект делегата строго типизированным образом, необходимо знать тип возвращаемого значения и список аргументов. Метод `LambdaExpression.Compile()` возвращает тип `Delegate`. Его необходимо будет

привести к правильному типу делегата, чтобы средства времени компиляции могли проверить список аргументов или тип возвращаемого значения.

Выполнение и время существования

Код выполняется путем вызова делегата, созданного при вызове `LambdaExpression.Compile()`. Это можно увидеть в приведенном выше примере, где `add.Compile()` возвращает делегат. Вызов этого делегата путем вызова `func()` приводит к выполнению кода.

Делегат представляет код в дереве выражения. Вы можете сохранить дескриптор делегата и вызвать его позднее. Вам не нужно компилировать дерево выражения каждый раз, когда вы хотите выполнить представляемый им код. (Помните, что деревья выражений являются неизменяемыми и повторная компиляция того же дерева выражения приведет к созданию делегата, который выполняет этот же код.)

Не рекомендуется создавать более сложные механизмы перехвата с целью повышения производительности за счет уменьшения количества вызовов компиляции. Сравнение двух произвольных деревьев выражений с целью определить, представляют ли они один и тот же алгоритм, также требует времени. Скорее всего, время, сэкономленное за счет уменьшения количества вызовов `LambdaExpression.Compile()`, будет больше времени, затрачиваемого на выполнение кода, который определяет, приводят ли два разных дерева выражений к созданию одного и того же исполняемого кода.

Предупреждения

Компиляция лямбда-выражения в делегат и вызов этого делегата — одна из простейших операций, которые можно выполнять с деревом выражения. Однако даже в случае с такой простой операцией следует учитывать ряд моментов.

Лямбда-выражения создают замыкания для всех локальных переменных, на которые в них имеются ссылки. Необходимо обеспечить возможность использования всех переменных, которые будут входить в делегат, в месте вызова `Compile` и при выполнении итогового делегата.

Как правило, эту задачу решает компилятор. Однако если выражение обращается к переменной, которая реализует `IDisposable`, в коде может быть удален объект, который все еще удерживается деревом выражения.

Например, следующий код выполняется нормально, так как `int` не реализует `IDisposable`:

```
private static Func<int, int> CreateBoundFunc()
{
    var constant = 5; // constant is captured by the expression tree
    Expression<Func<int, int>> expression = (b) => constant + b;
    var rVal = expression.Compile();
    return rVal;
}
```

Делегат захватил ссылку на локальную переменную `constant`. Доступ к этой переменной осуществляется позднее, когда выполняется функция, возвращаемая `CreateBoundFunc`.

Однако следует рассмотреть также следующий (несколько искусственный) класс, который реализует `IDisposable`:

```
public class Resource : IDisposable
{
    private bool isDisposed = false;
    public int Argument
    {
        get
        {
            if (!isDisposed)
                return 5;
            else throw new ObjectDisposedException("Resource");
        }
    }

    public void Dispose()
    {
        isDisposed = true;
    }
}
```

Если он используется в выражении, как показано ниже, то при выполнении кода, к которому обращается свойство `Resource.Argument`, возникнет исключение `ObjectDisposedException`:

```
private static Func<int, int> CreateBoundResource()
{
    using (var constant = new Resource()) // constant is captured by the expression tree
    {
        Expression<Func<int, int>> expression = (b) => constant.Argument + b;
        var rVal = expression.Compile();
        return rVal;
    }
}
```

Делегат, возвращенный этим методом, замкнулся на объекте `constant`, который был ликвидирован. (Причиной ликвидации является то, что он был объявлен в операторе `using`.)

Теперь при выполнении делегата, возвращаемого этим методом, будет возникать исключение `ObjectDisposedException`.

Может показаться странным, что ошибка времени выполнения представляет конструкцию времени компиляции, но таковы особенности работы с деревьями выражений.

Эта проблема может принимать разные формы, поэтому трудно дать общую рекомендацию по тому, как ее избежать. Проявляйте осторожность в отношении доступа к локальным переменным при определении выражений и в отношении доступа к состоянию в текущем объекте (представленном `this`) при создании дерева выражения, которое может возвращаться открытым интерфейсом API.

Код выражения может ссылаться на методы или свойства в других сборках. Эти сборки должны быть доступны при определении выражения, его компиляции и вызове итогового делегата. В случае их отсутствия будет возникать исключение `ReferencedAssemblyNotFoundException`.

Сводка

Деревья выражений, представляющие лямбда-выражения, можно компилировать с целью создания делегатов, которые можно выполнять. Таким образом обеспечивается механизм выполнения кода, представленного деревом выражения.

Дерево выражения представляет код, который будет выполняться для определенной конструкции, которую вы создаете. При условии, что среда, в которой компилируется и выполняется код, соответствует среде, в которой создается выражение, все работает правильно. В противном случае

ошибки очень предсказуемы и выявляются при проведении первых тестов кода, в котором используются деревья выражений.

Следующий раздел: "Интерпретация выражений"

Интерпретация выражений

18.12.2021 • 13 minutes to read

Предыдущий раздел — "Выполнение выражений"

Теперь напишем код для проверки структуры *дерева выражения*. Каждый узел в дереве выражения будет объектом класса, производного от класса `Expression`.

Такая структура позволяет посещать все узлы в дереве выражения в виде относительно простой рекурсивной операции. Общая стратегия заключается в том, что нужно начать с корневого узла и определить его тип.

Если тип узла имеет дочерние элементы, их нужно посетить в рекурсивном режиме. На каждом дочернем узле повторите процесс, используемый на корневом узле: определите тип и, если тип имеет дочерние элементы, пройдите по каждому из них.

Проверка выражения без дочерних узлов

Начнем с посещения каждого узла в простом дереве выражения. Ниже приведен код, который создает константное выражение, а затем проверяет его свойства:

```
var constant = Expression.Constant(24, typeof(int));

Console.WriteLine($"This is a/an {constant.NodeType} expression type");
Console.WriteLine($"The type of the constant value is {constant.Type}");
Console.WriteLine($"The value of the constant value is {constant.Value}");
```

Будут выведены следующие результаты:

```
This is an Constant expression type
The type of the constant value is System.Int32
The value of the constant value is 24
```

Теперь напишем код, который будет проверять это выражение и записывать некоторые важные свойства. Вот этот код:

Проверка простого выражения сложения

Начнем с примера сложения из вводной части к этому разделу.

```
Expression<Func<int>> sum = () => 1 + 2;
```

> Здесь не используется `var` для объявления этого дерева выражения, так как это невозможно, поскольку правая часть назначения неявно типизирована.

`LambdaExpression` — это корневой узел. Для получения нужного кода в правой части оператора `=>` следует найти один из дочерних элементов `LambdaExpression`. Это необходимо сделать со всеми выражениями в этом разделе. Благодаря родительскому узлу можно найти возвращаемый тип `LambdaExpression`.

Чтобы проверить каждый узел в этом выражении, необходимо рекурсивно пройти через несколько узлов. Вот простая первая реализация:

```
Expression<Func<int, int, int>> addition = (a, b) => a + b;

Console.WriteLine($"This expression is a {addition.NodeType} expression type");
Console.WriteLine($"The name of the lambda is {((addition.Name == null) ? "<null>" : addition.Name)}");
Console.WriteLine($"The return type is {addition.ReturnType.ToString()}");
Console.WriteLine($"The expression has {addition.Parameters.Count} arguments. They are:");
foreach(var argumentExpression in addition.Parameters)
{
    Console.WriteLine($"\tParameter Type: {argumentExpression.Type.ToString()}, Name:
{argumentExpression.Name}");
}

var additionBody = (BinaryExpression)addition.Body;
Console.WriteLine($"The body is a {additionBody.NodeType} expression");
Console.WriteLine($"The left side is a {additionBody.Left.NodeType} expression");
var left = (ParameterExpression)additionBody.Left;
Console.WriteLine($"\tParameter Type: {left.Type.ToString()}, Name: {left.Name}");
Console.WriteLine($"The right side is a {additionBody.Right.NodeType} expression");
var right= (ParameterExpression)additionBody.Right;
Console.WriteLine($"\tParameter Type: {right.Type.ToString()}, Name: {right.Name}");
```

Результатом является следующее:

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 arguments. They are:
        Parameter Type: System.Int32, Name: a
        Parameter Type: System.Int32, Name: b
The body is a/an Add expression
The left side is a Parameter expression
        Parameter Type: System.Int32, Name: a
The right side is a Parameter expression
        Parameter Type: System.Int32, Name: b
```

В примере кода выше можно заметить много повторений. Очистим код и создадим посетитель узлов выражения более общего назначения. Для этого нужно написать рекурсивный алгоритм. Любой узел может иметь тип с дочерними элементами. Любой узел, имеющий дочерние элементы, требует посетить эти дочерние элементы и определить тип узла. Вот очищенная версия, где используется рекурсия для посещения операций сложения:

```
// Base Visitor class:
public abstract class Visitor
{
    private readonly Expression node;

    protected Visitor(Expression node)
    {
        this.node = node;
    }

    public abstract void Visit(string prefix);

    public ExpressionType NodeType => this.node.NodeType;
    public static Visitor CreateFromExpression(Expression node)
    {
        switch(node.NodeType)
        {
            case ExpressionType.Constant:
                return new ConstantVisitor((ConstantExpression)node);
```

```
                return new ConstantVisitor((ConstantExpression)node);
            case ExpressionType.Lambda:
                return new LambdaVisitor((LambdaExpression)node);
            case ExpressionType.Parameter:
                return new ParameterVisitor((ParameterExpression)node);
            case ExpressionType.Add:
                return new BinaryVisitor((BinaryExpression)node);
            default:
                Console.Error.WriteLine($"Node not processed yet: {node.NodeType}");
                return default(Visitor);
        }
    }
}

// Lambda Visitor
public class LambdaVisitor : Visitor
{
    private readonly LambdaExpression node;
    public LambdaVisitor(LambdaExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType} expression type");
        Console.WriteLine($"{prefix}The name of the lambda is {((node.Name == null) ? "<null>" :
node.Name)}");
        Console.WriteLine($"{prefix}The return type is {node.ReturnType.ToString()}");
        Console.WriteLine($"{prefix}The expression has {node.Parameters.Count} argument(s). They are:");
        // Visit each parameter:
        foreach (var argumentExpression in node.Parameters)
        {
            var argumentVisitor = Visitor.CreateFromExpression(argumentExpression);
            argumentVisitor.Visit(prefix + "\t");
        }
        Console.WriteLine($"{prefix}The expression body is:");
        // Visit the body:
        var bodyVisitor = Visitor.CreateFromExpression(node.Body);
        bodyVisitor.Visit(prefix + "\t");
    }
}

// Binary Expression Visitor:
public class BinaryVisitor : Visitor
{
    private readonly BinaryExpression node;
    public BinaryVisitor(BinaryExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This binary expression is a {NodeType} expression");
        var left = Visitor.CreateFromExpression(node.Left);
        Console.WriteLine($"{prefix}The Left argument is:");
        left.Visit(prefix + "\t");
        var right = Visitor.CreateFromExpression(node.Right);
        Console.WriteLine($"{prefix}The Right argument is:");
        right.Visit(prefix + "\t");
    }
}

// Parameter visitor:
public class ParameterVisitor : Visitor
{
    private readonly ParameterExpression node;
    public ParameterVisitor(ParameterExpression node) : base(node)
    {
```

```
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This is an {NodeType} expression type");
        Console.WriteLine($"{prefix}Type: {node.Type.ToString()}, Name: {node.Name}, ByRef:
{node.IsByRef}");
    }
}

// Constant visitor:
public class ConstantVisitor : Visitor
{
    private readonly ConstantExpression node;
    public ConstantVisitor(ConstantExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This is an {NodeType} expression type");
        Console.WriteLine($"{prefix}The type of the constant value is {node.Type}");
        Console.WriteLine($"{prefix}The value of the constant value is {node.Value}");
    }
}
```

Этот алгоритм является основой для алгоритма, который может посещать любой произвольный `LambdaExpression`. Существует множество брешей, а именно: созданный код выполняет поиск совсем небольшого примера возможных наборов узлов дерева выражения, который он может найти. Однако в результатах вы по-прежнему сможете находить полезные данные. (По умолчанию при обнаружении нового типа узла метод `Visitor.CreateFromExpression` выводит сообщение на консоль ошибок. Таким образом, вы знаете, что нужно добавить новый тип выражения.)

При запуске этого посетителя в выражении сложения, показанном выше, вы получите следующие выходные данные:

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: a, ByRef: False
        This is an Parameter expression type
        Type: System.Int32, Name: b, ByRef: False
The expression body is:
        This binary expression is a Add expression
        The Left argument is:
                This is an Parameter expression type
                Type: System.Int32, Name: a, ByRef: False
        The Right argument is:
                This is an Parameter expression type
                Type: System.Int32, Name: b, ByRef: False
```

Теперь, когда вы создали более общую реализацию посетителя, можно посещать и обрабатывать гораздо больше разных типов выражений.

Проверка выражения сложения с несколькими уровнями

Рассмотрим более сложный пример, но по-прежнему ограничим типы узлов только сложением:

```
Expression<Func<int>> sum = () => 1 + 2 + 3 + 4;
```

Перед запуском примера в алгоритме посетителя попробуйте представить, какие выходные данные могут быть получены. Помните, что оператор `+` является *бинарным*: он должна иметь два дочерних элемента, представляющих левый и правый операнды. Существует несколько правильных возможных способов построения дерева:

```
Expression<Func<int>> sum1 = () => 1 + (2 + (3 + 4));
Expression<Func<int>> sum2 = () => ((1 + 2) + 3) + 4;

Expression<Func<int>> sum3 = () => (1 + 2) + (3 + 4);
Expression<Func<int>> sum4 = () => 1 + ((2 + 3) + 4);
Expression<Func<int>> sum5 = () => (1 + (2 + 3)) + 4;
```

Здесь выполнено разделение на два возможных ответа для выделения наиболее перспективного из них. Первый представляет *правоассоциативные* выражения. Второй представляет *левоассоциативные* выражения. Преимущества этих двух форматов в том, что формат масштабируется до любого произвольного числа в выражениях сложения.

При запуске этого выражения через посетитель вы увидите эти выходные данные, проверяющие, что выражение простого сложения является *левоассоциативным*.

Чтобы выполнить этот пример и увидеть полное дерево выражения, пришлось внести одно изменение в исходное дерево выражения. Если дерево выражения содержит все константы, результирующее дерево просто содержит константное значение, равное `10`. Компилятор выполняет все операции сложения и сокращает выражение до его простейшей формы. Для просмотра исходного дерева достаточно добавить одну переменную в выражение:

```
Expression<Func<int, int>> sum = (a) => 1 + a + 3 + 4;
```

Создайте посетитель для этой суммы и запустите его. После этого вы увидите эти выходные данные:

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 1 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: a, ByRef: False
The expression body is:
        This binary expression is a Add expression
        The Left argument is:
                This binary expression is a Add expression
                The Left argument is:
                        This binary expression is a Add expression
                        The Left argument is:
                                This is an Constant expression type
                                The type of the constant value is System.Int32
                                The value of the constant value is 1
                        The Right argument is:
                                This is an Parameter expression type
                                Type: System.Int32, Name: a, ByRef: False
                The Right argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
                        The value of the constant value is 3
        The Right argument is:
                This is an Constant expression type
                The type of the constant value is System.Int32
                The value of the constant value is 4
```

С помощью кода посетителя можно также выполнить другие примеры и просмотреть, какое дерево он представляет. Ниже приведен пример выражения `sum3` выше (с дополнительным параметром, чтобы компилятор не смог вычислить константу):

```
Expression<Func<int, int, int>> sum3 = (a, b) => (1 + a) + (3 + b);
```

Вот результат посетителя:

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: a, ByRef: False
        This is an Parameter expression type
        Type: System.Int32, Name: b, ByRef: False
The expression body is:
        This binary expression is a Add expression
        The Left argument is:
                This binary expression is a Add expression
                The Left argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
                        The value of the constant value is 1
                The Right argument is:
                        This is an Parameter expression type
                        Type: System.Int32, Name: a, ByRef: False
        The Right argument is:
                This binary expression is a Add expression
                The Left argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
                        The value of the constant value is 3
                The Right argument is:
                        This is an Parameter expression type
                        Type: System.Int32, Name: b, ByRef: False
```

Обратите внимание, что скобки не являются частью выходных данных. В дереве выражения нет узлов, представляющих круглые скобки во входном выражении. Структура дерева выражения содержит всю информацию, необходимую для сообщения о приоритете.

Расширение примера

В этом примере используются только самые элементарные деревья выражений. Код, который вы видели в этом разделе, обрабатывает только целочисленные константы и двоичный оператор `+`. Обратимся к последнему примеру и изменим посетитель для обработки более сложного выражения. Сделаем так, чтобы он работал следующим образом:

```
Expression<Func<int, int>> factorial = (n) =>
    n == 0 ?
    1 :
    Enumerable.Range(1, n).Aggregate((product, factor) => product * factor);
```

Этот код представляет одну из возможных реализаций для математической функции — *факториала*. В способе написания кода выделяется два ограничения для создания деревьев выражений путем присваивания лямбда-выражений выражениям. Во-первых, лямбды операторов не допускаются. Это означает, что нельзя использовать циклы, блоки, операторы if/else и другие структуры управления, распространенные в C#. Можно использовать только выражения. Во-вторых, нельзя выполнить рекурсивный вызов того же выражения. Это можно было бы сделать, если бы уже имелся делегат, но его нельзя вызвать в форме дерева выражения. В разделе о построении деревьев выражений вы узнаете о том, как преодолеть эти ограничения.

В этом выражении вы встретите узлы всех указанных далее типов:

1. Equal (двоичное выражение)
2. Multiply (двоичное выражение)
3. Conditional (выражение ? :)

4. Выражение вызова метода (вызов `Range()` и `Aggregate()`)

Одним из способов изменения алгоритма посетителя является поддержка его выполнения и запись типа узла при каждом достижении предложения `default`. После нескольких повторений вы просмотрите все возможные узлы. Итак, у вас есть все, что нужно. Результат будет выглядеть примерно следующим образом:

```
public static Visitor CreateFromExpression(Expression node)
{
    switch(node.NodeType)
    {
        case ExpressionType.Constant:
            return new ConstantVisitor((ConstantExpression)node);
        case ExpressionType.Lambda:
            return new LambdaVisitor((LambdaExpression)node);
        case ExpressionType.Parameter:
            return new ParameterVisitor((ParameterExpression)node);
        case ExpressionType.Add:
        case ExpressionType.Equal:
        case ExpressionType.Multiply:
            return new BinaryVisitor((BinaryExpression)node);
        case ExpressionType.Conditional:
            return new ConditionalVisitor((ConditionalExpression)node);
        case ExpressionType.Call:
            return new MethodCallVisitor((MethodCallExpression)node);
        default:
            Console.Error.WriteLine($"Node not processed yet: {node.NodeType}");
            return default(Visitor);
    }
}
```

ConditionalVisitor и MethodCallVisitor обрабатывают эти два узла:

```
public class ConditionalVisitor : Visitor
{
    private readonly ConditionalExpression node;
    public ConditionalVisitor(ConditionalExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType} expression");
        var testVisitor = Visitor.CreateFromExpression(node.Test);
        Console.WriteLine($"{prefix}The Test for this expression is:");
        testVisitor.Visit(prefix + "\t");
        var trueVisitor = Visitor.CreateFromExpression(node.IfTrue);
        Console.WriteLine($"{prefix}The True clause for this expression is:");
        trueVisitor.Visit(prefix + "\t");
        var falseVisitor = Visitor.CreateFromExpression(node.IfFalse);
        Console.WriteLine($"{prefix}The False clause for this expression is:");
        falseVisitor.Visit(prefix + "\t");
    }
}

public class MethodCallVisitor : Visitor
{
    private readonly MethodCallExpression node;
    public MethodCallVisitor(MethodCallExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType} expression");
        if (node.Object == null)
            Console.WriteLine($"{prefix}This is a static method call");
        else
        {
            Console.WriteLine($"{prefix}The receiver (this) is:");
            var receiverVisitor = Visitor.CreateFromExpression(node.Object);
            receiverVisitor.Visit(prefix + "\t");
        }

        var methodInfo = node.Method;
        Console.WriteLine($"{prefix}The method name is {methodInfo.DeclaringType}.{methodInfo.Name}");
        // There is more here, like generic arguments, and so on.
        Console.WriteLine($"{prefix}The Arguments are:");
        foreach(var arg in node.Arguments)
        {
            var argVisitor = Visitor.CreateFromExpression(arg);
            argVisitor.Visit(prefix + "\t");
        }
    }
}
```

Будет получен следующий результат для дерева выражения:

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 1 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: n, ByRef: False
The expression body is:
        This expression is a Conditional expression
        The Test for this expression is:
                This binary expression is a Equal expression
                The Left argument is:
                        This is an Parameter expression type
                        Type: System.Int32, Name: n, ByRef: False
                The Right argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
                        The value of the constant value is 0
        The True clause for this expression is:
                This is an Constant expression type
                The type of the constant value is System.Int32
                The value of the constant value is 1
        The False clause for this expression is:
                This expression is a Call expression
                This is a static method call
                The method name is System.Linq.Enumerable.Aggregate
                The Arguments are:
                        This expression is a Call expression
                        This is a static method call
                        The method name is System.Linq.Enumerable.Range
                        The Arguments are:
                                This is an Constant expression type
                                The type of the constant value is System.Int32
                                The value of the constant value is 1
                                This is an Parameter expression type
                                Type: System.Int32, Name: n, ByRef: False
                        This expression is a Lambda expression type
                        The name of the lambda is <null>
                        The return type is System.Int32
                        The expression has 2 arguments. They are:
                                This is an Parameter expression type
                                Type: System.Int32, Name: product, ByRef: False
                                This is an Parameter expression type
                                Type: System.Int32, Name: factor, ByRef: False
                        The expression body is:
                                This binary expression is a Multiply expression
                                The Left argument is:
                                        This is an Parameter expression type
                                        Type: System.Int32, Name: product, ByRef: False
                                The Right argument is:
                                        This is an Parameter expression type
                                        Type: System.Int32, Name: factor, ByRef: False
```

Расширение примера библиотеки

В примерах в этом разделе демонстрируются основные способы посещения и изучения узлов в дереве выражения. Вы узнали о целом ряде действий, которые может потребоваться выполнить для реализации основных задач, связанных с посещением узлов в дереве выражения.

Во-первых, посетители могут обрабатывать только константы, которые являются целыми числами. Константные значения могут иметь любой другой числовой тип, и язык C# поддерживает преобразования и повышения уровней этих типов. Все эти возможности будут отражены в более надежной версии этого кода.

Даже последний пример распознает подмножество возможных типов узлов. В него можно по-прежнему

вводить множество выражений, которые приведут к его сбою. Полная реализация включена в стандартную библиотеку .NET с именем ExpressionVisitor и может обрабатывать все возможные типы узлов.

И, наконец библиотека, которая использовалась в этой статье, была создана для демонстрации и обучения. Она не оптимизирована. Она была написана для четкого представления структур и для выделения методов, использовавшихся для посещения узлов и анализа их содержимого. В рабочей реализации производительности будет уделено гораздо больше внимания.

Даже с этими ограничениями вы будете обладать достаточными знаниями для написания алгоритмов по чтению и анализу деревьев выражений.

Следующий раздел — "Построение выражений"

Построение деревьев выражений

18.12.2021 • 4 minutes to read

Предыдущий раздел — "Интерпретация выражений"

Все деревья выражений, которые вы уже видели, были созданы с помощью компилятора C#. Все, что нужно было сделать, — это создать лямбда-выражение, которое было присвоено переменной, типизированной как `Expression<Func<T>>`, или имеющий схожий тип. Это не единственный способ создания дерева выражений. Во многих случаях может понадобиться создать выражение в памяти во время выполнения.

Процесс построения деревьев выражений усложняется тем фактом, что эти деревья выражений являются неизменяемыми. Неизменяемость означает, что дерево необходимо создать, начиная с листьев и заканчивая корнем. API-интерфейсы, которые будут использоваться для создания деревьев выражений, отражают этот факт: методы, используемые для создания узла, будут принимать все его дочерние узлы в качестве аргументов. Рассмотрим несколько примеров, демонстрирующих способы создания.

Создание узлов

Начнем с относительно простого варианта. Мы будем использовать выражение сложения, которое применялось в этих разделах:

```
Expression<Func<int>> sum = () => 1 + 2;
```

Чтобы построить дерево выражения, необходимо создать конечные узлы. Конечные узлы являются константами, поэтому для их создания можно использовать метод `Expression.Constant`:

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
```

Затем вы создадите выражение сложения:

```
var addition = Expression.Add(one, two);
```

После этого можно создать лямбда-выражение:

```
var lambda = Expression.Lambda(addition);
```

Это очень простое лямбда-выражение, так как оно не содержит аргументов. Далее в этом разделе вы узнаете, как сопоставлять аргументы с параметрами и создавать более сложные выражения.

Для выражений, которые так же просты, как и это, можно совместить все вызовы в одну инструкцию:

```
var lambda = Expression.Lambda(
    Expression.Add(
        Expression.Constant(1, typeof(int)),
        Expression.Constant(2, typeof(int))
    )
);
```

Построение дерева

Это основные принципы построения дерева выражения в памяти. Более сложные деревья обычно характеризуются большим количеством типов узлов и количеством самих узлов. Рассмотрим еще один пример и взглянем на два дополнительных типа узлов, которые обычно создаются при построении деревьев выражений: узлы аргументов и узлы вызовов методов.

Построим дерево выражения, чтобы создать это выражение:

```
Expression<Func<double, double, double>> distanceCalc =
    (x, y) => Math.Sqrt(x * x + y * y);
```

Начнем с создания выражений параметров для `x` и `y`:

```
var xParameter = Expression.Parameter(typeof(double), "x");
var yParameter = Expression.Parameter(typeof(double), "y");
```

Создание выражений умножения и сложения выполняется по шаблону, который вы уже видели:

```
var xSquared = Expression.Multiply(xParameter, xParameter);
var ySquared = Expression.Multiply(yParameter, yParameter);
var sum = Expression.Add(xSquared, ySquared);
```

Затем необходимо создать выражение вызова метода для вызова `Math.Sqrt`.

```
var sqrtMethod = typeof(Math).GetMethod("Sqrt", new[] { typeof(double) });
var distance = Expression.Call(sqrtMethod, sum);
```

И, наконец, нужно поместить вызов метода в лямбда-выражение и определить аргументы для лямбда-выражения:

```
var distanceLambda = Expression.Lambda(
    distance,
    xParameter,
    yParameter);
```

В этом более сложном примере используется несколько дополнительных приемов, зачастую необходимых при созданий деревьев выражений.

Во-первых, необходимо создать объекты, представляющие параметры или локальные переменные, перед их использованием. Созданные объекты можно использовать в дереве выражения там, где это необходимо.

Во-вторых, необходимо использовать подмножество API-интерфейсов отражения для создания объекта `MethodInfo`, чтобы можно было построить дерево выражения для получения доступа к этому методу. Необходимо ограничить подмножество API-интерфейсов отражения, доступных на платформе .NET Core. Эти способы можно будет использовать для других деревьев выражений.

Всестороннее создание кода

С помощью этих API вы можете создавать что угодно. Однако чем более сложное дерево выражения вы хотите построить, тем более сложными являются задачи чтения кода и управления им.

Давайте создадим дерево выражения, которое эквивалентно этому коду:

```
Func<int, int> factorialFunc = (n) =>
{
    var res = 1;
    while (n > 1)
    {
        res = res * n;
        n--;
    }
    return res;
};
```

Обратите внимание, что было построено не дерево выражения, а просто делегат. С помощью класса `Expression` нельзя создать лямбды операторов. Ниже приведен код, необходимый для формирования тех же функциональных возможностей. Он усложняется тем фактом, что API для создания цикла `while` не существует. Вместо этого необходимо создать цикл, который содержит проверку условий, и целевой объект метки для разрыва цикла.

```
var nArgument = Expression.Parameter(typeof(int), "n");
var result = Expression.Variable(typeof(int), "result");

// Creating a label that represents the return value
LabelTarget label = Expression.Label(typeof(int));

var initializeResult = Expression.Assign(result, Expression.Constant(1));

// This is the inner block that performs the multiplication,
// and decrements the value of 'n'
var block = Expression.Block(
    Expression.Assign(result,
        Expression.Multiply(result, nArgument)),
    Expression.PostDecrementAssign(nArgument)
);

// Creating a method body.
BlockExpression body = Expression.Block(
    new[] { result },
    initializeResult,
    Expression.Loop(
        Expression.IfThenElse(
            Expression.GreaterThan(nArgument, Expression.Constant(1)),
            block,
            Expression.Break(label, result)
        ),
        label
    )
);
```

Код для создания дерева выражения для функции факториала немного больше, более сложен и содержит операторы меток и прерываний и другие элементы, использования которых хотелось бы избежать при выполнении ежедневных задач по созданию кода.

Для работы в этом разделе также обновлен код посетителя для просмотра каждого узла в этом дереве выражения и записи сведений об узлах, созданных в этом примере. Просмотреть или скачать пример кода можно в репозитории dotnet/docs на сайте GitHub. Поэкспериментируйте со сборкой и использованием примеров кода. Инструкции по загрузке см. в разделе Просмотр и скачивание примеров.

Изучение API-интерфейсов

Возможности навигации по API-интерфейсам дерева выражений усложнены в .NET Core, но это нормально. Их назначение довольно сложное — написание кода, который создает код во время

выполнения. Они с трудом обеспечивают баланс между поддержкой всех структур управления, доступных на языке C#, и сохранением минимально возможного размера контактной зоны API-интерфейсов. Этот баланс означает, что многие структуры управления представлены не своими конструкциями C#, а конструкциями, которые представляют базовую логику, создаваемую компилятором из этих конструкций более высокого уровня.

Кроме того, в настоящее время не существует выражений C#, которые создаются напрямую с помощью методов класса `Expression`. Как правило, это будут новые операторы и выражения, добавленные в C# 5 и C# 6. (Например, выражения `async` не могут быть созданы, и новый оператор `?.` нельзя создать напрямую.)

Следующий раздел — "Преобразование выражений"

Преобразование деревьев выражений

18.12.2021 • 5 minutes to read

Предыдущий раздел — "Построение выражений"

В этом заключительном разделе вы узнаете, как перейти к каждому узлу в дереве выражения при создании измененной копии этого дерева выражений. В разделе представлены методы, которые будут использоваться в двух важных сценариях. Первый — понимание алгоритмов, выраженных деревом выражения, для преобразования в другую среду. Второй — необходимость изменить созданный алгоритм. Сюда можно отнести необходимость добавить ведение журнала и осуществлять перехват вызовов методов и отслеживать их, а также другие задачи.

Преобразование — это обход

Код, создаваемый для преобразования дерева выражения, является расширением кода, который используется для обхода всех узлов в дереве. Во время преобразования дерева выражения вы выполняете обход всех узлов и, таким образом, построение нового дерева. Новое дерево может содержать ссылки на исходные узлы или новые узлы, которые были помещены в дерево.

Рассмотрим это на примере, выполнив обход дерева выражения и создав новое дерево, заменяя в нем несколько узлов. В этом примере мы заменим любую константу константой, которая в десять раз больше. В противном случае мы оставим дерево выражения без изменений. Вместо того чтобы считывать значение константы и заменять ее новой константой, мы осуществим такую подстановку, заменив узел константы новым узлом, который выполняет умножение.

Итак, вы находите узел константы, создаете новый узел умножения, дочерний элемент которого является исходной константой, и константу `10`:

```
private static Expression ReplaceNodes(Expression original)
{
    if (original.NodeType == ExpressionType.Constant)
    {
        return Expression.Multiply(original, Expression.Constant(10));
    }
    else if (original.NodeType == ExpressionType.Add)
    {
        var binaryExpression = (BinaryExpression)original;
        return Expression.Add(
            ReplaceNodes(binaryExpression.Left),
            ReplaceNodes(binaryExpression.Right));
    }
    return original;
}
```

При замене исходного узла новым узлом формируется новое дерево, содержащее наши изменения. Мы можем проверить это, скомпилировав и выполнив измененное дерево.

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var addition = Expression.Add(one, two);
var sum = ReplaceNodes(addition);
var executableFunc = Expression.Lambda(sum);

var func = (Func<int>)executableFunc.Compile();
var answer = func();
Console.WriteLine(answer);
```

Создание нового дерева представляет собой сочетание обхода узлов в существующем дереве и создания новых узлов и вставки их в дерево.

Этот пример демонстрирует значимость неизменяемых деревьев выражений. Обратите внимание, что созданное нами новое дерево содержит сочетание вновь созданных узлов и узлов из существующего дерева. Это безопасно, поскольку узлы в существующем дереве изменить нельзя. Это позволяет существенно сократить потребление памяти. Одни и те же узлы можно использовать в одном дереве или в нескольких деревьях выражений. Поскольку узлы нельзя изменить, один узел можно при необходимости использовать повторно.

Обход и выполнение добавления

Давайте проверим это, создав второй посетитель, который обходит дерево дополнительных узлов и вычисляет результат. Для этого вы можете просто внести пару изменений в уже готовый посетитель. В этой новой версии посетитель вернет частичную сумму операции сложения до этой точки. Для константного выражения это будет просто значением константного выражения. Для выражения добавления результатом будет сумма левого и правого операндов, поскольку выполняется обход этих деревьев.

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var three= Expression.Constant(3, typeof(int));
var four = Expression.Constant(4, typeof(int));
var addition = Expression.Add(one, two);
var add2 = Expression.Add(three, four);
var sum = Expression.Add(addition, add2);

// Declare the delegate, so we can call it
// from itself recursively:
Func<Expression, int> aggregate = null;
// Aggregate, return constants, or the sum of the left and right operand.
// Major simplification: Assume every binary expression is an addition.
aggregate = (exp) =>
    exp.NodeType == ExpressionType.Constant ?
    (int)((ConstantExpression)exp).Value :
    aggregate(((BinaryExpression)exp).Left) + aggregate(((BinaryExpression)exp).Right);

var theSum = aggregate(sum);
Console.WriteLine(theSum);
```

Нам потребуется написать небольшой фрагмент кода, однако основные принципы очень просты. Этот код обходит дочерние элементы при поиске в глубину. При обнаружении узла константы посетитель возвращает значение этой константы. После обхода обоих дочерних элементов для них вычисляется сумма для этого поддерева. Теперь добавленный узел может вычислить свою сумму. После обхода всех узлов в дереве выражений будут вычислена сумма. Вы можете отслеживать выполнение, запустив пример в отладчике с трассировкой выполнения.

Мы можем упростить трассировку анализа узлов и вычисления суммы путем обхода дерева. Вот

обновленная версия метода агрегирования, который включает совсем немного данных трассировки:

```
private static int Aggregate(Expression exp)
{
    if (exp.NodeType == ExpressionType.Constant)
    {
        var constantExp = (ConstantExpression)exp;
        Console.Error.WriteLine($"Found Constant: {constantExp.Value}");
        return (int)constantExp.Value;
    }
    else if (exp.NodeType == ExpressionType.Add)
    {
        var addExp = (BinaryExpression)exp;
        Console.Error.WriteLine("Found Addition Expression");
        Console.Error.WriteLine("Computing Left node");
        var leftOperand = Aggregate(addExp.Left);
        Console.Error.WriteLine($"Left is: {leftOperand}");
        Console.Error.WriteLine("Computing Right node");
        var rightOperand = Aggregate(addExp.Right);
        Console.Error.WriteLine($"Right is: {rightOperand}");
        var sum = leftOperand + rightOperand;
        Console.Error.WriteLine($"Computed sum: {sum}");
        return sum;
    }
    else throw new NotSupportedException("Haven't written this yet");
}
```

Если выполнить его в том же выражении, получаются следующие выходные данные:

```
10
Found Addition Expression
Computing Left node
Found Addition Expression
Computing Left node
Found Constant: 1
Left is: 1
Computing Right node
Found Constant: 2
Right is: 2
Computed sum: 3
Left is: 3
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 3
Left is: 3
Computing Right node
Found Constant: 4
Right is: 4
Computed sum: 7
Right is: 7
Computed sum: 10
10
```

Проследите выходные данные по приведенному выше коду. Вы наверняка поймете, каким образом код обходит каждый узел и вычисляет сумму по мере обхода дерева и нахождения суммы.

Теперь рассмотрим другое выполнение, где выражение задается объектом `sum1`:

```
Expression<Func<int> sum1 = () => 1 + (2 + (3 + 4));
```

Ниже вы видите выходные данные проверки этого выражения:

```
Found Addition Expression
Computing Left node
Found Constant: 1
Left is: 1
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 2
Left is: 2
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 3
Left is: 3
Computing Right node
Found Constant: 4
Right is: 4
Computed sum: 7
Right is: 7
Computed sum: 9
Right is: 9
Computed sum: 10
10
```

Хотя окончательный ответ совпадает, способ обхода дерева совершенно другой. Узлы обходятся в другом порядке, так как дерево было создано с помощью других начальных операций.

Дополнительные сведения

В этом примере показана малая часть кода, который необходимо создать для обхода и интерпретации алгоритмов, представляемых деревом выражения. Более полное рассмотрение всех действий, необходимых для создания библиотеки общего назначения, которая преобразует деревья выражений в другой язык, см. в этой серии публикаций Мэтта Уоррена (Matt Warren). Здесь преобразование кода, который можно найти в дереве выражения, рассматривается гораздо подробнее.

Надеюсь, теперь вы оценили эффективность и удобство деревьев выражений. Вы можете проверить фрагмент кода, внести в него необходимые изменения и выполнить измененную версию. Поскольку деревья выражений являются неизменяемыми, вы можете создавать новые деревья на основе компонентов существующих деревьев. Это позволяет свести к минимуму потребление памяти для создания измененных деревьев выражений.

Далее — "Заключение"

Сводный обзор деревьев выражений

18.12.2021 • 2 minutes to read

Предыдущий раздел — "Преобразование выражений"

В этой серии мы продемонстрировали, как можно использовать *деревья выражений* для создания динамических программ, которые интерпретируют код как данные и создают новые функции на основе этого кода.

Вы можете изучить деревья выражений, чтобы понять суть алгоритма. Но код можно не только просматривать. Вы можете создавать новые деревья выражений, представляющие измененные версии исходного кода.

Кроме того, деревья выражений можно использовать для анализа алгоритма и его преобразования в другой язык или среду.

Ограничения

Существуют некоторые новые элементы языка C#, которые не подходят для преобразования в деревья выражений. Деревья выражений не могут содержать выражения `await` или лямбда-выражения `async`. Многие из функций, добавленных в версии C# 6, не отображаются точно так, как они написаны, в деревьях выражений. Новые функции будут предоставляться в деревьях выражений в эквивалентном синтаксисе прежних версий. Однако это не так ограничивает ваши возможности, как может показаться. На самом деле это означает, что код, который интерпретирует деревья выражений, сможет успешно обрабатывать новые возможности языка, которые будут добавлены в будущем.

Даже с этими ограничениями деревья выражений позволяют создавать динамические алгоритмы, основанные на интерпретации и изменении кода, представленного в виде структуры данных. Это очень эффективный инструмент и один из компонентов экосистемы .NET, которая обеспечивает преимущества широкого набора библиотек, таких как Entity Framework.

Взаимодействие (Руководство по программированию в C#)

18.12.2021 • 2 minutes to read

Возможность взаимодействия позволяет использовать уже созданный неуправляемый код, экономя средства на разработку. Код, который выполняется под управлением среды CLR, называется *управляемым кодом*, а код, который выполняется вне среды CLR, называется *неуправляемым кодом*. COM, COM +, компоненты C++, компоненты ActiveX и Microsoft Windows API являются примерами неуправляемого кода.

.NET обеспечивает взаимодействие с неуправляемым кодом посредством служб вызова неуправляемого кода (PInvoke), пространства имен System.Runtime.InteropServices, COM-взаимодействия и взаимодействия с C++.

В этом разделе

Общие сведения о взаимодействии
Описывает способы взаимодействия между управляемым кодом C# и неуправляемым кодом.

Практическое руководство. Доступ к объектам взаимодействия Office с помощью функций языка C#
Описывает возможности, представленные в Visual C#, которые упрощают программирование для Office.

Практическое руководство. Использование индексированных свойств в программировании COM-взаимодействия
Описывает использование индексированных свойств для доступа к свойствам COM, которые имеют параметры.

Практическое руководство. Использование вызова неуправляемого кода для воспроизведения WAV-файла
Описывает, как использовать платформу вызова служб, чтобы воспроизвести звуковой WAV-файл в операционной системе Windows.

Пошаговое руководство. Программирование для Office
Описывает процесс создания книги Excel и документа Word со ссылкой на эту книгу.

Пример COM-класса
Предлагает пример предоставления класса C# в качестве COM-объекта.

Спецификация языка C#

Дополнительные сведения см. в разделе Основные понятия в Спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Marshal.ReleaseComObject
- Руководство по программированию на C#
- Взаимодействие с неуправляемым кодом
- Пошаговое руководство: Программирование для Office

Управление версиями в C#

18.12.2021 • 4 minutes to read

В этом учебнике вы получите представление об управлении версиями в .NET. Кроме того, вы узнаете, какие факторы следует учитывать при управлении версиями библиотеки и обновлении до ее новой версии.

Разработка библиотек

Если вы занимались разработкой библиотек .NET для открытого использования, то, скорее всего, вам приходилось развертывать обновления. Способ выполнения этой процедуры важен, так как необходимо обеспечить плавный перевод существующего кода на новую версию библиотеки. Ниже описывается ряд моментов, которые следует учитывать при создании выпуска.

Семантическое управление версиями

Семантическое версионирование (SemVer) — это соглашение об именовании, применяемое к версиям библиотеки для обозначения ключевых вех. В идеале сведения о версии, включаемые в библиотеку, должны помочь разработчикам определить совместимость с проектами, в которых используются предыдущие версии этой библиотеки.

Базовым вариантом SemVer является трехкомпонентный формат `MAJOR.MINOR.PATCH`, где:

- `MAJOR` увеличивается при внесении изменений, приводящих к несовместимости API;
- `MINOR` увеличивается при добавлении функциональных возможностей с обеспечением обратной совместимости;
- `PATCH` увеличивается при исправлении ошибок с обеспечением обратной совместимости.

При применении сведений о версии к библиотеке .NET возможны также и особые случаи, например, можно указать, что версия является предварительной и т. д.

Обратная совместимость

При выпуске новых версий библиотеки обеспечение обратной совместимости с предыдущими версиями, скорее всего, будет одной из основных задач. Новая версия библиотеки является совместимой с предыдущей на уровне исходного кода, если код, который зависит от предыдущей версии, после перекомпиляции может работать с новой версией. Новая версия библиотеки является совместимой на уровне двоичного кода, если приложение, которое зависело от предыдущей версии, может работать с новой версией без перекомпиляции.

Ниже описывается ряд моментов, которые следует учитывать при обеспечении обратной совместимости с предыдущими версиями библиотеки.

- Виртуальные методы. Когда вы делаете виртуальный метод невиртуальным в новой версии, это означает, что проекты, в которых этот метод переопределяется, необходимо будет обновить. Это очень существенное изменение, которое настоятельно не рекомендуется производить.
- Сигнатуры методов. Если при изменении поведения метода также необходимо изменить его сигнатуру, вместо этого следует создать перегрузку, чтобы код, вызывающий этот метод, по-прежнему работал. Прежнюю сигнатуру метода всегда можно настроить так, чтобы она вызывала новую сигнатуру, обеспечив тем самым согласованность реализаций.
- Атрибут Obsolete. С помощью этого атрибута можно указывать в коде классы или члены классов, которые являются нерекомендуемыми и, вероятно, будут удалены в будущих версиях. Благодаря этому разработчики, использующие вашу библиотеку, будут лучше подготовлены к существенным

изменениям.

- Необязательные аргументы метода. Если вы делаете необязательные аргументы метода обязательными или меняете их значения по умолчанию, то код, который не задает эти аргументы, потребуется обновить.

> **NOTE**
> Если обязательные аргументы становятся необязательными, то, как правило, это не должно иметь особых последствий, особенно если при этом поведение метода не меняется.

Чем проще пользователям будет произвести обновление до новой версии библиотеки, тем скорее они это сделают.

Файл конфигурации приложения

Как разработчик .NET вы, весьма вероятно, встречали файл `app.config` в большинстве типов проектов. Этот простой файл конфигурации может очень упростить развертывание обновлений. В основном библиотеки следует проектировать так, чтобы сведения, которые, скорее всего, будут меняться регулярно, хранились в файле `app.config`. Благодаря этому при изменении таких сведений достаточно будет заменить файл конфигурации предыдущей версии на новый, не перекомпилируя библиотеку.

Использование библиотек

Если при разработке вы используете библиотеки .NET, созданные другими разработчиками, то, скорее всего, знаете, что новая версия библиотеки может быть не полностью совместима с вашим проектом. В таком случае приходится обновлять код в соответствии с изменениями в библиотеке.

К счастью, экосистема C# и .NET предоставляет возможности и методы, которые позволяют обновлять приложения для работы с новыми версиями библиотек, в которые были внесены существенные изменения.

Перенаправление привязки сборок

С помощью файла *app.config* можно изменить версию библиотеки, используемую приложением. Добавив так называемое *перенаправление привязки*, вы можете использовать новую версию библиотеки, не перекомпилируя приложение. В приведенном ниже примере показано, как можно изменить файл приложения *app.config* так, чтобы оно использовало версию исправления `1.0.1` библиотеки `ReferencedLibrary` вместо версии `1.0.0`, с которой оно изначально компилировалось.

```
<dependentAssembly>
    <assemblyIdentity name="ReferencedLibrary" publicKeyToken="32ab4ba45e0a69a1" culture="en-us" />
    <bindingRedirect oldVersion="1.0.0" newVersion="1.0.1" />
</dependentAssembly>
```

> **NOTE**
> Такой подход будет работать только в том случае, если новая версия библиотеки `ReferencedLibrary` совместима с приложением на уровне двоичного кода. Сведения об изменениях, на которые следует обращать внимание при определении совместимости, см. выше в подразделе Обратная совместимость.

new

Модификатор `new` служит для скрытия унаследованных членов базового класса. Это один из способов, которым производные классы могут реагировать на изменения в базовом классе.

Рассмотрим следующий пример:

```
public class BaseClass
{
    public void MyMethod()
    {
        Console.WriteLine("A base method");
    }
}

public class DerivedClass : BaseClass
{
    public new void MyMethod()
    {
        Console.WriteLine("A derived method");
    }
}

public static void Main()
{
    BaseClass b = new BaseClass();
    DerivedClass d = new DerivedClass();

    b.MyMethod();
    d.MyMethod();
}
```

Выходные данные

```
A base method
A derived method
```

В приведенном выше примере можно видеть, как в классе `DerivedClass` скрывается метод `MyMethod`, имеющийся в классе `BaseClass`. Таким образом, если в базовом классе в новой версии библиотеки появляется член, который уже имеется в вашем производном классе, можно просто использовать модификатор `new` для члена производного класса, чтобы скрыть соответствующий член базового класса.

Если модификатор `new` не указан, в производном классе будут по умолчанию скрываться конфликтующие члены базового класса. При этом будет выводиться предупреждение компилятора, однако код будет компилироваться. Это означает, что добавление новых членов в существующий класс делает новую версию библиотеки совместимой с кодом, в котором она используется, на уровне как исходного, так и двоичного кода.

override

Модификатор `override` означает, что производная реализация расширяет реализацию члена базового класса, а не скрывает его. К члену базового класса должен быть применен модификатор `virtual`.

```
public class MyBaseClass
{
    public virtual string MethodOne()
    {
        return "Method One";
    }
}

public class MyDerivedClass : MyBaseClass
{
    public override string MethodOne()
    {
        return "Derived Method One";
    }
}

public static void Main()
{
    MyBaseClass b = new MyBaseClass();
    MyDerivedClass d = new MyDerivedClass();

    Console.WriteLine("Base Method One: {0}", b.MethodOne());
    Console.WriteLine("Derived Method One: {0}", d.MethodOne());
}
```

Выходные данные

```
Base Method One: Method One
Derived Method One: Derived Method One
```

Модификатор `override` оценивается во время компиляции, и компилятор выдаст ошибку, если ему не удастся найти виртуальный член для переопределения.

Знание рассмотренных приемов и ситуаций, в которых их следует применять, поможет вам значительно упростить переход с одной версии библиотеки на другую.

Практическое руководство (C#)

18.12.2021 • 3 minutes to read

В практическом руководстве по языку C# в разделе с инструкциями содержатся ответы на часто задаваемые вопросы. В некоторых случаях статьи могут быть указаны в нескольких разделах. Мы хотели упростить их обнаружение при использовании нескольких вариантов поиска.

Основные понятия C#

Существует несколько общих советов и рекомендаций для разработчиков на C#:

- Инициализация объектов с помощью инициализатора объектов.
- Различия между передачей структуры и класса в метод.
- Использование перегрузки операторов.
- Реализация и вызов пользовательского метода расширения.
- Создание метода для типа `enum` с помощью методов расширения.

Элементы "класс", "запись" и "структура"

Классы, записи и структуры создаются для реализации программы. При написании классов, записей или структур часто используются перечисленные ниже методы.

- Объявление автоматически реализуемых свойств.
- Объявление и использование свойств чтения и записи.
- Определение констант.
- Переопределение метода `ToString` для предоставления строковых выходных данных.
- Определение абстрактных свойств.
- Использование возможностей XML-документации для документирования кода.
- Явная реализация членов интерфейса для обеспечения краткости открытого интерфейса.
- Явная реализация членов двух интерфейсов.

Работа с коллекциями

Следующие статьи содержат сведения о работе с коллекциями данных.

- Инициализация словаря с помощью инициализатора коллекций.

Работа со строками

Строки являются основным типом данных, используемым для отображения текста или работы с ним. В этих статьях приводятся распространенные варианты использования строк.

- Сравнение строк.
- Изменение содержимого строки.
- Определение того, представляет ли строка число.
- Использование `String.Split` для разделения строк.
- Объединение нескольких строк в одну.
- Поиск текста в строке.

Преобразование типов

Может потребоваться преобразовать объект к другому типу.

- Определение того, представляет ли строка число.
- Преобразование строк, представляющих шестнадцатеричные значения, и числа.
- Преобразование строки в `DateTime`.
- Преобразование массива байтов в значение типа int.
- Преобразование строки в число.
- Использование операторов сопоставления шаблонов `as` и `is` для безопасного приведения к другому типу.
- Определение пользовательского преобразования типов.
- Определение того, допускает ли тип значения NULL.
- Преобразование между типами значений, допускающие значения NULL, и типами, не допускающими значения NULL.

Упорядочение и сравнение на равенство

Можно создать типы, которые определяют собственные правила проверки на равенство или определяют естественное упорядочение объектов этого типа.

- Проверка на равенство на основе ссылок.
- Определение равенства на основе значений для типа.

Обработка исключений

В программах на .NET сообщения о неудачных завершениях методов выводятся путем создания исключений. В следующих статьях вы узнаете, как работать с исключениями.

- Обработка исключений с помощью `try` и `catch`.
- Очистка ресурсов с помощью предложений `finally`.
- Восстановление в результате возникновения исключений, несовместимых с CLS.

Делегаты и события

Делегаты и события предоставляют возможность формирования стратегий использования слабосвязанных блоков кода.

- Объявление, создание и использование делегатов.
- Объединение многоадресных делегатов.

События предоставляют механизм публикации уведомлений или подписки на них.

- Подписка и отмена подписки на события.
- Реализация событий, объявленных в интерфейсах.
- Соответствие рекомендациям .NET при публикации событий кодом.
- Создание событий, определенных в базовых классах, из производных классов.
- Реализация пользовательских методов доступа к событиям.

Рекомендации по LINQ

LINQ позволяет создавать код для запросов к любому источнику данных, который поддерживает шаблон выражения запроса LINQ. Сведения в следующих статьях помогут понять принципы использования шаблонов и работы с различными источниками данных.

- Запрос коллекции.

- Использование `var` в выражениях запросов.
- Возвращение поднаборов свойств элементов из запроса.
- Создание запросов со сложной фильтрацией.
- Сортировка элементов источника данных.
- Сортировка элементов по нескольким ключам.
- Управление типом проекции.
- Подсчет вхождений значения в исходной последовательности.
- Вычисление промежуточных значений.
- Слияние данных из нескольких источников.
- Нахождение разности наборов между двумя последовательностями.
- Отладка пустых результатов запроса.
- Добавление настраиваемых методов в запросы LINQ.

Несколько потоков и асинхронная обработка

В современных программах часто используются асинхронные операции. Сведения о работе с ними приводятся в следующих статьях.

- Повышение производительности асинхронных операций с помощью `System.Threading.Tasks.Task.WhenAll`.
- Параллельное выполнение нескольких веб-запросов с помощью `async` и `await`.
- Использование пула потоков.

Аргументы командной строки для программы

Как правило, в программах на C# используются аргументы командной строки. В следующих статьях содержатся сведения о доступе к этим аргументам командной строки и их обработке.

- Получение всех аргументов командной строки с помощью `for`.

Разделение строк с помощью метода String.Split на C#

18.12.2021 • 2 minutes to read

Метод String.Split создает массив подстрок, разбивая входную строку по одному или нескольким разделителям. Этот метод зачастую является самым простым способом разделить строку по границам слов. Он также используется для разбиения строк по другим конкретным символам или строкам.

> **NOTE**
>
> Примеры C# в этой статье выполняются во встроенном средстве выполнения кода и на площадке Try.NET. Нажмите на кнопку **Выполнить**, чтобы выполнить пример в интерактивном окне. После выполнения кода вы можете изменить его и выполнить измененный код, снова нажав на кнопку **Выполнить**. Либо в интерактивном окне выполняется измененный код, либо, если компиляция завершается с ошибкой, в интерактивном окне отображаются все сообщения об ошибках компилятора C#.

Следующий код разбивает обычную фразу на массив строк для каждого слова.

```
string phrase = "The quick brown fox jumps over the lazy dog.";
string[] words = phrase.Split(' ');

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

Каждый экземпляр знака разделения создает значение в возвращаемом массиве. Последовательные знаки разделения создают пустую строку в виде значения в возвращаемом массиве. В следующем примере показано создание пустой строки с использованием символа пробела в качестве разделителя.

```
string phrase = "The quick brown    fox     jumps over the lazy dog.";
string[] words = phrase.Split(' ');

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

Такое поведение упрощает работу с такими форматами, как файл данных с разделителями-запятыми (CSV), которые представляют табличные данные. Идущие подряд запятые представляют пустой столбец.

Чтобы исключить из возвращаемого массива все пустые строки, можно передать необязательный параметр StringSplitOptions.RemoveEmptyEntries. Для более сложной обработки возвращенной коллекции можно использовать LINQ, чтобы управлять результирующей последовательностью.

String.Split может использовать несколько знаков разделения. В приведенном ниже примере в качестве знаков разделения используются пробелы, запятые, точки, двоеточия и символы табуляции, которые передаются в Split в массиве. Цикл в конце кода отображает каждое из слов в возвращенном массиве.

```
char[] delimiterChars = { ' ', ',', '.', ':', '\t' };

string text = "one\ttwo three:four,five six seven";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(delimiterChars);
System.Console.WriteLine($"{words.Length} words in text:");

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

Последовательные экземпляры любого разделителя создают пустую строку в выходном массиве:

```
char[] delimiterChars = { ' ', ',', '.', ':', '\t' };

string text = "one\ttwo :,five six seven";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(delimiterChars);
System.Console.WriteLine($"{words.Length} words in text:");

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

Метод String.Split может принимать массив строк (в этом случае в качестве разделителей при анализе целевой строки используются последовательности символов, а не отдельные символы).

```
string[] separatingStrings = { "<<", "..." };

string text = "one<<two......three<four";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(separatingStrings, System.StringSplitOptions.RemoveEmptyEntries);
System.Console.WriteLine($"{words.Length} substrings in text:");

foreach (var word in words)
{
    System.Console.WriteLine(word);
}
```

См. также

- Извлечение элементов из строки
- Руководство по программированию на C#
- Строки
- Регулярные выражения .NET

Практическое руководство. Сцепка нескольких строк (руководство по C#)

18.12.2021 • 3 minutes to read

Объединение подразумевает добавление одной строки к концу другой. Вы можете сцеплять строки с помощью оператора `+`. Строковые литералы и константы сцепляются во время компиляции, а не во время выполнения. Строковые переменные сцепляются только во время выполнения.

> **NOTE**
> Примеры C# в этой статье выполняются во встроенном средстве выполнения кода и на площадке Try.NET. Нажмите на кнопку **Выполнить**, чтобы выполнить пример в интерактивном окне. После выполнения кода вы можете изменить его и выполнить измененный код, снова нажав на кнопку **Выполнить**. Либо в интерактивном окне выполняется измененный код, либо, если компиляция завершается с ошибкой, в интерактивном окне отображаются все сообщения об ошибках компилятора C#.

Строковые литералы

В следующем примере длинный строковый литерал разделяется на более короткие строки для повышения удобочитаемости исходного кода. Код сцепляет строки меньшего размера для создания длинного строкового литерала. Сегменты сцепляются в одну строку во время компиляции. Количество строк не влияет на производительность во время выполнения.

```
// Concatenation of literals is performed at compile time, not run time.
string text = "Historically, the world of data and the world of objects " +
"have not been well integrated. Programmers work in C# or Visual Basic " +
"and also in SQL or XQuery. On the one side are concepts such as classes, " +
"objects, fields, inheritance, and .NET Framework APIs. On the other side " +
"are tables, columns, rows, nodes, and separate languages for dealing with " +
"them. Data types often require translation between the two worlds; there are " +
"different standard functions. Because the object world has no notion of query, a " +
"query can only be represented as a string without compile-time type checking or " +
"IntelliSense support in the IDE. Transferring data from SQL tables or XML trees to " +
"objects in memory is often tedious and error-prone.";

System.Console.WriteLine(text);
```

Операторы `+` и `+=`

Для сцепки строковых переменных вы можете использовать операторы `+` или `+=`, интерполяцию строк, а также методы String.Format, String.Concat, String.Join или StringBuilder.Append. Оператор `+` прост в использовании и позволяет получить интуитивно понятный код. Даже если в одном выражении используется несколько операторов `+`, содержимое строки копируется только один раз. В следующем коде показаны примеры использования операторов `+` и `+=` для сцепки строк:

```
string userName = "<Type your name here>";
string dateString = DateTime.Today.ToShortDateString();

// Use the + and += operators for one-time concatenations.
string str = "Hello " + userName + ". Today is " + dateString + ".";
System.Console.WriteLine(str);

str += " How are you today?";
System.Console.WriteLine(str);
```

Интерполяция строк

В некоторых выражениях строки проще сцепить с помощью интерполяции, как показано в следующем коде:

```
string userName = "<Type your name here>";
string date = DateTime.Today.ToShortDateString();

// Use string interpolation to concatenate strings.
string str = $"Hello {userName}. Today is {date}.";
System.Console.WriteLine(str);

str = $"{str} How are you today?";
System.Console.WriteLine(str);
```

> **NOTE**
> В операциях сцепки строк компилятор C# обрабатывает строки NULL так же, как пустые строки.

Начиная с C# 10, можно использовать интерполяцию строк для инициализации константной строки, если все выражения, используемые для заполнителей, также являются константными строками.

`String.Format`

Другие методы сцепки строк: String.Format. Этот метод лучше использовать при создании строки из небольшого числа строк-компонентов.

`StringBuilder`

В других случаях вы можете сцеплять строки во время цикла и не знать, сколько исходных строк вы сцепляете. При этом фактическое число исходных строк может быть большим. Для этих сценариев предназначен класс StringBuilder. В следующем коде для сцепки строк используется метод Append класса StringBuilder.

```
// Use StringBuilder for concatenation in tight loops.
var sb = new System.Text.StringBuilder();
for (int i = 0; i < 20; i++)
{
    sb.AppendLine(i.ToString());
}
System.Console.WriteLine(sb.ToString());
```

См. дополнительные сведения о причинах для выбора объединения строк или класса `StringBuilder` .

`String.Concat` или `String.Join`

Другой вариант объединения строк из коллекции — использовать метод String.Concat. Используйте метод String.Join, если исходные строки должны быть разделены разделителем. Следующий код объединяет массив слов с помощью обоих методов:

```
string[] words = { "The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog." };

var unreadablePhrase = string.Concat(words);
System.Console.WriteLine(unreadablePhrase);

var readablePhrase = string.Join(" ", words);
System.Console.WriteLine(readablePhrase);
```

LINQ и `Enumerable.Aggregate`

Другой вариант объединения строк из коллекции — использовать LINQ и метод Enumerable.Aggregate. Этот метод объединяет исходные строки с помощью лямбда-выражения. Лямбда-выражение добавляет каждую строку к существующему накоплению. Следующий пример показывает объединение массива слов с добавлением между словами пробела:

```
string[] words = { "The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog." };

var phrase = words.Aggregate((partialPhrase, word) =>$"{partialPhrase} {word}");
System.Console.WriteLine(phrase);
```

Этот вариант может потребовать большего числа выделений памяти по сравнению с другими методами сцепки коллекций, так как он создает в каждой итерации промежуточную строку. Если критически важно оптимизировать производительность, попробуйте вместо `Enumerable.Aggregate` использовать для сцепки коллекции класс `StringBuilder` либо метод `String.Concat` или `String.Join`.

См. также

- String
- StringBuilder
- Руководство по программированию на C#
- Строки

Практическое руководство. Поиск по строкам

18.12.2021 • 3 minutes to read

Существует две основные стратегии для поиска текста в строках. Методы класса String выполняют поиск определенного текста. Регулярные выражения используются для поиска шаблонов в тексте.

> **NOTE**
> Примеры C# в этой статье выполняются во встроенном средстве выполнения кода и на площадке Try.NET. Нажмите на кнопку **Выполнить**, чтобы выполнить пример в интерактивном окне. После выполнения кода вы можете изменить его и выполнить измененный код, снова нажав на кнопку **Выполнить**. Либо в интерактивном окне выполняется измененный код, либо, если компиляция завершается с ошибкой, в интерактивном окне отображаются все сообщения об ошибках компилятора C#.

Тип string является псевдонимом класса System.String и реализует ряд полезных методов для поиска содержимого строк. Среди них: Contains, StartsWith, EndsWith, IndexOf, LastIndexOf. Класс System.Text.RegularExpressions.Regex предоставляет широкие возможности словаря для поиска шаблонов в тексте. В этой статье вы узнаете, как применять эти методы и как выбрать наилучший способ в зависимости от ваших потребностей.

Содержит ли строка текст?

Методы String.Contains, String.StartsWith и String.EndsWith выполняют поиск определенного текста в строке. В следующем примере показано использование каждого из этих методов, а также сценарии поиска без учета регистра:

```
string factMessage = "Extension methods have all the capabilities of regular static methods.";

// Write the string and include the quotation marks.
Console.WriteLine($"\"{factMessage}\"");

// Simple comparisons are always case sensitive!
bool containsSearchResult = factMessage.Contains("extension");
Console.WriteLine($"Contains \"extension\"? {containsSearchResult}");

// For user input and strings that will be displayed to the end user,
// use the StringComparison parameter on methods that have it to specify how to match strings.
bool ignoreCaseSearchResult = factMessage.StartsWith("extension",
System.StringComparison.CurrentCultureIgnoreCase);
Console.WriteLine($"Starts with \"extension\"? {ignoreCaseSearchResult} (ignoring case)");

bool endsWithSearchResult = factMessage.EndsWith(".", System.StringComparison.CurrentCultureIgnoreCase);
Console.WriteLine($"Ends with '.'? {endsWithSearchResult}");
```

В предыдущем примере показано важное правило использования этих методов. По умолчанию поиск выполняется **с учетом регистра**. Чтобы выполнить поиск без учета регистра, используйте значение перечисления StringComparison.CurrentCultureIgnoreCase.

Где искомый текст находится в строке?

Методы IndexOf и LastIndexOf также ищут текст в строках. Эти методы возвращают расположение текста, поиск которого выполняется. Если текст не найден, возвращается `-1`. В следующем примере происходит

поиск первого и последнего вхождения слова "methods" и отображение текста, находящегося между ними.

```
string factMessage = "Extension methods have all the capabilities of regular static methods.";

// Write the string and include the quotation marks.
Console.WriteLine($"\"{factMessage}\"");

// This search returns the substring between two strings, so
// the first index is moved to the character just after the first string.
int first = factMessage.IndexOf("methods") + "methods".Length;
int last = factMessage.LastIndexOf("methods");
string str2 = factMessage.Substring(first, last - first);
Console.WriteLine($"Substring between \"methods\" and \"methods\": '{str2}'");
```

Поиск определенного текста с помощью регулярных выражений

Класс System.Text.RegularExpressions.Regex можно использовать для поиска строк. Такой поиск может отличаться по сложности от самых простых до очень сложных текстовых шаблонов.

В следующем примере кода выполняется поиск слов "the" и "their" в предложении без учета регистра. Статический метод Regex.IsMatch выполняет поиск. В метод передается строка и шаблон поиска. В нашем примере третий аргумент задает поиск без учета регистра. Для получения дополнительной информации см. System.Text.RegularExpressions.RegexOptions.

Шаблон поиска описывает текст для поиска. Следующая таблица описывает каждый элемент шаблона поиска. (В таблице ниже используется один `\`, который в строке C# необходимо экранировать как `\\`).

ШАБЛОН	ЗНАЧЕНИЕ
`the`	соответствует тексту "the"
`(eir)?`	Соответствует 0 или 1 вхождению "eir"
`\s`	Соответствует пробелу.

```
string[] sentences =
{
    "Put the water over there.",
    "They're quite thirsty.",
    "Their water bottles broke."
};

string sPattern = "the(ir)?\\s";

foreach (string s in sentences)
{
    Console.Write($"{s,24}");

    if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern,
System.Text.RegularExpressions.RegexOptions.IgnoreCase))
    {
        Console.WriteLine($"  (match for '{sPattern}' found)");
    }
    else
    {
        Console.WriteLine();
    }
}
```

TIP

Методы `string` обычно удобнее при поиске точного совпадения со строкой. Регулярные выражения больше подходят при поиске определенных шаблонов в исходной строке.

Соответствует ли строка шаблону?

В следующем коде реализуется проверка формата каждой строки в массиве с использованием регулярных выражений. По условиям проверки каждая строка должна иметь формат номера телефона: три группы цифр, разделенных дефисами. Первые две группы содержат по три цифры, и третья группа состоит из четырех цифр. Шаблон поиска использует регулярное выражение `^\\d{3}-\\d{3}-\\d{4}$`. Дополнительные сведения см. в разделе Элементы языка регулярных выражений. Краткий справочник.

ШАБЛОН	ЗНАЧЕНИЕ
`^`	соответствует началу строки
`\d{3}`	соответствует в точности 3 цифрам
`-`	соответствует символу "–"
`\d{4}`	соответствует в точности 4 цифрам
`$`	соответствует концу строки

```
string[] numbers =
{
    "123-555-0190",
    "444-234-22450",
    "690-555-0178",
    "146-893-232",
    "146-555-0122",
    "4007-555-0111",
    "407-555-0111",
    "407-2-5555",
    "407-555-8974",
    "407-2ab-5555",
    "690-555-8148",
    "146-893-232-"
};

string sPattern = "^\\d{3}-\\d{3}-\\d{4}$";

foreach (string s in numbers)
{
    Console.Write($"{s,14}");

    if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern))
    {
        Console.WriteLine(" - valid");
    }
    else
    {
        Console.WriteLine(" - invalid");
    }
}
```

Один шаблон поиска соответствует множеству допустимых строк. Регулярные выражения больше подходят для поиска или проверки соответствия шаблону, а не для поиска отдельной строки текста.

См. также

- Руководство по программированию на C#
- Строки
- LINQ и строки
- System.Text.RegularExpressions.Regex
- Регулярные выражения .NET
- Элементы языка регулярных выражений — краткий справочник
- Рекомендации по использованию строк в .NET

Практическое руководство. Изменение содержимого строки в C#

18.12.2021 • 4 minutes to read

В этой статье демонстрируется несколько способов получения `string` путем изменения существующего объекта `string`. Все показанные методы возвращают результат изменений как новый объект `string`. Чтобы показать, что исходная и измененная строки являются отдельными экземплярами, в этих примерах результат сохраняется в новой переменной. При выполнении каждого примера можно изучить исходный `string` и новый, измененный `string`.

> **NOTE**
> Примеры C# в этой статье выполняются во встроенном средстве выполнения кода и на площадке Try.NET. Нажмите на кнопку **Выполнить**, чтобы выполнить пример в интерактивном окне. После выполнения кода вы можете изменить его и выполнить измененный код, снова нажав на кнопку **Выполнить**. Либо в интерактивном окне выполняется измененный код, либо, если компиляция завершается с ошибкой, в интерактивном окне отображаются все сообщения об ошибках компилятора C#.

В статье приводится несколько методов. Вы можете заменять существующий текст. Вы можете искать шаблоны и заменять соответствующий текст другим. Вы можете рассматривать строку как последовательность символов. Вы также можете использовать удобные методы удаления пробелов. Выберите метод, наиболее подходящий для вашего сценария.

Замена текста

Следующий код создает новую строку, заменяя существующий текст.

```
string source = "The mountains are behind the clouds today.";

// Replace one substring with another with String.Replace.
// Only exact matches are supported.
var replacement = source.Replace("mountains", "peaks");
Console.WriteLine($"The source string is <{source}>");
Console.WriteLine($"The updated string is <{replacement}>");
```

В коде выше демонстрируется *неизменяемое* свойство строк. В примере видно, что исходная строка `source` не изменяется. Метод String.Replace создает новый объект `string`, содержащий изменения.

Метод Replace может заменять строки или отдельные символы. В обоих случаях заменяется каждое вхождение искомого текста. В следующем примере все символы ' ' заменяются на '_':

```
string source = "The mountains are behind the clouds today.";

// Replace all occurrences of one char with another.
var replacement = source.Replace(' ', '_');
Console.WriteLine(source);
Console.WriteLine(replacement);
```

Исходная строка не изменяется. Возвращается новая строка с заменой.

Усечение пробелов

Используйте методы String.Trim, String.TrimStart и String.TrimEnd для удаления всех начальных или конечных пробелов. В приведенном ниже коде показан пример каждого метода. Исходная строка не изменяется. Эти методы возвращают новую строку с измененным содержимым.

```
// Remove trailing and leading white space.
string source = "    I'm wider than I need to be.      ";
// Store the results in a new string variable.
var trimmedResult = source.Trim();
var trimLeading = source.TrimStart();
var trimTrailing = source.TrimEnd();
Console.WriteLine($"<{source}>");
Console.WriteLine($"<{trimmedResult}>");
Console.WriteLine($"<{trimLeading}>");
Console.WriteLine($"<{trimTrailing}>");
```

Удаление текста

Вы можете удалять из строки текст с помощью метода String.Remove. Этот метод удаляет определенное число символов, начиная с указанного индекса. В следующем примере показано, как использовать String.IndexOf с Remove для удаления текста из строки:

```
string source = "Many mountains are behind many clouds today.";
// Remove a substring from the middle of the string.
string toRemove = "many ";
string result = string.Empty;
int i = source.IndexOf(toRemove);
if (i >= 0)
{
    result= source.Remove(i, toRemove.Length);
}
Console.WriteLine(source);
Console.WriteLine(result);
```

Замена совпадающих шаблонов

Используйте регулярные выражения для замены текста, соответствующего шаблонам, на новый текст, который может быть определен шаблоном. В следующем примере используется класс System.Text.RegularExpressions.Regex для поиска шаблона в исходной строке и замены его с правильным регистром. Метод Regex.Replace(String, String, MatchEvaluator, RegexOptions) принимает в качестве одного из аргументов функцию, которая предоставляет логику замены. В примере эта функция (`LocalReplaceMatchCase`) является **локальной функцией** и объявляется внутри демонстрируемого метода. `LocalReplaceMatchCase` использует класс System.Text.StringBuilder, чтобы создать замещающую строку с правильным регистром.

Регулярные выражения наиболее эффективны при поиске и замене текста, который соответствует шаблону, а не известного текста. Дополнительные сведения см. в практическом руководстве по поиску строк. Шаблон поиска "the\s" ищет слово "the", за которым следует пробел. Эта часть шаблона гарантирует пропуск слова "there" в исходной строке. Дополнительные сведения об элементах языка регулярных выражений см. в разделе Элементы языка регулярных выражений — краткий справочник.

```
string source = "The mountains are still there behind the clouds today.";

// Use Regex.Replace for more flexibility.
// Replace "the" or "The" with "many" or "Many".
// using System.Text.RegularExpressions
string replaceWith = "many ";
source = System.Text.RegularExpressions.Regex.Replace(source, "the\\s", LocalReplaceMatchCase,
    System.Text.RegularExpressions.RegexOptions.IgnoreCase);
Console.WriteLine(source);

string LocalReplaceMatchCase(System.Text.RegularExpressions.Match matchExpression)
{
    // Test whether the match is capitalized
    if (Char.IsUpper(matchExpression.Value[0]))
    {
        // Capitalize the replacement string
        System.Text.StringBuilder replacementBuilder = new System.Text.StringBuilder(replaceWith);
        replacementBuilder[0] = Char.ToUpper(replacementBuilder[0]);
        return replacementBuilder.ToString();
    }
    else
    {
        return replaceWith;
    }
}
```

Метод StringBuilder.ToString возвращает неизменяемую строку с содержимым в объекте StringBuilder.

Изменение отдельных символов

Вы можете создать из строки массив символов, изменить содержимое массива, а затем создать из измененного содержимого новую строку.

В примере ниже показано, как заменить набор символов в строке. Сначала используется метод String.ToCharArray() для создания массива символов. Для поиска начального индекса слова "fox" используется метод IndexOf. Следующие три символа заменяются другим словом. Наконец, из обновленного массива символов создается новая строка.

```
string phrase = "The quick brown fox jumps over the fence";
Console.WriteLine(phrase);

char[] phraseAsChars = phrase.ToCharArray();
int animalIndex = phrase.IndexOf("fox");
if (animalIndex != -1)
{
    phraseAsChars[animalIndex++] = 'c';
    phraseAsChars[animalIndex++] = 'a';
    phraseAsChars[animalIndex] = 't';
}

string updatedPhrase = new string(phraseAsChars);
Console.WriteLine(updatedPhrase);
```

Программная сборка содержимого строки

Поскольку строки являются неизменяемыми, в предыдущих примерах создаются временные строки или массивы символов. В высокопроизводительных сценариях, возможно, будет целесообразным избежать этих распределений куч. .NET Core предоставляет метод String.Create, позволяющий программно заполнять символьное содержимое строки с помощью обратного вызова, избегая промежуточного распределения строк.

```
// constructing a string from a char array, prefix it with some additional characters
char[] chars = { 'a', 'b', 'c', 'd', '\0' };
int length = chars.Length + 2;
string result = string.Create(length, chars, (Span<char> strContent, char[] charArray) =>
{
    strContent[0] = '0';
    strContent[1] = '1';
    for (int i = 0; i < charArray.Length; i++)
    {
        strContent[i + 2] = charArray[i];
    }
});

Console.WriteLine(result);
```

Вы можете изменить строку в фиксированном блоке с использованием ненадежного кода, но **крайне** не рекомендует изменять содержимое строки после ее создания. Такая попытка приведет к непредсказуемым последствиям. Например, если кто-то развернет строку, имеющую то же содержимое, что и ваша строка, он получит копию вашей строки и не будет рассчитывать на то, что вы изменяете его строку.

См. также

- Регулярные выражения .NET
- Элементы языка регулярных выражений — краткий справочник

Сравнение строк в C#

18.12.2021 • 10 minutes to read

Сравнивая строки, вы хотите ответить на один из двух вопросов: "Равны ли две эти строки?" или "В каком порядке должны следовать эти строки при их сортировке?".

Однако эту задачу осложняют факторы, влияющие на сравнение строк:

- Вы можете выбрать порядковое или лингвистическое сравнение.
- Вы можете указать, учитывается ли регистр.
- Вы можете выбрать сравнения для конкретного языка и региональных параметров.
- Лингвистические сравнения зависят от языка и региональных параметров, а также от используемой платформы.

> **NOTE**
> Примеры C# в этой статье выполняются во встроенном средстве выполнения кода и на площадке Try.NET. Нажмите на кнопку **Выполнить**, чтобы выполнить пример в интерактивном окне. После выполнения кода вы можете изменить его и выполнить измененный код, снова нажав на кнопку **Выполнить**. Либо в интерактивном окне выполняется измененный код, либо, если компиляция завершается с ошибкой, в интерактивном окне отображаются все сообщения об ошибках компилятора C#.

При сравнении строк вы определяете их порядок. Сравнения используются для сортировки последовательности строк. Если последовательность имеет известный порядок, это упрощает поиск как для программного обеспечения, так и для пользователей. Другие сравнения могут проверять совпадение строк. Эти проверки тождественности похожи на проверки равенства, но позволяют игнорировать некоторые различия, например различия в регистре.

Порядковые сравнения по умолчанию

Ниже представлены самые распространенные операции по умолчанию:

- String.Equals
- String.Equality и String.Inequality, т. е. операторы равенства `==` и `!=` соответственно

Выполните порядковое сравнение с учетом регистра, а в случае String.Equals можно указать аргумент StringComparison, чтобы изменить правила сортировки. Это показано в следующем примере:

```
string root = @"C:\users";
string root2 = @"C:\Users";

bool result = root.Equals(root2);
Console.WriteLine($"Ordinal comparison: <{root}> and <{root2}> are {(result ? "equal." : "not equal.")}");

result = root.Equals(root2, StringComparison.Ordinal);
Console.WriteLine($"Ordinal comparison: <{root}> and <{root2}> are {(result ? "equal." : "not equal.")}");

Console.WriteLine($"Using == says that <{root}> and <{root2}> are {(root == root2 ? "equal" : "not
equal")}");
```

При порядковых сравнениях строк по умолчанию лингвистические правила не учитываются. В них сравнивается двоичное значение каждого объекта Char в двух строках. Таким образом, в порядковом

сравнении по умолчанию также учитывается регистр.

Проверка на равенство с использованием String.Equals, а также операторов `==` и `!=` отличается от сравнения строк с использованием методов String.CompareTo и Compare(String, String). Хотя в проверках на равенство выполняется порядковое сравнение с учетом регистра, метод сравнения выполняет сравнение с учетом регистра, с учетом языка и региональных параметров с использованием текущего значения языка и региональных параметров. Поскольку методы сравнения по умолчанию часто выполняют сравнения различных типов, рекомендуется всегда четко определять назначение кода путем вызова перегрузки, которая явно указывает тип выполняемого сравнения.

Порядковые сравнения без учета регистра

Метод String.Equals(String, StringComparison) позволяет указать значение StringComparison для объекта StringComparison.OrdinalIgnoreCase для порядкового сравнения без учета регистра. Также имеется статический метод String.Compare(String, String, StringComparison), позволяющий проводить порядковое сравнение без учета регистра, если указать значение StringComparison.OrdinalIgnoreCase для аргумента StringComparison. Это показано в следующем коде:

```
string root = @"C:\users";
string root2 = @"C:\Users";

bool result = root.Equals(root2, StringComparison.OrdinalIgnoreCase);
bool areEqual = String.Equals(root, root2, StringComparison.OrdinalIgnoreCase);
int comparison = String.Compare(root, root2, comparisonType: StringComparison.OrdinalIgnoreCase);

Console.WriteLine($"Ordinal ignore case: <{root}> and <{root2}> are {(result ? "equal." : "not equal.")}");
Console.WriteLine($"Ordinal static ignore case: <{root}> and <{root2}> are {(areEqual ? "equal." : "not
equal.")}");
if (comparison < 0)
    Console.WriteLine($"<{root}> is less than <{root2}>");
else if (comparison > 0)
    Console.WriteLine($"<{root}> is greater than <{root2}>");
else
    Console.WriteLine($"<{root}> and <{root2}> are equivalent in order");
```

При выполнении порядкового сравнения без учета регистра эти методы используют соглашения о регистре для инвариантного языка и региональных параметров.

Лингвистические сравнения

Строки могут быть упорядочены с использованием лингвистических правил для текущих значений языка и региональных параметров. Иногда это называется порядком сортировки слов. При лингвистическом сравнении некоторые символы Юникода, отличные от алфавитно-цифровых, могут иметь особые весовые коэффициенты. Например, дефис "-" может иметь низкий весовой коэффициент, чтобы слова "co-op" и "coop" находились рядом друг с другом в порядке сортировки. Кроме того, некоторые символы Юникода могут быть эквивалентны последовательности экземпляров Char. В следующем примере используется фраза "Они танцуют на улице." на немецком языке с буквами "ss" (U+0073 U+0073) в одной строке и буквой "ß" (U+00DF) в другой. Лингвистически (в Windows) буквы "ss" равнозначны немецкому символу эсцет "ß" в языках "en-US" и "de-DE".

```
string first = "Sie tanzen auf der Straße.";
string second = "Sie tanzen auf der Strasse.";

Console.WriteLine($"First sentence is <{first}>");
Console.WriteLine($"Second sentence is <{second}>");

bool equal = String.Equals(first, second, StringComparison.InvariantCulture);
Console.WriteLine($"The two strings {(equal == true ? "are" : "are not")} equal.");
showComparison(first, second);

string word = "coop";
string words = "co-op";
string other = "cop";

showComparison(word, words);
showComparison(word, other);
showComparison(words, other);
void showComparison(string one, string two)
{
    int compareLinguistic = String.Compare(one, two, StringComparison.InvariantCulture);
    int compareOrdinal = String.Compare(one, two, StringComparison.Ordinal);
    if (compareLinguistic < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using invariant culture");
    else if (compareLinguistic > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using invariant culture");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order using invariant culture");
    if (compareOrdinal < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using ordinal comparison");
    else if (compareOrdinal > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using ordinal comparison");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order using ordinal comparison");
}
```

Этот пример демонстрирует зависимость операций лингвистического сравнения от операционной системы. Для интерактивного окна используется узел Linux. Лингвистические и порядковые сравнения дают одинаковые результаты. Если запустить этот пример на узле Windows, отображаются следующие выходные данные:

```
<coop> is less than <co-op> using invariant culture
<coop> is greater than <co-op> using ordinal comparison
<coop> is less than <cop> using invariant culture
<coop> is less than <cop> using ordinal comparison
<co-op> is less than <cop> using invariant culture
<co-op> is less than <cop> using ordinal comparison
```

В Windows порядок сортировки "cop", "coop" и "co-op" изменяется при переходе от лингвистического сравнения к порядковому. Два предложения на немецком языке также сравниваются по-разному при использовании разных типов сравнения.

Сравнения с использованием определенных языков и региональных параметров

Этот пример сохраняет объекты CultureInfo для языков "en-US" и "de-DE". Сравнения выполняются с использованием объекта CultureInfo, чтобы учесть язык и региональные параметры.

Используемые значения языка и региональных параметров влияют на операции лингвистического сравнения. В следующем примере показаны результаты сравнения двух предложений на немецком с использованием языка и региональных параметров "en US" и "de-DE":

```
string first = "Sie tanzen auf der Straße.";
string second = "Sie tanzen auf der Strasse.";

Console.WriteLine($"First sentence is <{first}>");
Console.WriteLine($"Second sentence is <{second}>");

var en = new System.Globalization.CultureInfo("en-US");

// For culture-sensitive comparisons, use the String.Compare
// overload that takes a StringComparison value.
int i = String.Compare(first, second, en, System.Globalization.CompareOptions.None);
Console.WriteLine($"Comparing in {en.Name} returns {i}.");

var de = new System.Globalization.CultureInfo("de-DE");
i = String.Compare(first, second, de, System.Globalization.CompareOptions.None);
Console.WriteLine($"Comparing in {de.Name} returns {i}.");

bool b = String.Equals(first, second, StringComparison.CurrentCulture);
Console.WriteLine($"The two strings {(b ? "are" : "are not")} equal.");

string word = "coop";
string words = "co-op";
string other = "cop";

showComparison(word, words, en);
showComparison(word, other, en);
showComparison(words, other, en);
void showComparison(string one, string two, System.Globalization.CultureInfo culture)
{
    int compareLinguistic = String.Compare(one, two, en, System.Globalization.CompareOptions.None);
    int compareOrdinal = String.Compare(one, two, StringComparison.Ordinal);
    if (compareLinguistic < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using en-US culture");
    else if (compareLinguistic > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using en-US culture");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order using en-US culture");
    if (compareOrdinal < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using ordinal comparison");
    else if (compareOrdinal > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using ordinal comparison");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order using ordinal comparison");
}
```

Сравнения с учетом языка и региональных параметров обычно используются для сравнения и сортировки строк, вводимых пользователями. Символы и правила сортировки этих строк могут различаться в зависимости от языкового стандарта компьютера пользователя. Даже строки, содержащие идентичные символы, могут быть отсортированы по-разному, в зависимости от языка и региональных параметров текущего потока. Кроме того, попробуйте запустить этот пример кода локально на компьютере с Windows, и вы получите следующие результаты:

```
<coop> is less than <co-op> using en-US culture
<coop> is greater than <co-op> using ordinal comparison
<coop> is less than <cop> using en-US culture
<coop> is less than <cop> using ordinal comparison
<co-op> is less than <cop> using en-US culture
<co-op> is less than <cop> using ordinal comparison
```

Лингвистические сравнения зависят от текущих значений языка и региональных параметров, а также от операционной системы. Это необходимо учитывать при работе со строковыми сравнениями.

Лингвистическая сортировка и поиск строк в массивах

Приведенные ниже примеры показывают, как сортировать и искать строки в массиве с помощью лингвистического сравнения, зависящего от текущих значений языка и региональных параметров. Используйте статические методы Array, которые принимают параметр System.StringComparer.

В этом примере показано, как сортировать массив строк с использованием текущих значений языка и региональных параметров:

```
string[] lines = new string[]
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};

Console.WriteLine("Non-sorted order:");
foreach (string s in lines)
{
    Console.WriteLine($"   {s}");
}

Console.WriteLine("\n\rSorted order:");

// Specify Ordinal to demonstrate the different behavior.
Array.Sort(lines, StringComparer.CurrentCulture);

foreach (string s in lines)
{
    Console.WriteLine($"   {s}");
}
```

После сортировки массива можно выполнить поиск записей с помощью двоичного поиска. Двоичный поиск начинается с середины коллекции, чтобы определить, какая половина коллекции содержит искомую строку. Каждое последующее сравнение делит оставшуюся часть коллекции пополам. Массив сортируется с использованием StringComparer.CurrentCulture. Локальная функция `ShowWhere` отображает сведения о том, где была найдена строка. Если строка не найдена, возвращаемое значение указывает, где бы оно находилось, если было бы найдено.

```
string[] lines = new string[]
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};
Array.Sort(lines, StringComparer.CurrentCulture);

string searchString = @"c:\public\TEXTFILE.TXT";
Console.WriteLine($"Binary search for <{searchString}>");
int result = Array.BinarySearch(lines, searchString, StringComparer.CurrentCulture);
ShowWhere<string>(lines, result);

Console.WriteLine($"{(result > 0 ? "Found" : "Did not find")} {searchString}");

void ShowWhere<T>(T[] array, int index)
{
    if (index < 0)
    {
        index = ~index;

        Console.Write("Not found. Sorts between: ");

        if (index == 0)
            Console.Write("beginning of sequence and ");
        else
            Console.Write($"{array[index - 1]} and ");

        if (index == array.Length)
            Console.WriteLine("end of sequence.");
        else
            Console.WriteLine($"{array[index]}.");
    }
    else
    {
        Console.WriteLine($"Found at index {index}.");
    }
}
```

Порядковая сортировка и поиск в коллекциях

Следующий код использует класс коллекции System.Collections.Generic.List<T> для хранения строк. Строки сортируются с помощью метода List<T>.Sort. Этому методу нужен делегат, который сравнивает и упорядочивает две строки. Метод String.CompareTo предоставляет эту функцию сравнения. Запустите пример и следите за порядком. Эта операция сортировки использует порядковую сортировку с учетом регистра. Можно использовать статические методы String.Compare, чтобы указать разные правила сравнения.

```
List<string> lines = new List<string>
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};

Console.WriteLine("Non-sorted order:");
foreach (string s in lines)
{
    Console.WriteLine($"   {s}");
}

Console.WriteLine("\n\rSorted order:");

lines.Sort((left, right) => left.CompareTo(right));
foreach (string s in lines)
{
    Console.WriteLine($"   {s}");
}
```

После сортировки по списку строк можно осуществлять двоичный поиск. В приведенном ниже примере показан поиск в отсортированном списке с использованием той же функции сравнения. Локальная функция `ShowWhere` показывает, где находится или находился бы искомый текст:

```
List<string> lines = new List<string>
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};
lines.Sort((left, right) => left.CompareTo(right));

string searchString = @"c:\public\TEXTFILE.TXT";
Console.WriteLine($"Binary search for <{searchString}>");
int result = lines.BinarySearch(searchString);
ShowWhere<string>(lines, result);

Console.WriteLine($"{(result > 0 ? "Found" : "Did not find")} {searchString}");

void ShowWhere<T>(IList<T> collection, int index)
{
    if (index < 0)
    {
        index = ~index;

        Console.Write("Not found. Sorts between: ");

        if (index == 0)
            Console.Write("beginning of sequence and ");
        else
            Console.Write($"{collection[index - 1]} and ");

        if (index == collection.Count)
            Console.WriteLine("end of sequence.");
        else
            Console.WriteLine($"{collection[index]}.");
    }
    else
    {
        Console.WriteLine($"Found at index {index}.");
    }
}
```

Всегда используйте один и тот же тип сравнения для сортировки и поиска. Использование разных типов сравнения приводит к неожиданным результатам.

Классы коллекций, такие как System.Collections.Hashtable, System.Collections.Generic.Dictionary<TKey,TValue>, и System.Collections.Generic.List<T>, имеют конструкторы, принимающие параметр System.StringComparer, если типом элементов или ключей является `string`. В целом, по возможности следует использовать эти конструкторы и задавать либо StringComparer.Ordinal, либо StringComparer.OrdinalIgnoreCase.

Равенство ссылок и интернирование строк

Ни один из примеров не использовал ReferenceEquals. Этот метод определяет, являются ли две строки одним и тем же объектом, что может привести к несогласованности результатов при сравнении строк. Следующий пример демонстрирует функцию *интернирования строк* в C#. При объявлении программой двух или более идентичных переменных строк компилятор сохраняет их в одном расположении. Вызвав метод ReferenceEquals, можно увидеть, что две строки фактически ссылаются на один и тот же объект в памяти. Чтобы избежать интернирования, используйте метод String.Copy. После копирования две строки имеют разное расположение хранения, хотя и имеют одинаковое значение. Запустите следующий пример, показывающий, что строки `a` и `b` *интернированы*, находятся в одном хранилище. Строки `a` и `c` таковыми не являются.

```
string a = "The computer ate my source code.";
string b = "The computer ate my source code.";

if (String.ReferenceEquals(a, b))
    Console.WriteLine("a and b are interned.");
else
    Console.WriteLine("a and b are not interned.");

string c = String.Copy(a);

if (String.ReferenceEquals(a, c))
    Console.WriteLine("a and c are interned.");
else
    Console.WriteLine("a and c are not interned.");
```

NOTE

При проверке строк на равенство нужно использовать методы, которые явно указывают, какой вид сравнения следует выполнить. Это делает код намного более понятным и удобочитаемым. Используйте перегрузки методов классов System.String и System.Array, которые принимают параметр перечисления StringComparison. Это позволяет указать тип выполняемого сравнения. Старайтесь избегать использования операторов `==` и `!=` при проверке на равенство. Методы экземпляра String.CompareTo всегда выполняют порядковое сравнение с учетом регистра. Они предназначены, прежде всего, для упорядочивания строк в алфавитном порядке.

Вы можете интернировать строку или получить ссылку на существующую интернированную строку, вызвав метод String.Intern. Чтобы определить, является ли строка интернированной, вызовите метод String.IsInterned.

См. также

- System.Globalization.CultureInfo
- System.StringComparer
- Строки
- Сравнение строк
- Глобализация и локализация приложений

Практическое руководство. Перехват несовместимого с CLS исключения

18.12.2021 • 2 minutes to read

Некоторые языки .NET, включая C++/CLI, позволяют объектам вызывать исключения, которые не являются производными от Exception. Такие исключения называются *несовместимыми с CLS исключениями* или *необработанными исключениями*. В C# невозможно вызвать несовместимые с CLS исключения, однако можно перехватить их следующими двумя способами.

- В блоке `catch (RuntimeWrappedException e)`.

  По умолчанию сборка Visual C# перехватывает несовместимые с CLS исключения как заключенные в оболочку. Этот метод следует использовать, если требуется доступ к исходному исключению, который можно получить с помощью свойства RuntimeWrappedException.WrappedException. Далее в этом разделе описывается процедура перехвата исключений таким способом.

- В общем блоке перехвата (блоке перехвата, для которого не указан тип исключения), который помещается после остальных блоков `catch`.

  Используйте этот способ, если требуется выполнить какое-либо действие (например, запись в файл журнала) в ответ на несовместимые с CLS исключения, и вам не нужен доступ к сведениям об исключении. По умолчанию среда CLR создает оболочку для всех исключений. Чтобы отключить этот режим, добавьте этот атрибут уровня сборки в код, как правило, в файле AssemblyInfo.cs: `[assembly: RuntimeCompatibilityAttribute(WrapNonExceptionThrows = false)]`.

Перехват несовместимого с CLS исключения

В блоке `catch(RuntimeWrappedException e)` для доступа к исходному исключению используйте свойство RuntimeWrappedException.WrappedException.

Пример

В следующем примере показано, как перехватить несовместимое с CLS исключение, которое было вызвано из библиотеки классов, написанной на C++/CLI. Обратите внимание, что в этом примере клиентскому коду C# заранее известно, что тип вызываемого исключения — System.String. Можно привести свойство RuntimeWrappedException.WrappedException к его исходному типу, если этот тип доступен из кода.

```
// Class library written in C++/CLI.
var myClass = new ThrowNonCLS.Class1();

try
{
    // throws gcnew System::String(
    // "I do not derive from System.Exception!");
    myClass.TestThrow();
}
catch (RuntimeWrappedException e)
{
    String s = e.WrappedException as String;
    if (s != null)
    {
        Console.WriteLine(s);
    }
}
```

См. также

- RuntimeWrappedException
- Исключения и обработка исключений

Пакет SDK для .NET Compiler Platform

18.12.2021 • 5 minutes to read

Компиляторы создают детализированную модель кода приложения по мере проверки синтаксиса и семантики этого кода. Они используют эту модель для сборки на выходе исполняемых файлов из исходного кода. Пакет SDK для .NET Compiler Platform предоставляет доступ к этой модели. Мы все шире используем возможности интегрированной среды разработки (IDE), такие как IntelliSense, рефакторинг, интеллектуальное переименование, функции "Найти все ссылки" и "Перейти к определению", чтобы повысить производительность. Мы применяем средства анализа кода, чтобы улучшить его качество, и генераторы кода, чтобы упростить разработку приложений. Эти средства становятся все более интеллектуальными, и им требуется доступ ко все большему числу моделей, которые создаются исключительно компиляторами при обработке кода приложения. Основное назначение API-интерфейсов Roslyn — открывать "черные ящики", чтобы средства и пользователи могли обмениваться данными, используя большой объем сведений о коде, доступных в компиляторе. В отличие от непрозрачной работы преобразователей исходного кода на входе в объектный код на выходе, компиляторы с использованием Roslyn выполняют функции платформ. Эти API-интерфейсы можно использовать для связанных с кодом задач в средствах и приложениях.

Основные преимущества пакета SDK для .NET Compiler Platform

Пакет SDK для .NET Compiler Platform значительно сокращает требования к созданию средств и приложений, ориентированных на код. Он предоставляет множество возможностей для инноваций в таких областях, как метапрограммирование, создание и преобразование кода, интерактивное использование языков C# и Visual Basic, а также их внедрение в доменные языки.

Пакет SDK для.NET Compiler Platform позволяет создавать ***анализаторы*** и ***средства исправления кода*** _ *для выявления и устранения ошибок.* _ *Анализаторы* изучают синтаксис (структуру кода) и семантику, а также выявляют методы, которые нужно исправить. ***Средства исправления кода*** предоставляют один или несколько рекомендуемых способов устранения ошибок в коде, выявленных анализаторами или при диагностики в компиляторе. Обычно анализатор и соответствующие средства исправления кода упакованы в один проект.

Для изучения кода в анализаторах и средствах исправления используется статический анализ. Они не запускают код и не предоставляют другие преимущества тестирования. Но они могут указывать на методы, использование которых часто приводит к ошибкам, помогают выявить неподдерживаемый код и нарушение стандартных рекомендаций.

Помимо анализаторов и средств исправления кода, пакет SDK для.NET Compiler Platform позволяет создавать ***средства для рефакторинга кода***. Пакет также предоставляет единый набор API-интерфейсов, которые позволяют проверять и анализировать базу кода C# или Visual Basic. Единая база кода упрощает создание анализаторов и средств исправления. При этом используются API-интерфейсы из пакета SDK для .NET Compiler Platform, которые позволяют проводить синтаксический и семантический анализ. Предоставьте выполнение комплексного анализа компилятору и сосредоточьтесь на более конкретных задачах. Выявляйте и устраняйте распространенные ошибки кодирования для своего проекта или библиотеки.

Дополнительным преимуществом является то, что анализаторы и средства исправления кода используют гораздо меньше памяти при загрузке в Visual Studio, чем понадобилось бы при написании собственной базы кода для анализа кода в проекте. Используя те же классы, что компилятор и Visual Studio, вы можете создавать собственные средства статического анализа. Это означает, что члены вашей команды

смогут использовать анализаторы и средства исправления кода без существенного воздействия на производительность IDE.

Есть три основных сценария использования анализаторов и средств исправления кода:

1. *Применение стандартов кодирования в команде.*
2. *Предоставление рекомендаций с пакетами библиотек.*
3. *Предоставление общих рекомендаций*

Применение стандартов кодирования в команде

У многих команд разработчиков есть стандарты кодирования, которые применяются при совместной проверке кода. Анализаторы и средства исправления кода значительно повысят эффективность этого процесса. Проверка кода происходит после того, как разработчик представит выполненную задачу на рассмотрение в команде. За весь период создания функции он не получает никаких комментариев. Разработчик может неделями приучаться к практикам разработки, которые идут вразрез со стандартами команды.

Анализаторы же начинают работу вместе с ним. Разработчик сразу же получает отзывы, которые помогают привести код в соответствие с рекомендациями. Он вырабатывает методику написания совместимого кода, как только начинает создавать прототипы. Когда функция будет готова к проверке в команде, все стандартные рекомендации будут соблюдены.

Команды могут создавать анализаторы и средства исправления кода, которые будут выявлять распространенные приемы, нарушающие командную методику написания кода. Анализаторы и средства исправления можно установить на компьютере каждого разработчика, чтобы обеспечить применение стандартов.

> **TIP**
> Прежде чем создать собственный анализатор, ознакомьтесь со встроенными анализаторами. Дополнительные сведения см. в разделе Анализ стиля кода.

Предоставление рекомендаций с пакетами библиотек

В NuGet есть широкий набор библиотек для разработчиков .NET. Некоторые из них получены из корпорации Майкрософт, некоторые — из сторонних компаний, другие же предоставлены членами сообществ и волонтерами. Такие библиотеки чаще внедряются и выше оцениваются, если разработчики успешно используют их в своих решениях.

Помимо документации, вы можете предоставить анализаторы и средства исправления кода для выявления и устранения распространенных ошибок в использовании вашей библиотеки. Эти мгновенные исправления помогут разработчикам быстрее достигать своих целей.

Вы можете упаковать анализаторы и средства исправления кода вместе с библиотекой в NuGet. Тогда каждый разработчик, который установит ваш пакет NuGet, получит и пакет с анализаторами. Все разработчики, использующие вашу библиотеку, сразу же получат рекомендации от вашей команды с уведомлением об ошибке и рекомендуемыми способами ее устранения.

Предоставление общих рекомендаций

Сообществом разработчиков .NET на практике были определены эффективные схемы и схемы, которых лучше избегать. Несколько членов сообщества создали анализаторы, в которых реализованы рекомендуемые схемы. Чем больше мы учимся, тем больше у нас возникает идей.

Эти анализаторы можно отправить в Visual Studio Marketplace. Затем разработчики смогут скачать их с помощью Visual Studio. Новички в работе с языком и платформой быстро освоят правильные методики и достигнут требуемого уровня производительности на самых ранних этапах работы с NET. Как только методики начинают широко использоваться, сообщество их принимает.

Следующие шаги

Пакет SDK для .NET Compiler Platform включает новейшие объектные модели языков для создания, анализа и оптимизации кода. Этот раздел содержит обзор основных преимуществ пакета SDK для .NET Compiler Platform. Дополнительные сведения можно найти в кратких руководствах, примерах и другой документации.

Дополнительные сведения об основных преимуществах пакета SDK для .NET Compiler Platform см. в следующих пяти разделах:

- Изучение кода с помощью визуализатора синтаксиса
- Understand the .NET Compiler Platform SDK model (Сведения о модели SDK для .NET Compiler Platform)
- Работа с синтаксисом
- Работа с семантикой
- Использование рабочей области

Чтобы приступить к работе, потребуется установить **пакет SDK для .NET Compiler Platform**:

Инструкции по установке — Visual Studio Installer

Найти SDK-**пакет .NET Compiler Platform** в **Visual Studio Installer** можно двумя способами:

Установка с помощью Visual Studio Installer — представление "Рабочие нагрузки"

SDK-пакет .NET Compiler Platform не выбирается автоматически в рамках рабочей нагрузки разработки расширений Visual Studio. Его необходимо выбрать как дополнительный компонент.

1. Запустите **Visual Studio Installer**.
2. Выберите **Изменить**.
3. Отметьте рабочую нагрузку **Разработка расширений Visual Studio**.
4. Откройте узел **Разработка расширений Visual Studio** в дереве сводки.
5. Установите флажок SDK-**пакет .NET Compiler Platform**. Нужный пакет будет представлен последним в списке дополнительных компонентов.

Кроме того, вы можете настроить **редактор** DGML для отображения диаграмм в средстве визуализации:

1. Откройте узел **Отдельные компоненты** в дереве сводки.
2. Установите флажок **Редактор** DGML

Установка с помощью Visual Studio Installer — вкладка "Отдельные компоненты"

1. Запустите **Visual Studio Installer**.
2. Выберите **Изменить**.
3. Откройте вкладку **Отдельные компоненты**.
4. Установите флажок SDK-**пакет .NET Compiler Platform**. Нужный пакет будет представлен в разделе **Компиляторы, средства сборки и среды выполнения** в самом начале.

Кроме того, вы можете настроить **редактор** DGML для отображения диаграмм в средстве визуализации:

1. Установите флажок **Редактор** DGML. Нужный пакет будет представлен в разделе **Средства для**

работы с кодом.

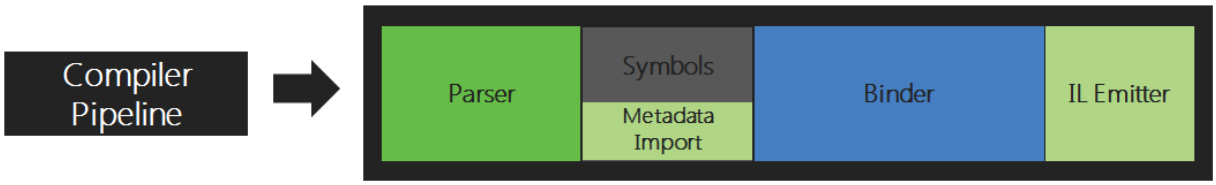
Общие сведения о модели пакета SDK для .NET Compiler Platform

18.12.2021 • 2 minutes to read

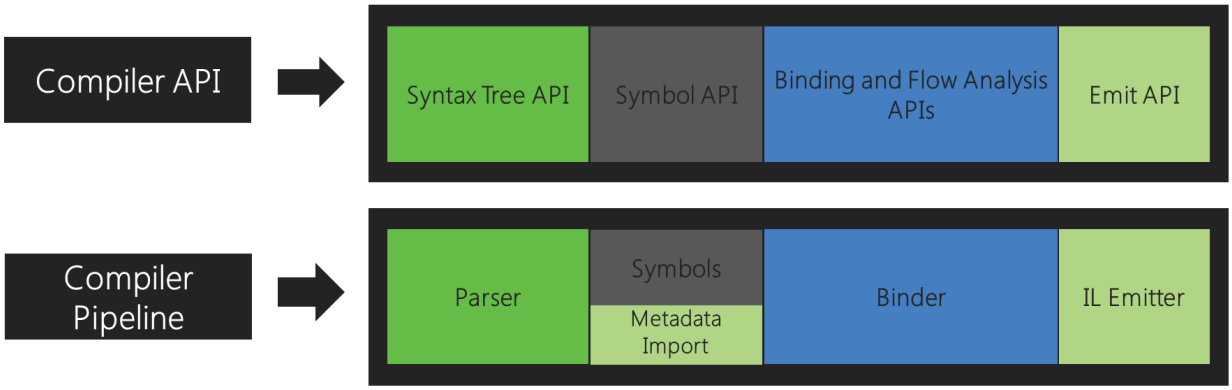
Компиляторы обрабатывают код, который вы пишете, следуя структурированным правилам, которые часто отличаются от того, как читают и воспринимают код люди. Понимание принципов работы модели, используемой компиляторами, важно для изучения API, используемых при создании средств на основе Roslyn.

Функциональные области конвейера компилятора

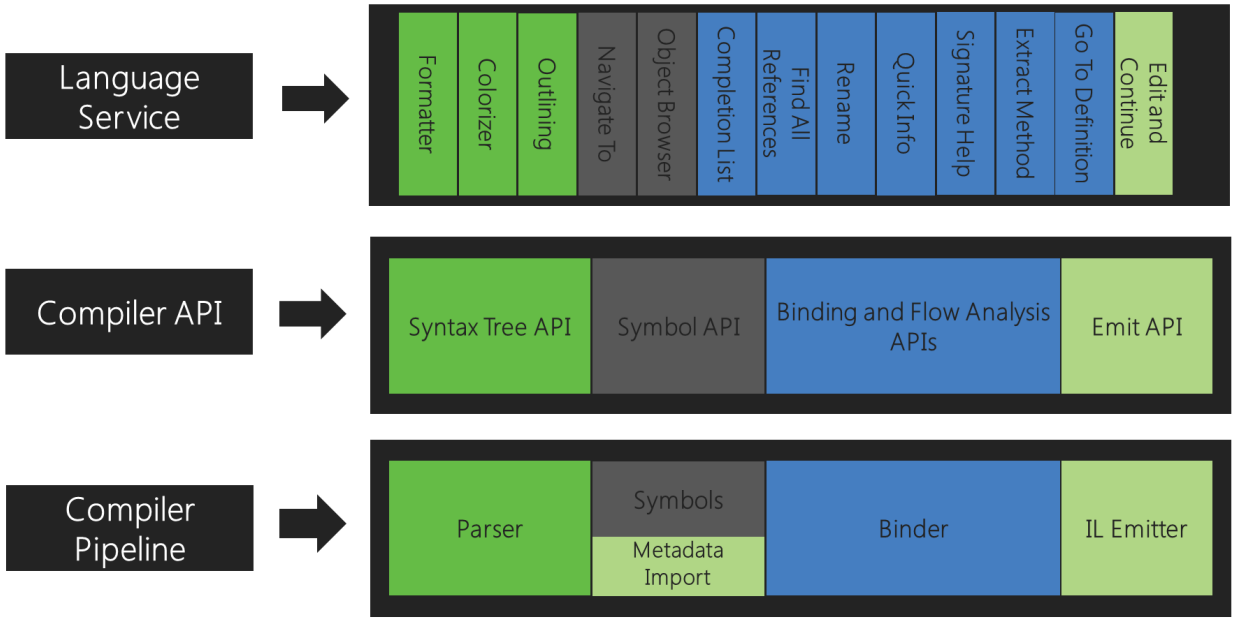
Пакет SDK для .NET Compiler Platform реализует функции анализа кода в компиляторах C# и Visual Basic, предоставляя слой API, отражающий традиционный конвейер компилятора.



Каждый этап конвейера — это отдельный компонент. Сначала на этапе синтаксического анализа выполняется расстановка токенов и анализ текста исходного кода с учетом синтаксиса, а затем и грамматики языка. Потом на этапе объявления анализируется источник и импортированные метаданные для формирования именованных символов. Далее на этапе привязки идентификаторы в коде сопоставляются с символами. Наконец, на этапе порождения создается сборка со всеми сведениями, сформированными компилятором.



В соответствии с каждым из этих этапов пакет SDK для .NET Compiler Platform предоставляет объектную модель, обеспечивающую доступ к информации на этом этапе. Этап анализа предоставляет дерево синтаксиса, этап объявления предоставляет таблицу иерархических символов, этап привязки предоставляет результат семантического анализа компилятора, а этап порождения — это API, создающий коды байтов IL.



Каждой компилятор сочетает в себе эти компоненты, образуя единый процесс.

Эти API также используются в Visual Studio. Например, функции структурирования и форматирования кода используют деревья синтаксиса, **обозреватель объектов** и функции навигации — таблицу символов, рефакторинг и операции **Перейти к определению** — семантическую модель, а операция **Изменить и продолжить** использует их все, включая API порождения.

Слои API

Пакет SDK компилятора .NET состоит из нескольких основных слоев API: API компилятора, API диагностики, API скриптов и API рабочих областей.

API компилятора

Слой компилятора содержит объектные модели, которые соответствуют сведениям, представляемым на каждом этапе конвейера компилятора, как синтаксическим, так и семантическим. Слой компилятора также содержит неизменяемый моментальный снимок однократного вызова компилятора, включая ссылки на сборки, параметры компилятора и файлы исходного кода. Существует два отдельных API, представляющих язык C# и Visual Basic. Эти два API похожи по форме, но для обеспечения высокой точности привязаны к конкретному языку. Этот слой не имеет зависимостей от компонентов Visual Studio.

API диагностики

В процессе анализа компилятор может создать набор данных диагностики, охватывающий все — от синтаксиса, семантики и явных ошибок присваивания до различных предупреждений и информационных диагностических сообщений. Слой API компилятора предоставляет диагностику посредством расширяемого API, который разрешает подключать пользовательские анализаторы к процессу компиляции. Она позволяет создавать заданные пользователем диагностические данные, например, формируемые такими программами, как StyleCop, параллельно с диагностическими данными, определенными компилятором. Такое формирование данных диагностики дает преимущество естественной интеграции со средствами, такими как MSBuild и Visual Studio, которые зависят от диагностики при применении таких процедур, как прерывание сборки на основе политики, отображение динамических волнистых линий в редакторе и предложение исправлений кода.

API скриптов

API размещения и скриптов являются частью слоя компилятора. Их можно использовать для выполнения фрагментов кода и накопления контекста выполнения во время выполнения. Интерактивная среда REPL (read-evaluate-print loop) C# использует эти API. REPL позволяет использовать C# в качестве скриптового

языка C#, выполняя код в интерактивном режиме по мере его написания.

API рабочих областей

Слой рабочих областей содержит соответствующий API, который является отправной точкой для анализа кода и рефакторинга в рамках целых решений. Он помогает упорядочить всю информацию о проектах в решении в рамках одной объектной модели, предоставляя вам прямой доступ к объектным моделям слоя компилятора без необходимости анализировать файлы, настраивать параметры или управлять зависимостями между проектами.

Кроме того, слой рабочих областей предоставляет набор API, используемый реализацией средств анализа кода и рефакторинга, которые работают в среде размещения, таких как Visual Studio IDE. В качестве примеров можно привести API "Найти все ссылки", "Форматирование" и "Создание кода".

Этот слой не имеет зависимостей от компонентов Visual Studio.

Работа с синтаксисом

18.12.2021 • 7 minutes to read

Дерево синтаксиса представляет собой базовую неизменяемую структуру данных, предоставляемую API компилятора. Эти деревья представляют лексическую и синтаксическую структуру исходного кода. Они служат для двух важных целей:

- Чтобы позволить средствам, таким как интегрированная среда разработки, надстройки, средства анализа кода и рефакторинг, просматривать и обрабатывать синтаксическую структуру исходного кода в проекте пользователя.
- Чтобы позволить средствам, таким как рефакторинг и интегрированная среда разработки, создавать, изменять и переупорядочивать исходный код естественным образом без прямых изменений текста. Создавая деревья и управляя ими, средства легко могут создавать и переупорядочивать исходный код.

Деревья синтаксиса

Деревья синтаксиса образуют первичную структуру, используемую для компиляции, анализа кода, привязки, рефакторинга, функций интегрированной среды разработки и создания кода. Любая часть исходного кода становится понятной только после того, как будет идентифицирована и классифицирована по одному из многих известных структурных элементов языка.

Деревья синтаксиса имеют три ключевых атрибута.

- Они хранят всю информацию об источнике во всей ее полной точности. Полная точность означает, что дерево синтаксиса содержит всю информацию, отраженную в исходном тексте, каждую грамматическую конструкцию, каждую лексему, а также все промежуточные элементы, включая пробелы, комментарии и директивы препроцессора. Например, каждый упомянутый в исходном коде литерал представлен точно так, как был указан. Деревья синтаксиса также перехватывают ошибки в исходном коде, если программа является неполной или неправильно сформированной из-за пропущенных или отсутствующих токенов.
- С их помощью можно создать точный текст, из которого они были сформированы. От любого синтаксического узла можно получить текстовое представление поддерева, корнем которого он является. Такая возможность означает, что деревья синтаксиса можно использовать для создания и изменения текста исходного кода. Создавая дерево, вы косвенно создаете эквивалентный текст, а получая новое дерево на основе изменений существующего, вы фактически редактируете текст.
- Они являются неизменяемыми и потокобезопасными. После получения дерево представляет собой моментальный снимок текущего состояния кода и никогда не меняется. Это позволяет нескольким пользователям одновременно взаимодействовать с одним деревом синтаксиса в разных потоках без блокировки или дублирования. Так как деревья являются неизменяемыми, то есть напрямую в них невозможно внести никакие изменения, фабричные методы помогают создавать и изменять деревья синтаксиса путем создания их дополнительных снимков. Деревья эффективно используют базовые узлы повторно, поэтому новую версию можно перестроить быстро и с небольшими затратами памяти.

Дерево синтаксиса фактически является древовидной структурой данных, где нетерминальные структурные элементы являются родительскими для других элементов. Каждое дерево синтаксиса состоит из узлов, токенов и дополнительной синтаксической информации (trivia).

Синтаксические узлы

Синтаксические узлы являются одним из основных элементов деревьев синтаксиса. Они представляют такие синтаксические конструкции, как объявления, операторы, предложения и выражения. Каждая категория синтаксических узлов представлена отдельным классом, производным от Microsoft.CodeAnalysis.SyntaxNode. Набор классов узлов не является расширяемым.

Все синтаксические узлы являются нетерминальными узлами в дереве синтаксиса, значит, всегда имеют дочерние элементы в виде других узлов и токенов. Как дочерний элемент другого узла каждый узел имеет родительский узел, к которому можно обратиться с помощью свойства SyntaxNode.Parent. Так как узлы и деревья являются неизменяемыми, родительский элемент узла никогда не меняется. Корень дерева имеет родительский элемент null.

Каждый узел имеет метод SyntaxNode.ChildNodes(), возвращающий список дочерних узлов в последовательном порядке с учетом их позиции в тексте исходного кода. Этот список не содержит токены. Каждый узел также имеет методы для изучения дочерних объектов, такие как DescendantNodes, DescendantTokens или DescendantTrivia. Они представляют список всех узлов, токенов или дополнительной синтаксической информации, которые присутствуют в поддереве, являющемся корнем этого узла.

Кроме того, каждый подкласс синтаксических узлов предоставляет те же дочерние элементы посредством строго типизированных свойств. Например, класс узлов BinaryExpressionSyntax имеет три дополнительные свойства для отдельных бинарных операторов: Left, OperatorToken и Right. Типом Left и Right является ExpressionSyntax, а типом OperatorToken — SyntaxToken.

Некоторые синтаксические узлы имеют дополнительные дочерние элементы. Например, IfStatementSyntax имеет дополнительный ElseClauseSyntax. Если дочерний элемент не указан, свойство возвращает значение null.

Синтаксические токены

Синтаксические токены являются терминалами грамматики языка, представляющими наименьшие синтаксические фрагменты кода. Они никогда не бывают родителями других узлов или токенов. Синтаксические токены состоят из ключевых слов, идентификаторов, литералов и знаков препинания.

Для повышения эффективности тип SyntaxToken является типом значений среды CLR. Таким образом, в отличие от синтаксических узлов существует всего одна структура для всех видов токенов с набором свойств, значение которых зависит от типа представленного токена.

Например, токен целочисленного литерала представляет числовое значение. Кроме необработанного текста исходного кода, охватываемого токеном, токен литерала имеет свойство Value, сообщающее точное декодированное целочисленное значение. Это свойство имеет тип Object, так как может быть одним из многих типов-примитивов.

Свойство ValueText сообщает те же сведения, что и свойство Value, однако оно всегда имеет тип String. Идентификатор в тексте исходного кода C# может включать escape-символы Юникода, при этом синтаксис самой escape-последовательности не считается частью имени идентификатора. Поэтому хотя охватываемый токеном необработанный текст и включает escape-последовательность, свойство ValueText ее не содержит. Вместо этого оно включает в себя символы Юникода, определяемые этой escape-последовательностью. Например, если текст исходного кода содержит идентификатор, записанный как `\u03C0`, то свойство ValueText для этого токена возвратит `π`.

Дополнительная синтаксическая информация

Дополнительная синтаксическая информация (syntax trivia) представляет части текста в исходном коде, которые очень мало влияют на понимание кода, например пробелы, комментарии и директивы препроцессора. Как и синтаксические токены, элементы trivia являются типами значений. Для описания

всех видов trivia используется один тип Microsoft.CodeAnalysis.SyntaxTrivia.

Так как элементы trivia не являются частью обычного синтаксиса и могут стоять в любом месте между двумя любыми токенами, они не включаются в дерево синтаксиса в качестве дочернего элемента узла. Хотя, так как они важны при реализации таких функций, как рефакторинг, и обеспечения полного соответствия тексту исходного кода, они входят в состав дерева синтаксиса.

Вы можете обратиться к trivia, просмотрев коллекции SyntaxToken.LeadingTrivia или SyntaxToken.TrailingTrivia токена. При синтаксическом анализе текста исходного кода последовательности trivia сопоставляются с токенами. Как правило, токену принадлежит любой элемент trivia, расположенный в той же строке до следующего токена. Любой элемент trivia после этой строки сопоставляется со следующим токеном. Первый токен в исходном файле получает все начальные элементы trivia, а последняя последовательность trivia в файле присоединяется к токену конца файла, который в противном случае имеет нулевую ширину.

В отличие от синтаксических узлов и токенов элементы syntax trivia не имеют родительских элементов. Однако, так как они являются частью дерева и каждый из них сопоставлен с одним токеном, к соответствующему токену можно получить доступ с помощью свойства SyntaxTrivia.Token.

Диапазоны

Каждому узлу, токену или элементу trivia известно его место в тексте исходного кода и число символов, из которых он состоит. Положение в тексте представлено в виде 32-разрядного целого числа, являющегося отсчитываемым от нуля индексом `char`. Объект TextSpan обозначает начальное положение и количество символов, представленные в виде целых чисел. Если TextSpan имеет нулевую длину, он обозначает расположение между двумя символами.

У каждого узла есть два свойства TextSpan: Span и FullSpan.

Свойство Span соответствует текстовому диапазону от начала первого токена в поддереве узла до конца последнего токена. Этот диапазон не включает в себя никакие начальные или конечные элементы trivia.

Свойство FullSpan является текстовым диапазоном, включающим в себя обычный диапазон узла, а также диапазон любых начальных или конечных элементов trivia.

Пример:

```
      if (x > 3)
      {
||        // this is bad
          |throw new Exception("Not right.");|  // better exception?||
      }
```

Узел оператора внутри блока имеет диапазон, обозначаемый отдельными вертикальными чертами (|). Он включает символы `throw new Exception("Not right.");`. Полный диапазон обозначается двойными вертикальными чертами (||). Он содержит те же символы, что и обычный диапазон, а также символы, связанные с начальными или конечными элементами trivia.

Типы

Каждый узел, токен или элемент trivia имеет свойство SyntaxNode.RawKind типа System.Int32, определяющее конкретный представленный элемент синтаксиса. Это значение можно привести к перечислению определенного языка. Каждый язык, C# или Visual Basic имеет одно перечисление `SyntaxKind` (Microsoft.CodeAnalysis.CSharp.SyntaxKind и Microsoft.CodeAnalysis.VisualBasic.SyntaxKind соответственно), содержащее все возможные узлы, токены и другие элементы грамматики. Это преобразование может выполняться автоматически с помощью методов расширения

CSharpExtensions.Kind или VisualBasicExtensions.Kind.

Свойство RawKind позволяет легко устранить неоднозначность типов синтаксических узлов, которые используют один класс узлов. Для токенов и элементов trivia это свойство является единственным способом отличить один тип элемента от другого.

Например, один класс BinaryExpressionSyntax имеет дочерние элементы Left, OperatorToken и Right. Свойство Kind позволяет определить, имеет ли этот синтаксический узел тип AddExpression, SubtractExpression или MultiplyExpression.

> **TIP**
> Рекомендуется проверять типы с помощью методов расширения IsKind (для C#) или IsKind (для VB).

Ошибки

Даже если текст исходного кода содержит синтаксические ошибки, предоставляется полное дерево синтаксиса, которое обеспечивает круговой путь до источника. Когда средство синтаксического анализа обнаруживает код, который не соответствует заданному синтаксису языка, для создания дерева синтаксиса используется один из двух способов.

- Если средство синтаксического анализа ожидает определенный тип токена, но не находит его, оно может добавить отсутствующий токен в дерево синтаксиса в том расположении, где он ожидался. Отсутствующий токен представляет фактический ожидаемый токен, однако он имеет пустой диапазон, а его свойство SyntaxNode.IsMissing возвращает `true`.
- Средство синтаксического анализа может пропускать токены, пока не найдет такой, с которого может продолжить анализ. В этом случае пропущенные токены добавляются в виде узла trivia типа SkippedTokensTrivia.

Работа с семантикой

18.12.2021 • 2 minutes to read

Деревья синтаксиса представляют лексическую и синтаксическую структуру исходного кода. Хотя одной этой информации достаточно для описания всех объявлений и логики в источнике, ее недостаточно для определения того, на что указывает ссылка. Имя может представлять:

- тип;
- поле;
- метод;
- локальную переменную.

Хотя каждый из этих элементов имеет уникальные отличия, чтобы определить, на какой из них ссылается идентификатор, часто требуется хорошо разбираться в правилах языка.

В исходном коде представлены программные элементы, а программы также могут ссылаться на ранее скомпилированные библиотеки, упакованные в файлы сборки. Хотя исходный код, а, следовательно, и синтаксические узлы и деревья синтаксиса для сборок недоступны, программы по-прежнему могут ссылаться на элементы внутри них.

Для этих задач требуется **семантическая модель**.

Дополняя синтаксическую модель исходного кода, семантическая модель инкапсулирует правила языка, позволяя вам легко соотнести идентификаторы с программным элементом, на который указывает ссылка.

Компиляция

Компиляция является представлением всего, что необходимо для компиляции программы C# или Visual Basic, куда входят все ссылки на сборки, параметры компилятора и исходные файлы.

Так как эти сведения находятся в одном месте, элементы, содержащиеся в исходном коде, можно описать более подробно. Компиляция представляет каждый объявленный тип, член или переменную в виде символа. Компиляция содержит различные методы, помогающие найти и соотнести символы, которые были объявлены в исходном коде или импортированы в виде метаданных из сборки.

Аналогично деревьям синтаксиса компиляции являются неизменяемыми. После создания компиляция не может быть изменена ни вами, ни другим человеком, совместно с которым вы ее используете. Однако вы можете создать компиляцию из существующей компиляции, указав при этом некоторое изменение. Например, можно создать компиляцию, которая во всем аналогична существующей компиляции, за исключением того, что она может включать дополнительный исходный файл или ссылку на сборку.

Символы

Символ представляет отдельный элемент, объявленный исходным кодом или импортированный из сборки в виде метаданных. Символом представлены все пространства имен, типы, методы, свойства, поля, события, параметры или локальные переменные.

Различные методы и свойства в типе Compilation помогают вам искать символы. Например, можно найти символ для объявленного типа по его общему имени метаданных. Можно также получить доступ к целой таблице символов в виде дерева, корнем которого является глобальное пространство имен.

Символы также содержат дополнительную информацию, которую компилятор определяет из источника или метаданных, например другие символы, на которые указывает ссылка. Каждый тип символа представлен отдельным интерфейсом, производным от ISymbol, а также имеет собственные методы и свойства, описывающие собранные компилятором сведения. Многие из этих свойств напрямую ссылаются на другие символы. Например, свойство IMethodSymbol.ReturnType сообщает вам фактический символ типа, который возвращает метод.

Символы представляют общее представление пространств имен, типов и членов между исходным кодом и метаданными. Например, метод, который был объявлен в исходном коде, и метод, который был импортирован из метаданных, оба представлены IMethodSymbol с одинаковыми свойствами.

По своей сути символы похожи на систему типов среды CLR, представленную API System.Reflection, однако их возможности шире, так как они моделируют не только типы. Пространства имен, локальные переменные и метки — все это символы. Кроме того, символы являются представлением концепций языка, а не концепций среды CLR. Есть много похожего, но имеется и большое количество значимых различий. Например, метод итератора в C# или Visual Basic является отдельным символом. Однако когда метод итератора преобразуется в метаданные среды CLR, он превращается в тип и несколько методов.

Семантическая модель

Семантическая модель представляет всю семантическую информацию для одного исходного файла. Ее можно использовать для обнаружения:

- символов, на которые указывает ссылка в определенном месте в исходном коде;
- результирующего типа выражения;
- всех данных диагностики, относящихся к ошибкам и предупреждениям;
- перемещений переменных в регионы исходного кода и из них;
- ответов на более отвлеченные вопросы.

Использование рабочей области

18.12.2021 • 2 minutes to read

Слой **Рабочие области** является отправной точкой для анализа кода и рефакторинга в рамках целых решений. В этом слое API рабочей области помогает упорядочить всю информацию о проектах в решении в рамках одной объектной модели, предоставляя вам прямой доступ к объектным моделям слоя компилятора, например тексту исходного кода, деревьям синтаксиса, семантическим моделям и компиляциям, без необходимости анализировать файлы, настраивать параметры или управлять зависимостями между проектами.

Среды размещения, такие как интегрированная среда разработки, предоставляют рабочую область, соответствующую открытому решению. Эту модель можно использовать и вне интегрированной среды разработки, просто загрузив файл решения.

Рабочая область

Рабочая область является активным представлением вашего решения в виде коллекции проектов, каждый из которых содержит коллекцию документов. Рабочая область обычно привязывается к среде размещения, которая постоянно изменяется по мере того, как пользователь вводит или использует свойства.

Workspace предоставляет доступ к текущей модели решения. При возникновении изменения в среде размещения рабочая область выдает соответствующие события, после чего обновляется свойство Workspace.CurrentSolution. Например, когда пользовательские типы в текстовом редакторе соответствуют одному из исходных документов, рабочая область использует событие, чтобы сообщить всей модели об изменении решения и измененном документе. После этого вы можете отреагировать на такие изменения, проанализировав новую модель на корректность, выделив значимые области или внеся предложение по изменению кода.

Кроме того, вы можете создать автономные рабочие области, которые отключены от среды размещения или используются в приложении без такой среды.

Решения, проекты и документы

Хотя рабочая область может меняться при каждом нажатии клавиши, вы можете работать с моделью решения изолированно.

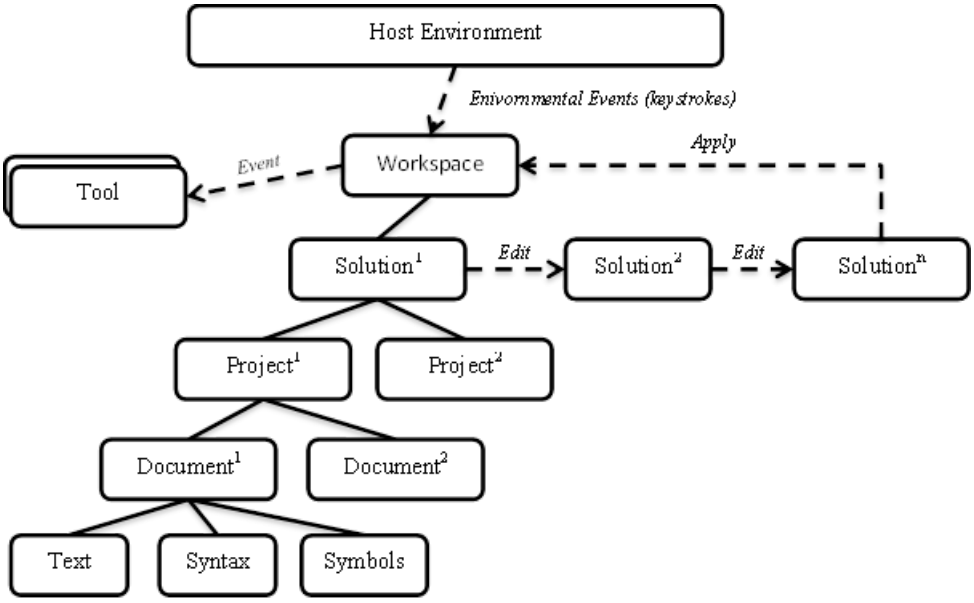
Решение представляет собой неизменяемую модель проектов и документов. Это означает, что модель можно использовать совместно без блокировки или дублирования. После получения экземпляра решения из свойства Workspace.CurrentSolution этот экземпляр больше не меняется. Однако, как и в случае с деревьями синтаксиса и компиляциями, вы можете изменять решения, создавая экземпляры на основе существующих решений и определенных изменений. Чтобы рабочая область отражала ваши изменения, нужно явным образом применить измененное решение обратно к ней.

Проект является частью общей неизменяемой модели решения. Он представляет все документы исходного кода, параметры синтаксического анализа и компиляции, а также ссылки на сборки и ссылки между проектами. Из проекта вы можете обратиться к соответствующей компиляции без необходимости определять зависимости проекта или анализировать исходные файлы.

Документ также является частью общей неизменяемой модели решения. Он представляет один исходный файл, из которого можно обратиться к тексту файла, дереву синтаксиса и семантической

модели.

Следующая схема показывает, как рабочая область связана со средой размещения, средствами и способом внесения изменений.



Сводка

Roslyn содержит набор API компиляторов и API рабочих областей, предоставляющий подробные сведения об исходном коде и обеспечивающий полную точность для языков C# и Visual Basic. Пакет SDK для .NET Compiler Platform значительно сокращает требования к созданию средств и приложений, ориентированных на код. Он предоставляет множество возможностей для инноваций в таких областях, как метапрограммирование, создание и преобразование кода, интерактивное использование языков C# и Visual Basic, а также их внедрение в доменные языки.

Изучение кода с помощью визуализатора синтаксиса Roslyn в Visual Studio

18.12.2021 • 7 minutes to read

Эта статья содержит обзор средства визуализатора синтаксиса, входящего в состав пакета SDK для .NET Compiler Platform ("Roslyn"). Визуализатор синтаксиса — окно инструментов, помогающее просматривать и изучать деревья синтаксиса. Это средство играет важную роль, так как понимает понять модели для кода, который требуется проанализировать. Оно также помогает выполнять отладку при разработке приложений с помощью пакета SDK для .NET Compiler Platform ("Roslyn"). Откройте это средство, чтобы создать свои первые анализаторы. Визуализатор помогает понять модели, используемые API-интерфейсами. Вы также можете использовать такие средства, как SharpLab или LINQPad, для просмотра кода и изучения деревьев синтаксиса.

Инструкции по установке — Visual Studio Installer

Найти SDK-**пакет** .NET Compiler Platform в Visual Studio Installer можно двумя способами:

Установка с помощью Visual Studio Installer — представление "Рабочие нагрузки"

SDK-пакет .NET Compiler Platform не выбирается автоматически в рамках рабочей нагрузки разработки расширений Visual Studio. Его необходимо выбрать как дополнительный компонент.

1. Запустите Visual Studio Installer.
2. Выберите **Изменить**.
3. Отметьте рабочую нагрузку **Разработка расширений** Visual Studio.
4. Откройте узел **Разработка расширений** Visual Studio в дереве сводки.
5. Установите флажок SDK-**пакет** .NET Compiler Platform. Нужный пакет будет представлен последним в списке дополнительных компонентов.

Кроме того, вы можете настроить **редактор** DGML для отображения диаграмм в средстве визуализации:

1. Откройте узел **Отдельные компоненты** в дереве сводки.
2. Установите флажок **Редактор** DGML

Установка с помощью Visual Studio Installer — вкладка "Отдельные компоненты"

1. Запустите Visual Studio Installer.
2. Выберите **Изменить**.
3. Откройте вкладку **Отдельные компоненты**.
4. Установите флажок SDK-**пакет** .NET Compiler Platform. Нужный пакет будет представлен в разделе **Компиляторы, средства сборки и среды выполнения** в самом начале.

Кроме того, вы можете настроить **редактор** DGML для отображения диаграмм в средстве визуализации:

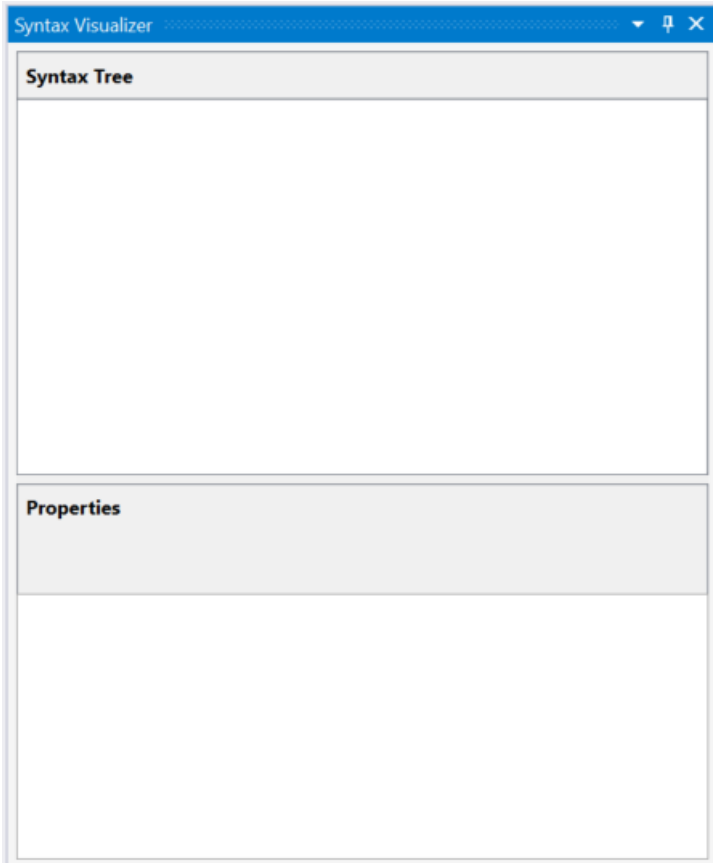
1. Установите флажок **Редактор** DGML. Нужный пакет будет представлен в разделе **Средства для работы с кодом**.

Вы можете ознакомиться с основными понятиями, используемыми в пакете SDK для .NET Compiler Platform, прочитав обзорную статью. Она содержит вводные сведения о деревьях синтаксиса, узлах, токенах и дополнительной информации.

Визуализатор синтаксиса

Визуализатор синтаксиса позволяет просматривать дерево синтаксиса для файла кода C# или Visual Basic в текущем активном окне редактора внутри интегрированной среды разработки Visual Studio. Визуализатор можно запустить, выбрав **Вид** > **Другие окна** > **Визуализатор синтаксиса**. Вы также можете воспользоваться панелью инструментов **Быстрый запуск** в правом верхнем углу. Введите "синтаксис", после чего должна появиться команда, позволяющая открыть **визуализатор синтаксиса**.

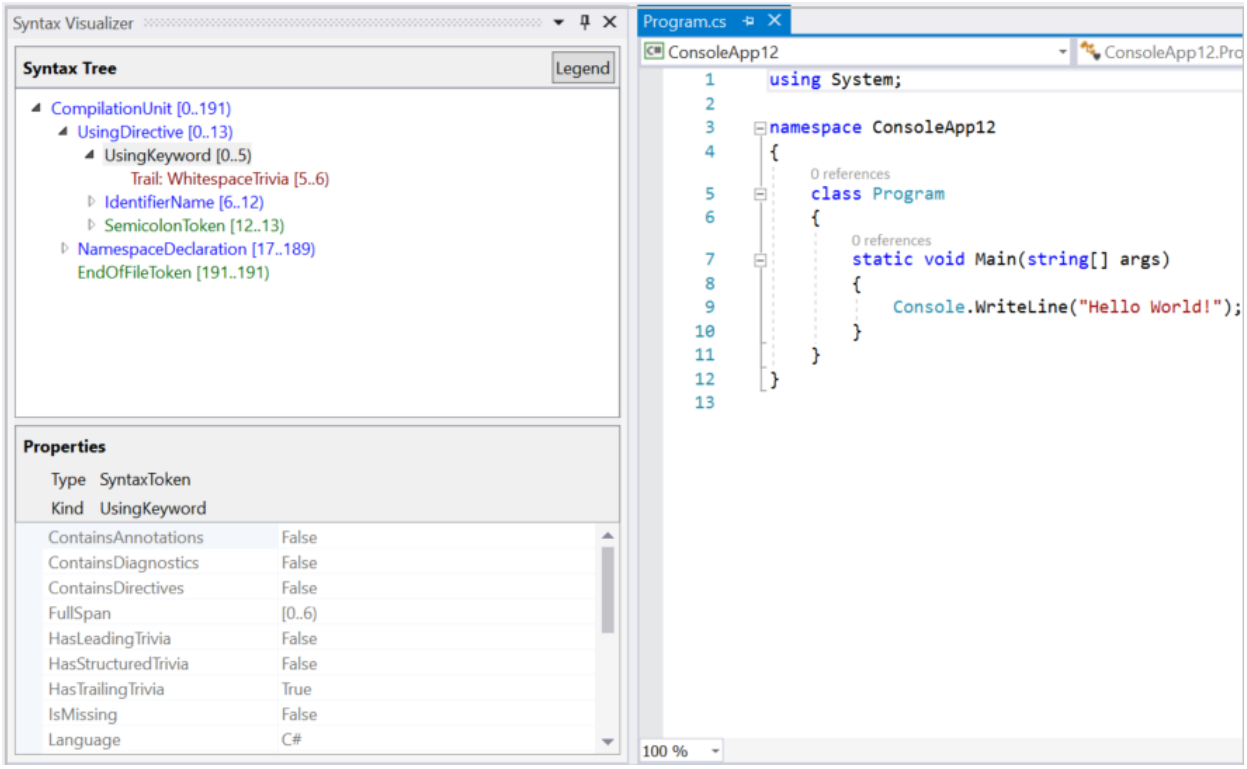
Эта команда открывает визуализатор синтаксиса в виде перемещаемого окна инструментов. Если у вас не открыто окно редактора кода, никакие данные не отображаются, как показано на рисунке ниже.



Закрепите это окно инструментов в удобном месте среды Visual Studio, например слева. Визуализатор отображает сведения о текущем файле кода.

Создайте проект с помощью команды **Файл** > **Создать проект**. Можно создать проект Visual Basic или C#. Когда Visual Studio открывает основной файл кода для этого проекта, визуализатор отображает для него дерево синтаксиса. Вы можете открыть любой существующий файл C#/Visual Basic в этом экземпляре Visual Studio, и визуализатор отобразит для него дерево синтаксиса. Если в Visual Studio открыто несколько файлов кода, визуализатор отображает дерево синтаксиса для активного файла (на который установлен фокус ввода).

- C#
- Visual Basic



Как показано на рисунках выше, окно инструментов визуализатора отображает дерево синтаксиса сверху и сетку свойств снизу. Сетка свойств содержит свойства элемента, выбранного в дереве, включая *Тип* .NET и *Вид* (SyntaxKind) этого элемента.

Деревья синтаксиса состоят из трех типов элементов — *узлов*, *токенов* и *дополнительной информации*. Подробнее об этих типах см. в статье Работа с синтаксисом. Элементы каждого типа выделены определенным цветом. Нажмите кнопку "Условные обозначения", чтобы просмотреть используемые цвета.

Каждый элемент в дереве также отображает свой **диапазон**. **Диапазон** — это индексы (начальное и конечное положение) узла в текстовом файле. В предыдущем примере C# выбранный токен "UsingKeyword [0..5)" имеет **диапазон** шириной пять символов — [0..5). Нотация "[..)" означает, что начальный индекс входит в диапазон, а конечный — нет.

Перемещаться по дереву можно двумя способами:

- Разверните или щелкните элементы в дереве. Визуализатор автоматически выбирает текст, соответствующий диапазону этого элемента в редакторе кода.
- Щелкните или выберите текст в редакторе кода. Если в предыдущем примере Visual Basic выбрать в редакторе кода строку, содержащую "Module Module1", визуализатор автоматически переходит к соответствующему узлу ModuleStatement в дереве.

Визуализатор выделяет в дереве элемент, диапазон которого лучше всего соответствует диапазону текста, выделенного в редакторе.

Визуализатор обновляет дерево, чтобы отразить изменения в активном файле кода. Добавьте вызов `Console.WriteLine()` внутрь `Main()`. По мере ввода визуализатор обновляет дерево.

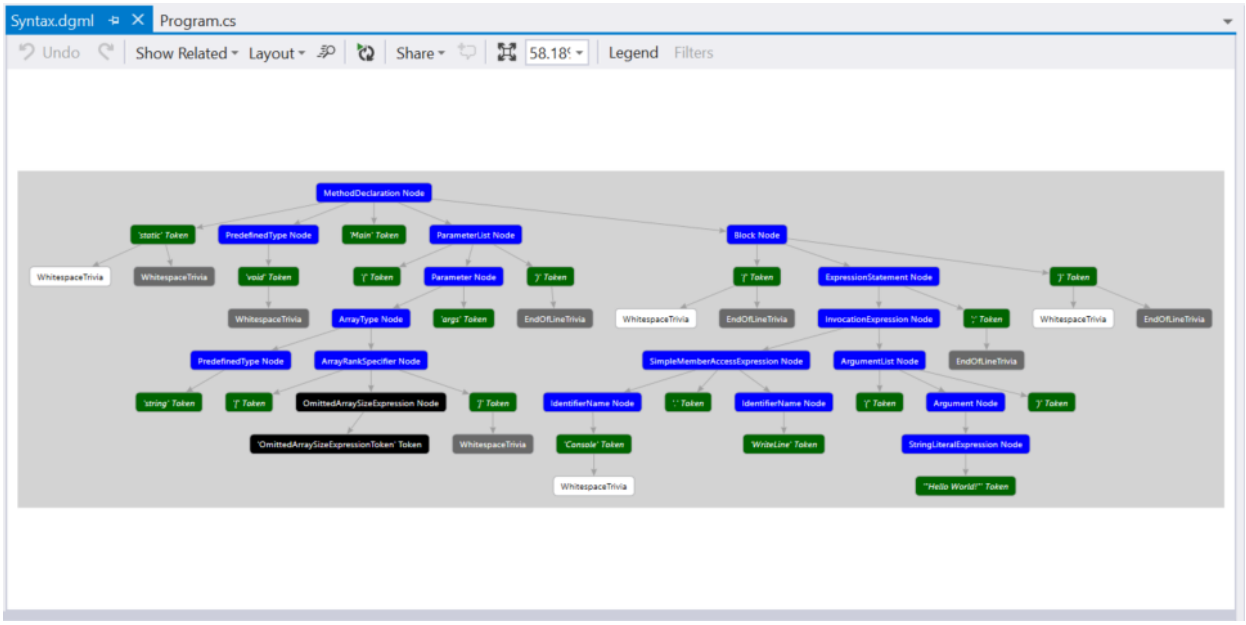
Введя `Console.`, остановитесь. Некоторые элементы в дереве выделены розовым цветом. На данном этапе введенный код содержит ошибки (это также называется диагностикой). Эти ошибки присоединяются к узлам, токенам и дополнительной информации в дереве синтаксиса. Визуализатор показывает, к каким элементам присоединены ошибки, выделяя их фон розовым цветом. Вы можете просмотреть ошибки для любого элемента, выделенного розовым цветом, наведя на него указатель мыши. Визуализатор отображает только синтаксические ошибки (связанные с синтаксисом введенного кода); семантические ошибки он не показывает.

Графы синтаксиса

Щелкните правой кнопкой мыши любой элемент в дереве и выберите команду **View Directed Syntax Graph** (Просмотреть направленный граф синтаксиса).

- C#
- Visual Basic

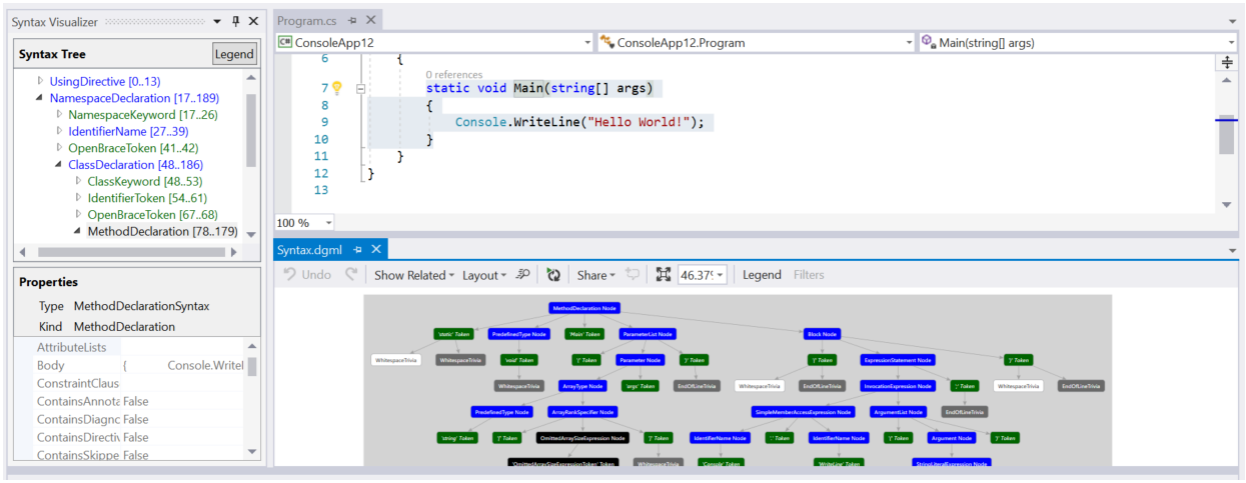
Визуализатор отображает графическое представление поддерева, корнем которого является выбранный элемент. Повторите эти шаги для узла **MethodDeclaration**, соответствующего методу `Main()` в примере C#. Визуализатор отображает граф синтаксиса, который выглядит следующим образом:



Средство просмотра графов синтаксиса имеет параметр для отображения условных обозначений для цветовой схемы. Кроме того, можно навести указатель мыши на отдельные элементы графа синтаксиса, чтобы просмотреть соответствующие им свойства.

Можно многократно просматривать графы синтаксиса для разных элементов в дереве многократно, и эти графы всегда будут отображаться в одном и том же окне Visual Studio. Это окно можно закрепить в удобном месте среды Visual Studio, чтобы не нужно было переключаться между вкладками для просмотра нового графа синтаксиса. Обычно это удобнее всего сделать в нижней части под окнами редактора кода.

Ниже представлен макет закрепления для окна инструментов визуализатора и окна графа синтаксиса:

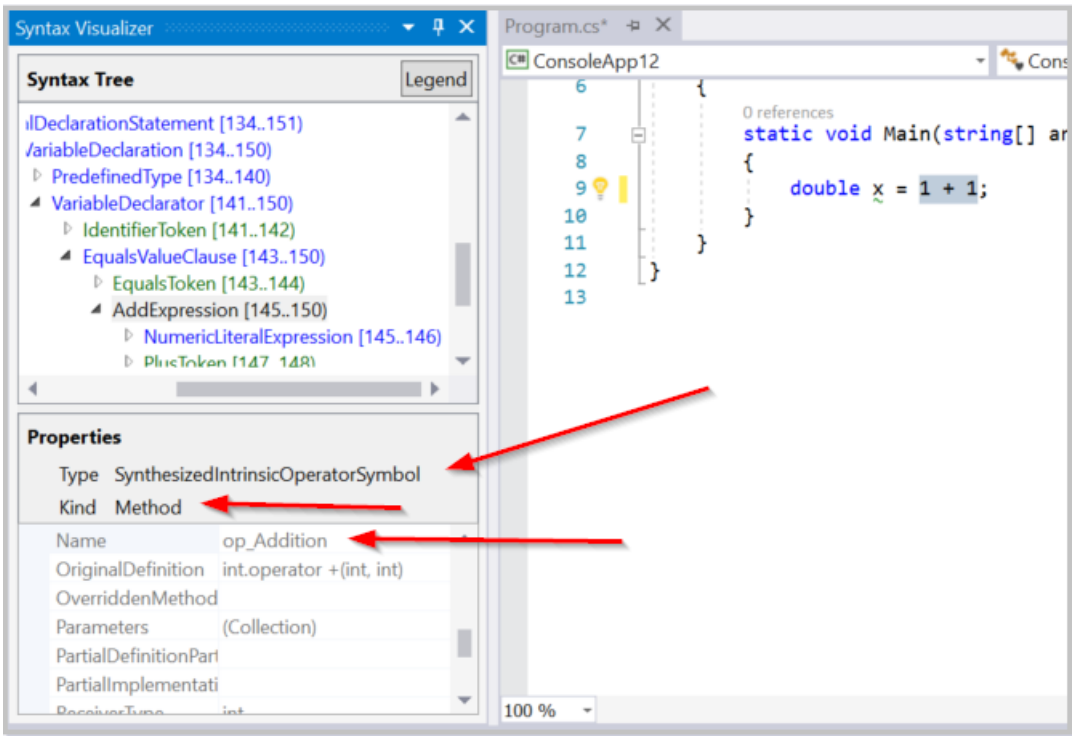


При работе с двумя мониторами можно также вывести окно графа синтаксиса на второй монитор.

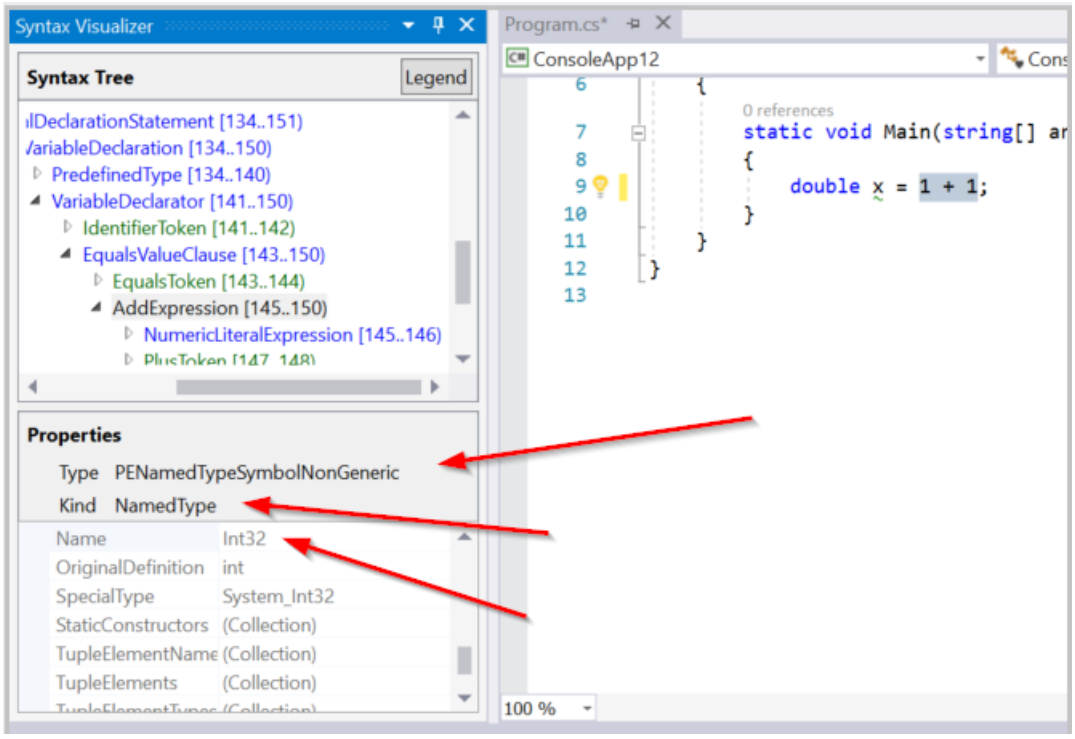
Изучение семантики

Визуализатор синтаксиса обладает простейшими функциями для просмотра символов и семантической информации. Введите `double x = 1 + 1;` внутри Main() в примере C#. Выберите выражение `1 + 1` в окне редактора кода. Визуализатор выделяет узел **AddExpression**. Щелкните правой кнопкой мыши узел **AddExpression** и выберите команду **View Symbol (if any)** (Просмотреть символ (при наличии)). Обратите внимание, что большинство пунктов меню имеют квалификатор "if any" (при наличии). Визуализатор синтаксиса проверяет свойства узла, включая свойства, которые могут присутствовать не для всех узлов.

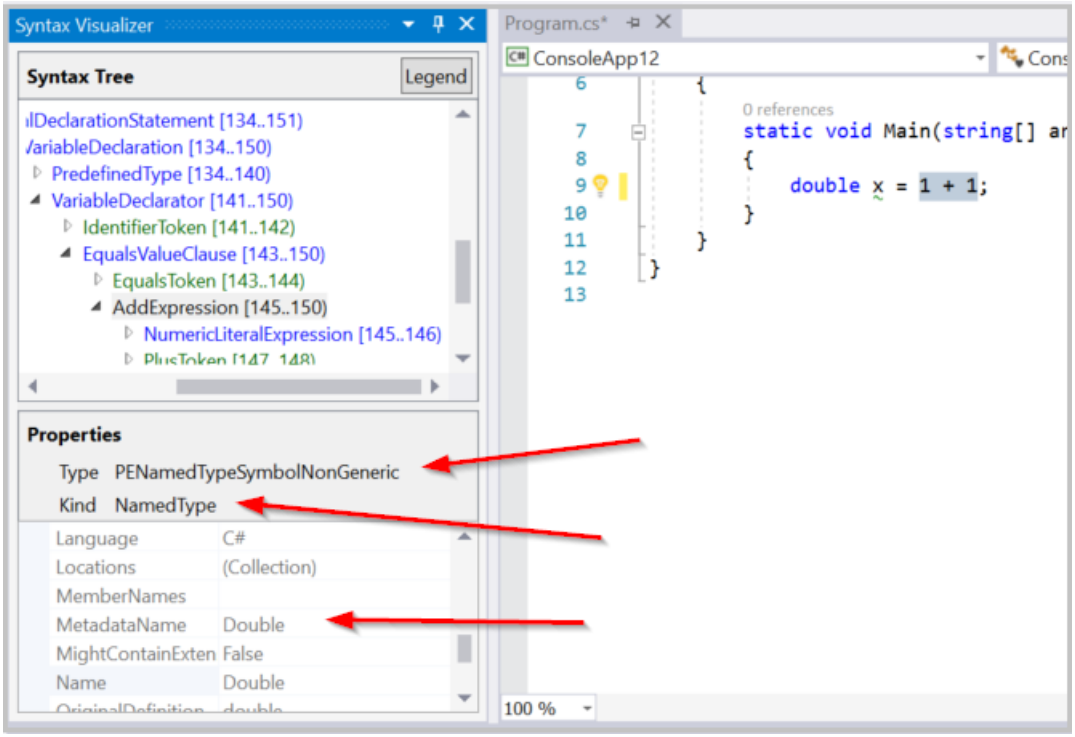
Сетка свойств в визуализаторе обновляется, как показано на рисунке ниже. Символом для выражения является **SynthesizedIntrinsicOperatorSymbol**, у которого **Вид** = **Метод**.



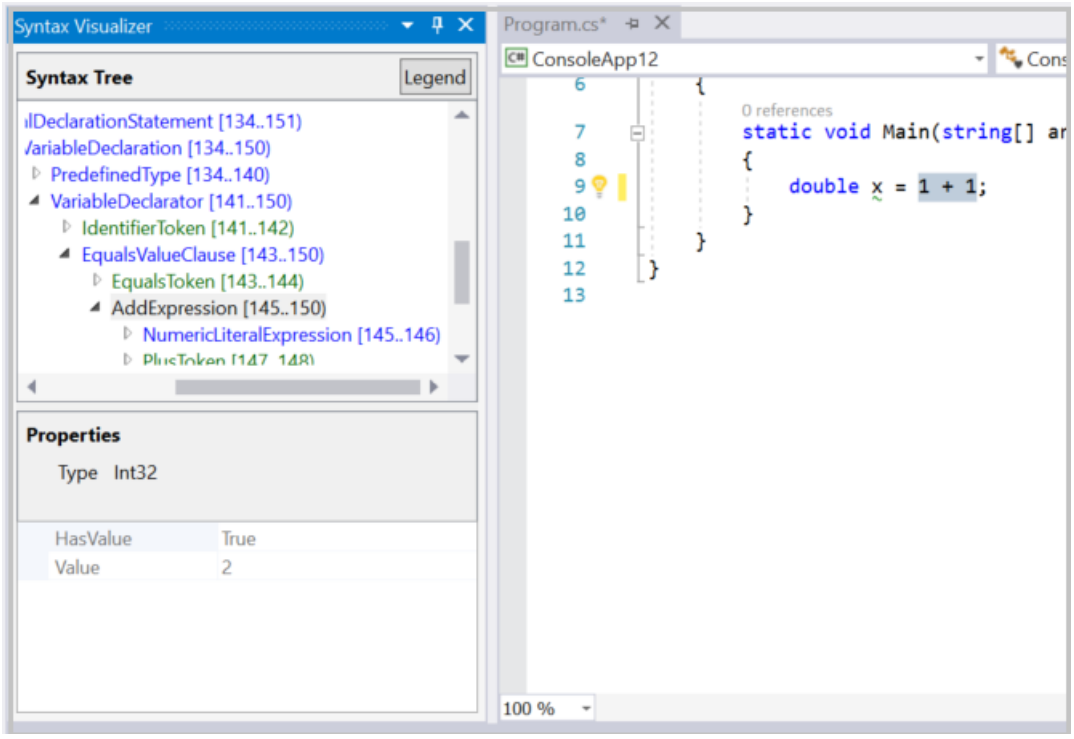
Попробуйте выбрать команду **View TypeSymbol (if any)** (Просмотреть символ типа (при наличии)) для того же узла **AddExpression**. Сетка свойств в визуализаторе обновляется, как показано на рисунке ниже, указывая, что выбранное выражение имеет тип `Int32`.



Попробуйте выбрать команду **View Converted TypeSymbol (if any)** (Просмотреть преобразованный символ типа (при наличии)) для того же узла **AddExpression**. Сетка свойств обновляется, указывая, что, хотя выражение имеет тип `Int32`, преобразованным типом выражения является `Double`, как показано на рисунке ниже. Этот узел включает сведения о преобразованном символе типа, так как выражение `Int32` находится в контексте, где его необходимо преобразовать в `Double`. Это преобразование удовлетворяет типу `Double`, указанному для переменной `x` в левой части оператора присваивания.



Наконец, попробуйте выбрать команду **View Constant Value (if any)** (Просмотреть постоянное значение (при наличии)) для того же узла **AddExpression**. Сетка свойств показывает, что значением выражения является константа времени компиляции со значением `2`.

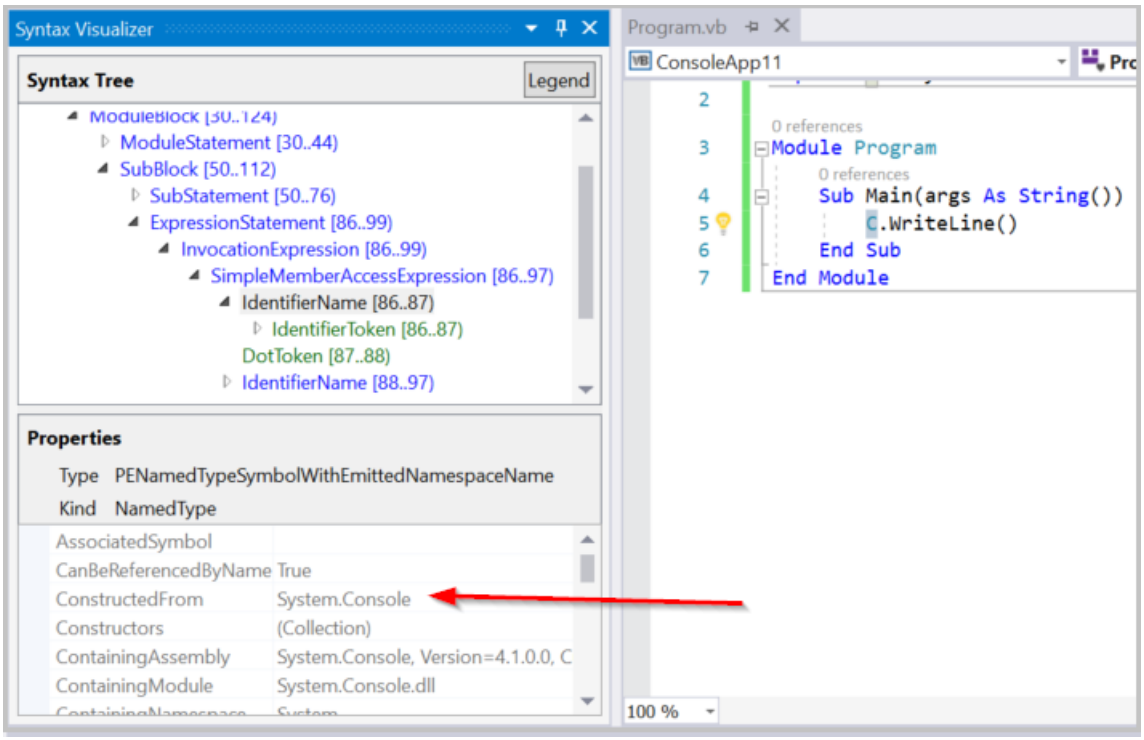


Предыдущий пример также можно воспроизвести в Visual Basic. Введите `Dim x As Double = 1 + 1` в файле Visual Basic. Выберите выражение `1 + 1` в окне редактора кода. Визуализатор выделяет соответствующий узел **AddExpression**. Повторите предыдущие шаги для данного узла **AddExpression**, полученные результаты должны быть теми же самыми.

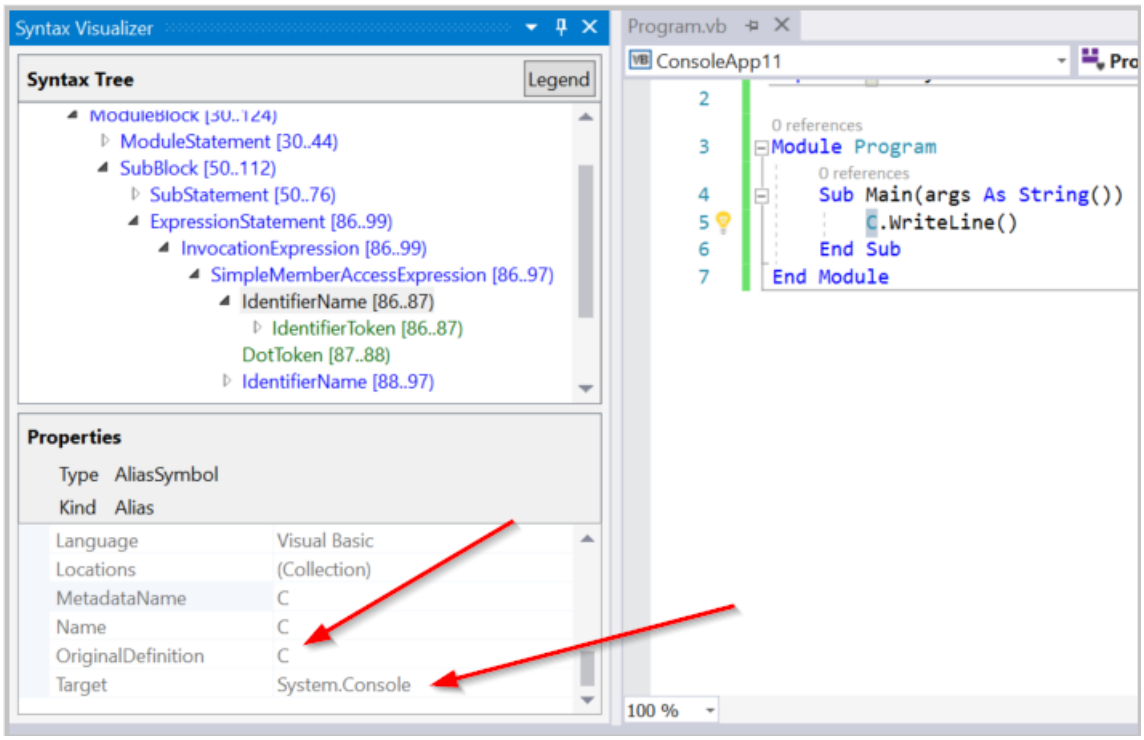
Изучите дополнительный код в Visual Basic. Измените основной файл Visual Basic, используя следующий код:

```
Imports C = System.Console

Module Program
    Sub Main(args As String())
        C.WriteLine()
    End Sub
End Module
```

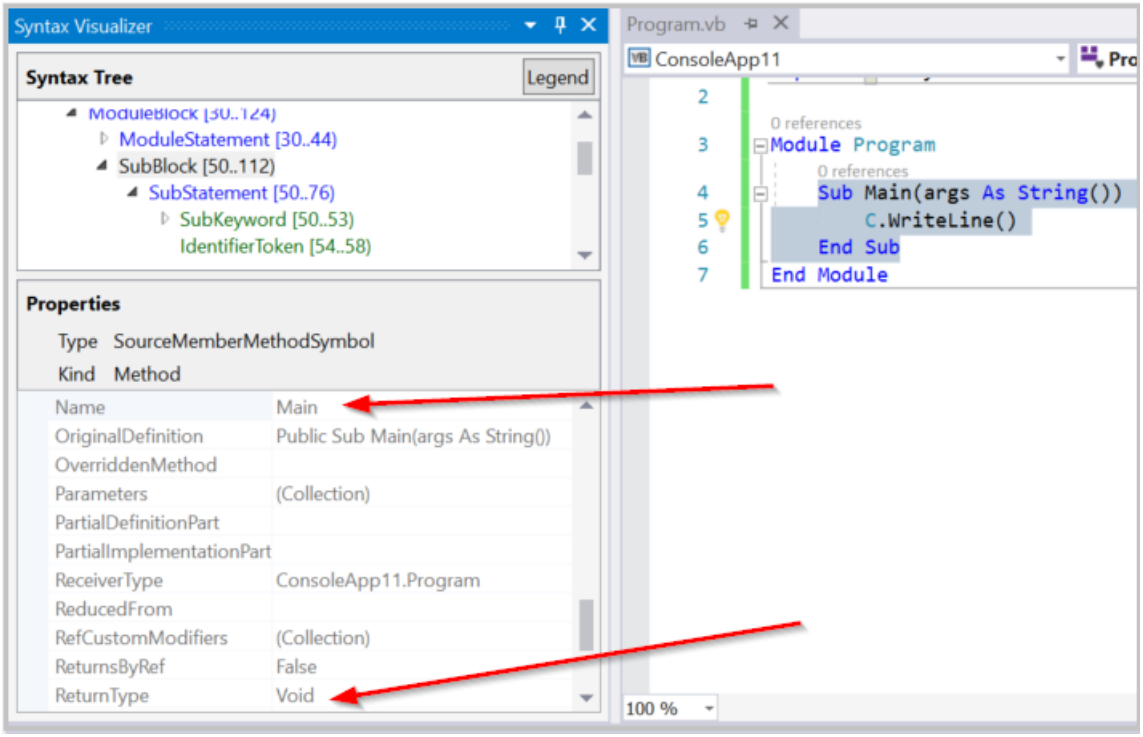
Этот код представляет псевдоним `C`, сопоставляемый с типом `System.Console` в начале файла, и использует его внутри `Main()`. Выберите использование этого псевдонима — `C` в `C.WriteLine()` — внутри метода `Main()`. Визуализатор выбирает соответствующий узел **IdentifierName**. Щелкните этот узел правой кнопкой мыши и выберите команду **View Symbol (if any)** (Просмотреть символ, если есть). Сетка свойств указывает, что этот идентификатор привязан к типу `System.Console`, как показано на следующем рисунке:



Попробуйте выбрать команду **View AliasSymbol (if any)** (Просмотреть символ псевдонима (при наличии)) для того же узла **IdentifierName**. Сетка свойств указывает, что этот идентификатор является псевдонимом с именем `C`, который привязан к целевому объекту `System.Console`. Другими словами, сетка свойств предоставляет сведения о **AliasSymbol**, соответствующем идентификатору `C`.



Изучите символ, соответствующей любому объявленному типу, методу, свойству. Выберите соответствующий узел в визуализаторе и щелкните элемент **View Symbol (if any)** (Просмотреть символ (при наличии)). Выберите метод `Sub Main()`, включая его тело. Щелкните **View Symbol (if any)** (Просмотреть символ (при наличии)) для соответствующего узла **SubBlock** в визуализаторе. Сетка свойств показывает, что **MethodSymbol** для этого **SubBlock** имеет имя `Main` с типом возвращаемого значения `Void`.



Приведенные выше примеры Visual Basic можно легко воспроизвести в C#. Введите `using C = System.Console;` вместо `Imports C = System.Console` для псевдонима. Выполнение описанных выше действий в C# приводит к аналогичным результатам в окне визуализатора.

Операции семантической проверки доступны только на узлах. Они недоступны для токенов или дополнительной информации. Не все узлы имеют значимые семантические сведения для изучения. Если узел не содержит значимой семантической информации, при выборе элемента **View * Symbol (if any)** (Просмотреть символ (при наличии) отображается пустая сетка свойств.

Подробнее об API для выполнения семантического анализа см. в обзорном документе Работа с семантикой.

Закрытие визуализатора синтаксиса

Вы можете закрыть окно визуализатора, когда оно не используется для изучения исходного кода. Визуализатор синтаксиса обновляет отображаемые данные по мере того, как вы перемещаетесь по исходному коду, редактируете и изменяете его. Когда вы не используете визуализатор, он может отвлекать вас от работы.

Генераторы исходного кода

18.12.2021 • 4 minutes to read

Эта статья содержит обзор генераторов исходного кода, входящих в состав пакета SDK для .NET Compiler Platform ("Roslyn"). Генераторы исходного кода — это функция компилятора C#, позволяющая разработчикам C# проверять код пользователя по мере его компиляции и создавать на лету новые исходные файлы C#, которые добавляются в пользовательскую компиляцию.

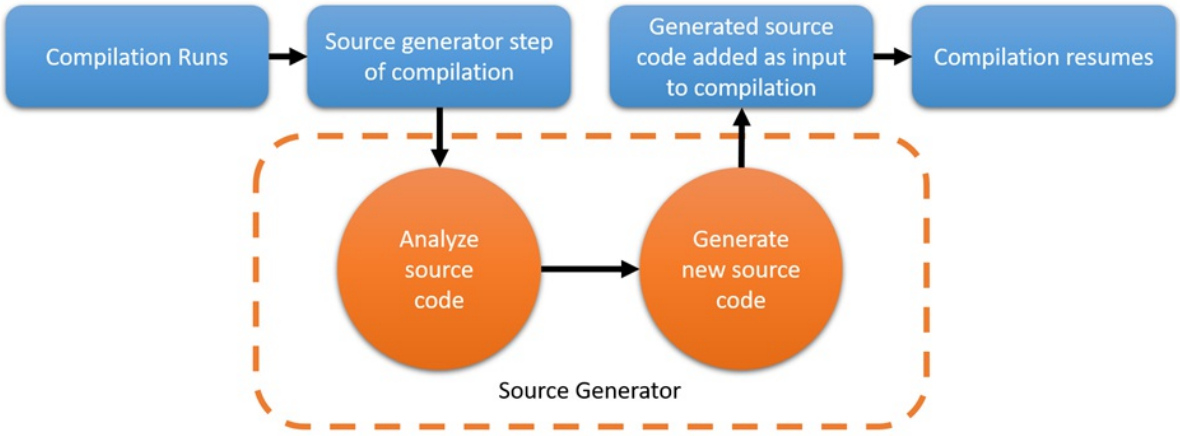
Генератор исходного кода — это фрагмент кода, который выполняется во время компиляции и может проверять программу для создания дополнительных исходных файлов, компилируемых вместе с остальной частью кода.

Генератор исходного кода — это новый тип компонента, который разработчики C# могут использовать, чтобы выполнять два основных действия:

1. Извлеките *объект компиляции*, представляющий весь компилируемый пользовательский код. Этот объект можно проверить, а вы можете написать код, который работает с синтаксисом и семантическими моделями компилируемого кода, как и с анализаторами, уже сегодня.
2. Создать исходные файлы C#, которые можно добавить в объект компиляции во время компиляции. Иными словами, во время компиляции кода можно указать дополнительный исходный код в качестве входных данных для компиляции.

В сочетании эти две вещи делают также полезным генераторы исходного кода. Вы можете изучить пользовательский код со всеми метаданными, которые компилятор создает во время компиляции, а затем выдать код C# обратно в ту же компиляцию, основанную на анализируемых данных! Если вы знакомы с анализаторами Roslyn, вы можете рассматривать исходные генераторы как анализаторы, которые могут выдавать исходный код C#.

Генераторы исходного кода выполняются в фазе компиляции, представленной ниже.



Генератор исходного кода — это сборка .NET Standard 2.0, которая загружается компилятором вместе с любыми анализаторами. Его можно использовать в средах, где можно загружать и запускать компоненты .NET Standard.

Распространенные сценарии

Сегодня есть три общих подхода к проверке пользовательского кода и созданию информации или кода на основе анализа, используемого современными технологиями: отражение среды выполнения, плетение

с помощью IL и жонглирование задачами MSBuild. Генераторы исходного кода могут быть усовершенствованы при любом подходе. Отражение среды выполнения — это мощная технология, которая была добавлена в .NET довольно давно. Существует множество сценариев для ее использования. Очень распространенный сценарий заключается в выполнении некоторого анализа пользовательского кода при запуске приложения и использовании этих данных для создания объектов.

Например, ASP.NET Core использует отражение при первом запуске веб-службы для обнаружения определенных конструкций, чтобы она могла "подсоединить" такие вещи, как контроллеры и страницы Razor. Хотя этот подход позволяет написать простой код с эффективными абстракциями, он может приводить к снижению производительности во время выполнения: когда веб-служба или приложение впервые запускается, оно не принимает никаких запросов до тех пор, пока не завершится весь код отражения среды выполнения, обнаруживающий сведения о коде. Несмотря на то, что это снижение производительности не является критичным, оно неизбежно присутствует, и его нельзя улучшить в приложении.

При использовании генератора исходного кода этап обнаружения контроллера может быть создан во время компиляции путем анализа исходного кода и создания кода, необходимого для "подключения" приложения. Это может сократить время запуска, поскольку действие, происходящее сейчас во время выполнения, может быть отправлено во время компиляции. Генераторы исходного кода могут повысить производительность так, чтобы не ограничиваться отражением во время выполнения для обнаружения типов. Некоторые сценарии предусматривают многократный вызов задачи C# MSBuild (называемой CSC), чтобы они могли проверять данные из компиляции. Как можно себе представить, многократный вызов компилятора влияет на общее время, затрачиваемое на создание приложения! Мы изучаем, как можно использовать генераторы исходного кода, чтобы избежать задач, таких как эта, так как генераторы исходного кода не просто предлагают некоторые преимущества MSBuild для повышения производительности, но также позволяют инструментам работать на правильном уровне абстракции.

Другие возможности генераторов исходного кода могут состоять в исключении некоторых "строго типизированных" API, например, как при маршрутизации ASP.NET Core между контроллерами и страницами Razor. При использовании генератора исходного кода маршрутизация может быть строго типизирована с помощью необходимых строк, формируемых как данные времени компиляции. Это уменьшит количество раз, когда неверно типизированный строковый литерал приводит к тому, что запрос не достигает нужного контроллера.

Знакомство с генераторами исходного кода

В этом разделе вы ознакомитесь с созданием генератора исходного кода с помощью API ISourceGenerator.

1. Создайте консольного приложения .NET.
2. Замените класс "Program" на следующий код.

```
partial class Program
{
    static void Main(string[] args)
    {
        HelloFrom("Generated Code");
    }

    static partial void HelloFrom(string name);
}
```

> **NOTE**
>
> Вы можете запустить этот пример кода, не изменяя, однако сейчас ничего не изменится.

3. Далее мы создадим генератор исходного кода, который будет заполнять содержимое метода `HelloFrom`.

4. Создайте стандартный проект библиотеки .NET, который выглядит следующим образом:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.CodeAnalysis.CSharp" Version="3.9.0" PrivateAssets="all" />
    <PackageReference Include="Microsoft.CodeAnalysis.Analyzers" Version="3.0.0" PrivateAssets="all"
/>
  </ItemGroup>

</Project>
```

5. Измените или создайте файл на C#, который определит ваш генератор исходного кода, например так:

```
using Microsoft.CodeAnalysis;

namespace MyGenerator
{
    [Generator]
    public class MySourceGenerator : ISourceGenerator
    {
        public void Execute(GeneratorExecutionContext context)
        {
            // Code generation goes here
        }

        public void Initialize(GeneratorInitializationContext context)
        {
            // No initialization required for this one
        }
    }
}
```

6. Замените содержимое метода `Execute`, чтобы оно выглядело следующим образом:

```
public void Execute(GeneratorExecutionContext context)
{
    // find the main method
    var mainMethod = context.Compilation.GetEntryPoint(context.CancellationToken);

    // build up the source code
    string source = $@"
using System;

namespace {mainMethod.ContainingNamespace.ToDisplayString()}
{{
    public static partial class {mainMethod.ContainingType.Name}
    {{
        static partial void HelloFrom(string name)
        {{
            Console.WriteLine($""Generator says: Hi from '{{name}}'"");
        }}
    }}
}}
";
    // add the source code to the compilation
    context.AddSource("generatedSource", source);
}
```

7. Теперь у нас есть функционирующий генератор, однако его еще нужно подключить к консольному приложению. Измените исходный проект консольного приложения и добавьте следующие данные, заменив путь к проекту на путь к созданному ранее проекту .NET Standard:

```
  <!-- Add this as a new ItemGroup, replacing paths and names appropriately -->
  <ItemGroup>
    <!-- Note that this is not a "normal" ProjectReference.
         It needs the additional 'OutputItemType' and 'ReferenceOutputAssembly' attributes. -->
    <ProjectReference Include="path-to-sourcegenerator-project.csproj"
                      OutputItemType="Analyzer"
                      ReferenceOutputAssembly="false" />
  </ItemGroup>
```

8. Теперь при запуске консольного приложения созданный код должен выполнятся с выводом соответствующего процесса на экран.

> **NOTE**
>
> Теперь вам нужно перезапустить Visual Studio, чтобы увидеть IntelliSense и избавиться от ошибок с помощью предыдущего интерфейса инструментов.

Дальнейшие действия

Сборник рецептов для генераторов исходного кода содержит примеры и некоторые рекомендуемые подходов по их решению. Кроме того, у нас есть набор примеров на портале GitHub, на котором вы можете попрактиковаться.

Дополнительные сведения о генераторах исходного кода см. в следующих разделах:

- Документ по разработке генераторов исходного кода
- Сборник рецептов для генераторов исходного кода

Начало работы с функциями синтаксического анализа

18.12.2021 • 11 minutes to read

В этом учебнике рассматривается **синтаксический API**. Синтаксический API предоставляет доступ к структурам данных, описывающих программы на языке C# или Visual Basic. Эти структуры данных достаточно подробны, чтобы обеспечивать полное представление любой программы любого размера. Эти структуры могут описывать готовые программы, обеспечивающие корректное выполнение и работу. Они также могут описывать неполные программы по мере их написания в редакторе.

Для обеспечения такого богатого выражения структуры данных и API, составляющие синтаксический API, имеют высокую сложность. Начнем с описания структуры данных для типичной программы "Hello, World!".

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Посмотрите на текст предыдущей программы. Вы узнаете знакомые элементы. Весь текст представляет один исходный файл, или **блок компиляции**. Первые три строки этого исходного файла являются **директивами** using. Остальной исходный код содержится в **объявлении пространства имен**. Объявление пространства имен содержит дочернее **объявление класса**. Объявление класса содержит одно **объявление метода**.

Синтаксический API создает древовидную структуру, корень которой представляет блок компиляции. Узлы в дереве представляют директивы using, объявление пространства имен и все остальные элементы программы. Структура дерева продолжается до самых нижних уровней. Строка "Hello World!" представляет собой **токен строкового литерала**, который является потомком **аргумента**. Синтаксический API предоставляет доступ к структуре программы. Можно создавать запросы к определенным практикам написания кода, выполнять обход всего дерева для понимания кода и создавать новые деревья путем изменения существующего дерева.

Это краткое описание представляет собой обзор видов сведений, к которым можно получить доступ с помощью синтаксического API. Синтаксический API является не более чем формальным API, который описывает уже знакомые конструкции кода из C#. Все возможности включают в себя сведения о форматировании кода, в том числе об использовании разрывов строк, пробелов и отступов. С помощью этих сведений можно полностью представить код как написанный и прочитанный программистами или компилятором. Эта структура позволяет взаимодействовать с исходным кодом на глубоко осмысленном уровне. Это уже не просто строки текста, а данные, представляющие структуру программы на языке C#.

Чтобы приступить к работе, потребуется установить **пакет SDK для .NET Compiler Platform**:

Инструкции по установке — Visual Studio Installer

Найти SDK-**пакет** .NET Compiler Platform в Visual Studio Installer можно двумя способами:

Установка с помощью Visual Studio Installer — представление "Рабочие нагрузки"

SDK-пакет .NET Compiler Platform не выбирается автоматически в рамках рабочей нагрузки разработки расширений Visual Studio. Его необходимо выбрать как дополнительный компонент.

1. Запустите Visual Studio Installer.
2. Выберите **Изменить**.
3. Отметьте рабочую нагрузку **Разработка расширений** Visual Studio.
4. Откройте узел **Разработка расширений** Visual Studio в дереве сводки.
5. Установите флажок SDK-**пакет** .NET Compiler Platform. Нужный пакет будет представлен последним в списке дополнительных компонентов.

Кроме того, вы можете настроить **редактор** DGML для отображения диаграмм в средстве визуализации:

1. Откройте узел **Отдельные компоненты** в дереве сводки.
2. Установите флажок **Редактор** DGML

Установка с помощью Visual Studio Installer — вкладка "Отдельные компоненты"

1. Запустите Visual Studio Installer.
2. Выберите **Изменить**.
3. Откройте вкладку **Отдельные компоненты**.
4. Установите флажок SDK-**пакет** .NET Compiler Platform. Нужный пакет будет представлен в разделе **Компиляторы, средства сборки и среды выполнения** в самом начале.

Кроме того, вы можете настроить **редактор** DGML для отображения диаграмм в средстве визуализации:

1. Установите флажок **Редактор** DGML. Нужный пакет будет представлен в разделе **Средства для работы с кодом**.

Основные сведения о синтаксисе деревьев

Синтаксический API используется для любого анализа структуры кода C#. **Синтаксический** API предоставляет средства синтаксического анализа, деревья синтаксиса и служебные программы для анализа и создания деревьев синтаксиса. С его помощью выполняется поиск определенных элементов синтаксиса в коде или чтение кода программы.

Дерево синтаксиса — это структура данных, которая используется компиляторами C# и Visual Basic для понимания программ на языках C# и Visual Basic. Деревья синтаксиса создает тот же синтаксический анализатор, что выполняется при построении проекта или при нажатии на клавишу F5. Деревья синтаксиса полностью соответствуют языку. Каждый бит сведений в файле кода представлен в дереве. При записи дерева синтаксиса в текст воспроизводится в точности тот исходный текст, который был проанализирован. Деревья синтаксиса являются **неизменяемыми**. После создания дерево синтаксиса невозможно изменить. Пользователи могут анализировать деревья в нескольких потоках без блокировки и принятия других меры по обеспечению параллелизма, поскольку данные не изменяются. Можно использовать API для создания новых деревьев посредством изменения существующего дерева.

Ниже перечислены четыре основных стандартных блока деревьев синтаксиса.

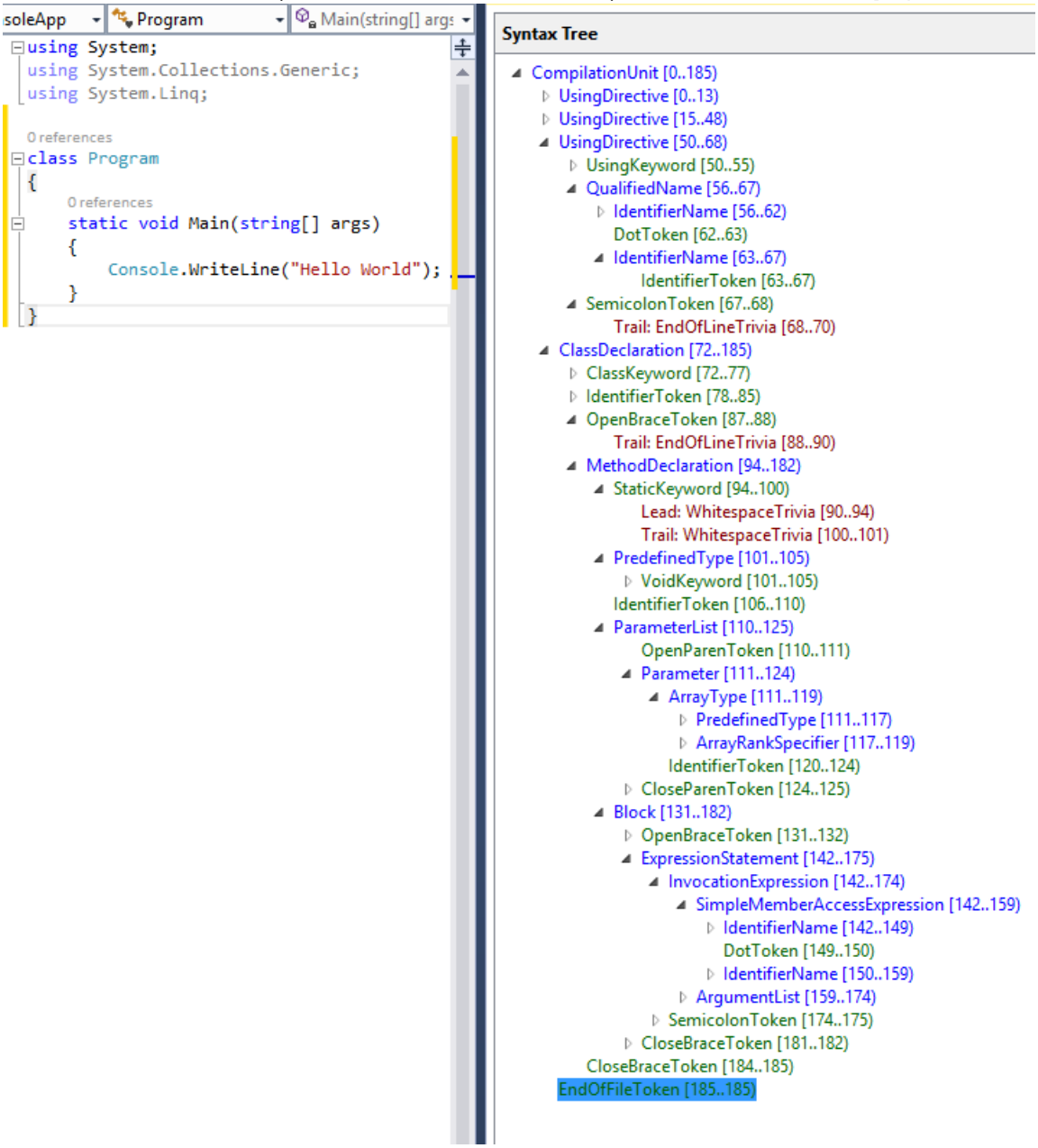
- Класс Microsoft.CodeAnalysis.SyntaxTree, экземпляр которого представляет дерево синтаксического анализа целиком. SyntaxTree является абстрактным классом с производными, соответствующими

конкретному языку. Для синтаксического анализа текста на языке C# (или Visual Basic) используются методы синтаксического анализа класса Microsoft.CodeAnalysis.CSharp.CSharpSyntaxTree (или Microsoft.CodeAnalysis.VisualBasic.VisualBasicSyntaxTree).

- Класс Microsoft.CodeAnalysis.SyntaxNode, экземпляры которого представляют такие синтаксические конструкции, как объявления, инструкции, предложения и выражения.
- Структура Microsoft.CodeAnalysis.SyntaxToken, которая представляет отдельные ключевые слова, идентификаторы, операторы или знаки препинания.
- И, наконец, структура Microsoft.CodeAnalysis.SyntaxTrivia, которая представляет малозначимые с точки зрения синтаксиса элементы сведений, такие как пробелы между токенами, директивы предварительной обработки и комментарии.

Заметки, токены и узлы иерархически компонуются для формирования дерева, которое полностью представляет все содержимое в фрагменте кода Visual Basic или C#. Эту структуру можно посмотреть с помощью окна **визуализатора синтаксиса**. В Visual Studio выберите **Вид** > **Другие окна** > **Визуализатор синтаксиса**. Например, предыдущий исходный файл на языке C# при просмотре с помощью **визуализатора синтаксиса** выглядит, как показано на следующем рисунке.

Узел синтаксиса: синий | **Токен синтаксиса**: зеленый | **Заметки к синтаксису**: красный



При перемещении по структуре дерева вы можете найти в файле кода все инструкции, выражения, токены и пробелы.

С помощью синтаксического API можно найти все, что угодно, в файле кода, но большинство сценариев предусматривает исследование небольших фрагментов кода либо поиск определенных инструкций или фрагментов. Ниже приводятся два примера, иллюстрирующие типичные сценарии использования для просмотра структуры кода или поиска отдельных инструкций.

Обход деревьев

Узлы дерева синтаксиса можно исследовать двумя способами. Можно выполнить обход дерева для исследования каждого узла или выполнить запрос к определенным элементам или узлам.

Обход вручную

Окончательный код этого примера доступен в репозитории на сайте GitHub.

> **NOTE**
> Типы синтаксического дерева используют наследование для описания различных элементов синтаксиса, которые являются допустимыми в разных местах в программе. Применение этих API часто означает приведение свойств или элементов коллекции к конкретным производным типам. В следующих примерах назначения и приведения являются отдельными инструкциями, использующими явно типизированные переменные. Прочитайте код, чтобы увидеть типы возвращаемых значений API и тип среды выполнения возвращаемых объектов. На практике более распространено использование неявно типизированных переменных и имен API для описания типа рассматриваемых объектов.

Создайте новый проект C# для **автономного средства анализа кода**:

- В Visual Studio последовательно выберите **Файл** > **Создать** > **Проект**, чтобы открыть диалоговое окно "Новый проект".
- В разделе **Visual C#** > **Расширяемость** выберите **Автономное средство анализа кода**.
- Присвойте проекту имя "**SyntaxTreeManualTraversal**" и нажмите кнопку ОК.

Вы будете анализировать программу "Hello World!", показанную ранее. Добавьте текст для программы "Hello World" в качестве константы в класс `Program`:

```
        const string programText =
@"using System;
using System.Collections;
using System.Linq;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(""Hello, World!"");
        }
    }
}";
```

Затем добавьте в константу `programText` следующий код для создания **дерева синтаксиса** для текста кода. Добавьте следующую строку в метод `Main`:

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);
CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

Эти две строки создают дерево и извлекают корневой узел этого дерева. Теперь можно исследовать узлы в дереве. Добавьте следующие строки в метод `Main` для отображения некоторых свойств корневого узла дерева:

```
WriteLine($"The tree is a {root.Kind()} node.");
WriteLine($"The tree has {root.Members.Count} elements in it.");
WriteLine($"The tree has {root.Usings.Count} using statements. They are:");
foreach (UsingDirectiveSyntax element in root.Usings)
    WriteLine($"\t{element.Name}");
```

Запустите приложение для просмотра сведений, которые код собрал о корневом узле этого дерева.

Как правило, для получения сведений о коде выполняется обход дерева. В этом примере анализируется известный вам код для изучения API. Добавьте следующий код для исследования первого элемента узла `root`:

```
MemberDeclarationSyntax firstMember = root.Members[0];
WriteLine($"The first member is a {firstMember.Kind()}.");
var helloWorldDeclaration = (NamespaceDeclarationSyntax)firstMember;
```

Этот элемент — Microsoft.CodeAnalysis.CSharp.Syntax.NamespaceDeclarationSyntax. Он представляет все, что входит в объявление `namespace HelloWorld`. Добавьте следующий код, чтобы посмотреть, какие узлы объявляются внутри пространства имен `HelloWorld`:

```
WriteLine($"There are {helloWorldDeclaration.Members.Count} members declared in this namespace.");
WriteLine($"The first member is a {helloWorldDeclaration.Members[0].Kind()}.");
```

Запустите программу, чтобы посмотреть, что вы узнали.

Теперь, когда известно, что объявление является Microsoft.CodeAnalysis.CSharp.Syntax.ClassDeclarationSyntax, объявите новую переменную этого типа для исследования объявления класса. Этот класс содержит только один элемент: метод `Main`. Добавьте следующий код, чтобы найти метод `Main`, и приведите его к Microsoft.CodeAnalysis.CSharp.Syntax.MethodDeclarationSyntax.

```
var programDeclaration = (ClassDeclarationSyntax)helloWorldDeclaration.Members[0];
WriteLine($"There are {programDeclaration.Members.Count} members declared in the
{programDeclaration.Identifier} class.");
WriteLine($"The first member is a {programDeclaration.Members[0].Kind()}.");
var mainDeclaration = (MethodDeclarationSyntax)programDeclaration.Members[0];
```

Узел объявления метода содержит полные синтаксические сведения о методе. Давайте посмотрим тип возвращаемого значения для метода `Main`, количество и типы аргументов, а также основной текст метода. Добавьте следующий код:

```
WriteLine($"The return type of the {mainDeclaration.Identifier} method is {mainDeclaration.ReturnType}.");
WriteLine($"The method has {mainDeclaration.ParameterList.Parameters.Count} parameters.");
foreach (ParameterSyntax item in mainDeclaration.ParameterList.Parameters)
    WriteLine($"The type of the {item.Identifier} parameter is {item.Type}.");
WriteLine($"The body text of the {mainDeclaration.Identifier} method follows:");
WriteLine(mainDeclaration.Body.ToFullString());

var argsParameter = mainDeclaration.ParameterList.Parameters[0];
```

Запустите программу, чтобы просмотреть все полученные сведения об этой программе:

```
The tree is a CompilationUnit node.
The tree has 1 elements in it.
The tree has 4 using statements. They are:
        System
        System.Collections
        System.Linq
        System.Text
The first member is a NamespaceDeclaration.
There are 1 members declared in this namespace.
The first member is a ClassDeclaration.
There are 1 members declared in the Program class.
The first member is a MethodDeclaration.
The return type of the Main method is void.
The method has 1 parameters.
The type of the args parameter is string[].
The body text of the Main method follows:
        {
            Console.WriteLine("Hello, World!");
        }
```

Методы запросов

Помимо обхода деревьев можно также исследовать деревья синтаксиса с помощью методов запросов, определенных в Microsoft.CodeAnalysis.SyntaxNode. Эти методы известны всем, кто знаком с XPath. Эти методы можно использовать совместно с LINQ для быстрого поиска в дереве. SyntaxNode содержит такие методы запросов, как DescendantNodes, AncestorsAndSelf и ChildNodes.

Эти методы запроса можно использовать для поиска аргументов к методу `Main` в качестве альтернативы навигации по дереву. Добавьте следующий код в конец метода `Main`:

```
var firstParameters = from methodDeclaration in root.DescendantNodes()
                                        .OfType<MethodDeclarationSyntax>()
                      where methodDeclaration.Identifier.ValueText == "Main"
                      select methodDeclaration.ParameterList.Parameters.First();

var argsParameter2 = firstParameters.Single();

WriteLine(argsParameter == argsParameter2);
```

В первой инструкции используется выражение LINQ и метод DescendantNodes для обнаружения того же параметра, что и в предыдущем примере.

Запустите программу, и выражение LINQ обнаружит тот же параметр, что и при навигации по дереву вручную.

В этом образце используются инструкции `WriteLine` для отображения сведений о деревьях синтаксиса по мере их обхода. Гораздо больше сведений можно получить, запустив готовую программу в отладчике. Можно просмотреть другие свойства и методы в составе дерева синтаксиса, созданного для программы "hello world".

Средства обхода синтаксиса

Часто бывает необходимо найти все узлы определенного типа в дереве синтаксиса, например каждое объявление свойств в файле. Путем расширения класса Microsoft.CodeAnalysis.CSharp.CSharpSyntaxWalker и переопределения метода VisitPropertyDeclaration(PropertyDeclarationSyntax) можно обработать каждое объявление свойств в дереве синтаксиса, не зная заранее его структуры. CSharpSyntaxWalker — это особый тип CSharpSyntaxVisitor, который рекурсивно обходит узел и все его дочерние элементы.

Этот пример реализует CSharpSyntaxWalker для исследования дерева синтаксиса. Он собирает

обнаруженные директивы `using`, которые не импортируют пространство имен `System`.

Создайте новый проект C# для **автономного средства анализа кода**. Присвойте ему имя "**SyntaxWalker**".

Окончательный код этого примера доступен в репозитории на сайте GitHub. Пример на GitHub содержит оба проекта, описанные в этом учебнике.

Как показано в предыдущем примере, можно определить строковую константу, которая будет содержать текст программы для анализа:

```
        const string programText =
@"using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;

namespace TopLevel
{
    using Microsoft;
    using System.ComponentModel;

    namespace Child1
    {
        using Microsoft.Win32;
        using System.Runtime.InteropServices;

        class Foo { }
    }

    namespace Child2
    {
        using System.CodeDom;
        using Microsoft.CSharp;

        class Bar { }
    }
}";
```

Этот исходный текст содержит директивы `using`, расположенные в четырех разных местах: на уровне файлов, в пространстве имен верхнего уровня и в двух вложенных пространствах имен. В этом примере представлен основной сценарий использования класса CSharpSyntaxWalker для выполнения запросов к коду. Было бы сложно посетить каждый узел в корневом дереве синтаксиса для поиска объявлений using. Вместо этого создается производный класс и переопределяется метод, который вызывается только в том случае, когда текущий узел в дереве является директивой using. Обходчик не работает с узлами других типов. Этот единичный метод проверяет каждую из инструкций `using` и создает коллекцию пространств имен, которые не входят в пространство имен `System`. Создается CSharpSyntaxWalker для проверки всех инструкций `using`, но только инструкций `using`.

Теперь, когда вы определили текст программы, необходимо создать `SyntaxTree` и получить корень этого дерева:

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);
CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

Теперь создайте новый класс. В Visual Studio выберите **Проект** > **Добавить новый элемент**. В диалоге **Добавление нового элемента** введите имя файла *UsingCollector.cs*.

Вы реализуете функцию обходчика `using` в классе `UsingCollector`. Для начала сделайте класс `UsingCollector` производным от CSharpSyntaxWalker.

```
class UsingCollector : CSharpSyntaxWalker
```

Потребуется хранилище для хранения узлов пространства имен, которые вы будете собирать. Объявите свойство только для чтения в классе `UsingCollector`. Эта переменная будет использоваться для хранения найденных узлов UsingDirectiveSyntax:

```
public ICollection<UsingDirectiveSyntax> Usings { get; } = new List<UsingDirectiveSyntax>();
```

Базовый класс, CSharpSyntaxWalker, реализует логику для посещения каждого узла в дереве синтаксиса. Производный класс переопределяет методы, вызываемые для определенных интересующих вас узлов. В этом примере вас интересует любая директива `using`. Это означает, что требуется переопределить метод VisitUsingDirective(UsingDirectiveSyntax). Единственный аргумент этого метода — объект Microsoft.CodeAnalysis.CSharp.Syntax.UsingDirectiveSyntax. Это важное преимущество обходчиков: они вызывают переопределенные методы с использованием аргументов, уже приведенных к типу конкретного узла. Класс Microsoft.CodeAnalysis.CSharp.Syntax.UsingDirectiveSyntax имеет свойство Name, которое содержит имя импортируемого пространства имен. Это Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax. Добавьте следующий код в переопределение VisitUsingDirective(UsingDirectiveSyntax):

```
public override void VisitUsingDirective(UsingDirectiveSyntax node)
{
    WriteLine($"\tVisitUsingDirective called with {node.Name}.");
    if (node.Name.ToString() != "System" &&
        !node.Name.ToString().StartsWith("System."))
    {
        WriteLine($"\t\tSuccess. Adding {node.Name}.");
        this.Usings.Add(node);
    }
}
```

Как и в прошлом примере, были добавлены разнообразные инструкции `WriteLine` для облегчения понимания этого метода. Вы можете посмотреть, когда он вызывается и какие при этом передаются аргументы.

Наконец, необходимо добавить две строки кода для создания `UsingCollector` и направить его в корневой узел для сбора всех инструкций `using`. Затем добавьте цикл `foreach` для отображения всех инструкций `using`, найденных сборщиком:

```
var collector = new UsingCollector();
collector.Visit(root);
foreach (var directive in collector.Usings)
{
    WriteLine(directive.Name);
}
```

Скомпилируйте и запустите программу. Вы должны увидеть следующий результат.

```
        VisitUsingDirective called with System.
        VisitUsingDirective called with System.Collections.Generic.
        VisitUsingDirective called with System.Linq.
        VisitUsingDirective called with System.Text.
        VisitUsingDirective called with Microsoft.CodeAnalysis.
                Success. Adding Microsoft.CodeAnalysis.
        VisitUsingDirective called with Microsoft.CodeAnalysis.CSharp.
                Success. Adding Microsoft.CodeAnalysis.CSharp.
        VisitUsingDirective called with Microsoft.
                Success. Adding Microsoft.
        VisitUsingDirective called with System.ComponentModel.
        VisitUsingDirective called with Microsoft.Win32.
                Success. Adding Microsoft.Win32.
        VisitUsingDirective called with System.Runtime.InteropServices.
        VisitUsingDirective called with System.CodeDom.
        VisitUsingDirective called with Microsoft.CSharp.
                Success. Adding Microsoft.CSharp.
Microsoft.CodeAnalysis
Microsoft.CodeAnalysis.CSharp
Microsoft
Microsoft.Win32
Microsoft.CSharp
Press any key to continue . . .
```

Поздравляем! Вы использовали **синтаксический** API для поиска определенных типов инструкций и объявлений C# в исходном коде C#.

Начало работы с семантическим анализом

18.12.2021 • 7 minutes to read

В учебнике предполагается, что вы знакомы с синтаксическим API. Вводные сведения можно найти в статье о начале работы с синтаксическим анализом.

В этом учебнике вы изучите **символы** и **API привязки**. Эти API предоставляют сведения о *семантическом значении* программы. Они позволяют задавать вопросы, касающиеся типов, представленных любыми символами в программе, и получать на них ответы.

Вам нужно установить **пакет** SDK **для** .NET Compiler Platform:

Инструкции по установке — Visual Studio Installer

Найти SDK-**пакет** .NET Compiler Platform в Visual Studio Installer можно двумя способами:

Установка с помощью Visual Studio Installer — представление "Рабочие нагрузки"

SDK-пакет .NET Compiler Platform не выбирается автоматически в рамках рабочей нагрузки разработки расширений Visual Studio. Его необходимо выбрать как дополнительный компонент.

1. Запустите Visual Studio Installer.
2. Выберите **Изменить**.
3. Отметьте рабочую нагрузку **Разработка расширений** Visual Studio.
4. Откройте узел **Разработка расширений** Visual Studio в дереве сводки.
5. Установите флажок SDK-**пакет** .NET Compiler Platform. Нужный пакет будет представлен последним в списке дополнительных компонентов.

Кроме того, вы можете настроить **редактор** DGML для отображения диаграмм в средстве визуализации:

1. Откройте узел **Отдельные компоненты** в дереве сводки.
2. Установите флажок **Редактор** DGML

Установка с помощью Visual Studio Installer — вкладка "Отдельные компоненты"

1. Запустите Visual Studio Installer.
2. Выберите **Изменить**.
3. Откройте вкладку **Отдельные компоненты**.
4. Установите флажок SDK-**пакет** .NET Compiler Platform. Нужный пакет будет представлен в разделе **Компиляторы, средства сборки и среды выполнения** в самом начале.

Кроме того, вы можете настроить **редактор** DGML для отображения диаграмм в средстве визуализации:

1. Установите флажок **Редактор** DGML. Нужный пакет будет представлен в разделе **Средства для работы с кодом**.

Общие сведения о компиляциях и символах

В процессе работы с пакетом SDK для компилятора .NET вы узнаете различия между синтаксическим API и семантическим API. **Синтаксический** API позволяет получить представление о *структуре* программы. Однако часто требуются более полные сведения о семантике или *значении* программы. Хотя

синтаксический анализ свободного файла либо фрагмента кода Visual Basic или C# можно выполнить изолированно, задавать вопросы, к примеру о типе переменной, в отрыве от реальности не имеет смысла. Значение имени типа может зависеть от ссылок на сборки, операций импорта пространств имен или других файлов кода. Ответы на эти вопросы можно получить с помощью **семантического** API, в частности класса Microsoft.CodeAnalysis.Compilation.

Экземпляр Compilation является аналогом отдельного проекта с точки зрения компилятора и представляет все необходимое для компиляции программы Visual Basic или C#. **Компиляция** включает в себя набор компилируемых исходных файлов, ссылки на сборки и параметры компилятора. О значении кода можно рассуждать, используя в этом контексте все остальные сведения. Compilation позволяет находить **символы** — сущности, такие как типы, пространства имен, члены и переменные, на которые указывают имена и другие выражения. Процесс связывания имен и выражений с **символами** называется **привязкой**.

Как Microsoft.CodeAnalysis.SyntaxTree, Compilation является абстрактным классом с производными, соответствующими конкретному языку. При создании экземпляра компиляции необходимо вызвать фабричный метод в классе Microsoft.CodeAnalysis.CSharp.CSharpCompilation (или Microsoft.CodeAnalysis.VisualBasic.VisualBasicCompilation).

Выполнение запросов к символам

В этом учебнике мы снова обратимся к программе "Hello, World!". На этот раз вы будете запрашивать символы в программе, чтобы понять, какие типы они представляют. Вы будете запрашивать типы в пространстве имен и узнаете, как найти методы, доступные для типа.

Окончательный код этого примера доступен в репозитории на сайте GitHub.

> **NOTE**
> Типы синтаксического дерева используют наследование для описания различных элементов синтаксиса, которые являются допустимыми в разных местах в программе. Применение этих API часто означает приведение свойств или элементов коллекции к конкретным производным типам. В следующих примерах назначения и приведения являются отдельными инструкциями, использующими явно типизированные переменные. Прочитайте код, чтобы увидеть типы возвращаемых значений API и тип среды выполнения возвращаемых объектов. На практике более распространено использование неявно типизированных переменных и имен API для описания типа рассматриваемых объектов.

Создайте новый проект C# для **автономного средства анализа кода**:

- В Visual Studio последовательно выберите **Файл** > **Создать** > **Проект**, чтобы открыть диалоговое окно "Новый проект".
- В разделе **Visual C#** > **Расширяемость** выберите **Автономное средство анализа кода**.
- Присвойте проекту имя "**SemanticQuickStart**" и нажмите кнопку "ОК".

Вы будете анализировать программу "Hello World!", показанную ранее. Добавьте текст для программы "Hello World" в качестве константы в класс `Program`:

```
        const string programText =
@"using System;
using System.Collections.Generic;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(""Hello, World!"");
        }
    }
}";
```

Затем добавьте в константу `programText` следующий код для создания дерева синтаксиса для текста кода. Добавьте следующую строку в метод `Main`:

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);

CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

Далее постройте CSharpCompilation из уже созданного дерева. В примере "Hello World!" используются типы String и Console. Необходимо сослаться на сборку, объявляющую эти два типа при компиляции. Добавьте следующую строку в метод `Main` для создания компиляции дерева синтаксиса, включая ссылку на соответствующую сборку:

```
var compilation = CSharpCompilation.Create("HelloWorld")
    .AddReferences(MetadataReference.CreateFromFile(
        typeof(string).Assembly.Location))
    .AddSyntaxTrees(tree);
```

Метод CSharpCompilation.AddReferences добавит ссылки на компиляцию. Метод MetadataReference.CreateFromFile загрузит сборку как ссылку.

Выполнение запросов к семантической модели

К существующему объекту Compilation можно отправить запрос на класс SemanticModel для любого класса SyntaxTree, содержащегося в этом объекте Compilation. Семантическую модель можно представить как источник всей информации, которая обычно доступна из IntelliSense. SemanticModel может ответить, какие имена находятся в области в том или ином расположении, какие элементы доступны в определенном методе, какие переменные используются в неком блоке текста, на что ссылается некое имя или выражение, а также на другие подобные вопросы. Добавьте этот оператор для создания семантической модели:

```
SemanticModel model = compilation.GetSemanticModel(tree);
```

Привязка имени

Compilation создает SemanticModel из SyntaxTree. После создания модели можно выполнить запрос на поиск первой директивы `using` и получение сведений о символах для пространства имен `System`. Добавьте следующие две строки в метод `Main` для создания семантической модели и получения символа для первого оператора using:

```
// Use the syntax tree to find "using System;"
UsingDirectiveSyntax usingSystem = root.Usings[0];
NameSyntax systemName = usingSystem.Name;

// Use the semantic model for symbol information:
SymbolInfo nameInfo = model.GetSymbolInfo(systemName);
```

В предыдущем коде показано, как привязать имя в первой директиве `using`, чтобы получить Microsoft.CodeAnalysis.SymbolInfo для пространства имен `System`. В предыдущем коде также демонстрируется, что для поиска структуры кода используется **синтаксическая модель**, а для понимания ее значения — **семантическая модель**. **Синтаксическая модель** находит строку `System` в операторе using. **Семантическая модель** располагает всеми сведениями о типах, определенных в пространстве имен `System`.

Из объекта SymbolInfo можно получить интерфейс Microsoft.CodeAnalysis.ISymbol с помощью свойства SymbolInfo.Symbol. Это свойство возвращает символ, на который ссылается это выражение. Для выражений, не ссылающихся ни на какие объекты (например, числовые литералы), это свойство имеет значение `null`. Если SymbolInfo.Symbol не задано значение NULL, ISymbol.Kind обозначает тип символа. В этом примере свойству ISymbol.Kind присваивается значение SymbolKind.Namespace. Добавьте приведенный ниже код в метод `Main`. Он возвращает символ для пространства имен `System`, а затем отображает все дочерние пространства имен, объявленные в пространстве имен `System`:

```
var systemSymbol = (INamespaceSymbol)nameInfo.Symbol;
foreach (INamespaceSymbol ns in systemSymbol.GetNamespaceMembers())
{
    Console.WriteLine(ns);
}
```

Запустите программу. Вы должны увидеть следующие результаты:

```
System.Collections
System.Configuration
System.Deployment
System.Diagnostics
System.Globalization
System.IO
System.Numerics
System.Reflection
System.Resources
System.Runtime
System.Security
System.StubHelpers
System.Text
System.Threading
Press any key to continue . . .
```

> **NOTE**
> Выходные данные не содержат каждое пространство имен, имеющее дочернее пространство имен от пространства имен `System`. В них отображается каждое пространство имен, которое присутствует в этой компиляции и ссылается только на сборку, где объявлено `System.String`. Этой компиляции неизвестно о пространствах имен, объявленных в других сборках.

Привязка выражения

В предыдущем коде показано, как найти символ путем привязки к имени. В программе C# существуют и

другие не являющиеся именами выражения, которые можно привязать. Чтобы продемонстрировать эту возможность, выполним привязку к простому строковому литералу.

Программа "Hello World" содержит класс Microsoft.CodeAnalysis.CSharp.Syntax.LiteralExpressionSyntax, строку "Hello World", отображаемую в консоли.

Строку "Hello World" можно найти путем поиска одного строкового литерала в программе. Затем после обнаружения узла синтаксиса получите сведения о типе для этого узла из семантической модели. Добавьте приведенный ниже код в метод `Main`:

```
// Use the syntax model to find the literal string:
LiteralExpressionSyntax helloWorldString = root.DescendantNodes()
    .OfType<LiteralExpressionSyntax>()
    .Single();

// Use the semantic model for type information:
TypeInfo literalInfo = model.GetTypeInfo(helloWorldString);
```

Структура Microsoft.CodeAnalysis.TypeInfo содержит свойство TypeInfo.Type, которое обеспечивает доступ к семантическим сведениям о типе литерала. В этом примере типом является `string`. Добавьте объявление, которое присваивает это свойство локальной переменной:

```
var stringTypeSymbol = (INamedTypeSymbol)literalInfo.Type;
```

Чтобы завершить действия в этом учебнике, сформируем запрос LINQ, который создает последовательность всех открытых методов, объявленных в типе `string` и возвращающих `string`. Поскольку запрос сложный, мы будем создавать его построчно, а затем перестроим в один запрос. Источником для этого запроса является последовательность всех элементов, объявленных в типе `string`:

```
var allMembers = stringTypeSymbol.GetMembers();
```

Эта исходная последовательность содержит все элементы, включая свойства и поля, поэтому ее нужно отфильтровать с помощью метода ImmutableArray<T>.OfType, чтобы найти элементы, которые являются объектами Microsoft.CodeAnalysis.IMethodSymbol:

```
var methods = allMembers.OfType<IMethodSymbol>();
```

Затем добавьте другой фильтр, чтобы вернуть только открытые методы, возвращающие `string`:

```
var publicStringReturningMethods = methods
    .Where(m => m.ReturnType.Equals(stringTypeSymbol) &&
    m.DeclaredAccessibility == Accessibility.Public);
```

Выберите только свойство имени и только уникальные имена, удаляя все перегрузки:

```
var distinctMethods = publicStringReturningMethods.Select(m => m.Name).Distinct();
```

Можно также создать полный запрос с помощью синтаксиса запросов LINQ и затем отобразить имена всех методов в консоли:

```
foreach (string name in (from method in stringTypeSymbol
                         .GetMembers().OfType<IMethodSymbol>()
                         where method.ReturnType.Equals(stringTypeSymbol) &&
                         method.DeclaredAccessibility == Accessibility.Public
                         select method.Name).Distinct())
{
    Console.WriteLine(name);
}
```

Выполните сборку и запуск программы. Вы должны увидеть следующий результат.

```
Join
Substring
Trim
TrimStart
TrimEnd
Normalize
PadLeft
PadRight
ToLower
ToLowerInvariant
ToUpper
ToUpperInvariant
ToString
Insert
Replace
Remove
Format
Copy
Concat
Intern
IsInterned
Press any key to continue . . .
```

Вы использовали семантический API для поиска и отображения сведений о символах, которые являются частью этой программы.

Начало работы с синтаксическими преобразованиями

18.12.2021 • 11 minutes to read

В основе этого руководства лежат концепции и методы, описанные в кратких руководствах Начало работы с функциями синтаксического анализа и Начало работы с семантическим анализом. Если вы еще не изучили их, сделайте это до начала работы с этим руководством.

В этом кратком руководстве вы изучите методы создания и преобразования деревьев синтаксиса. Объединив их с методами, рассмотренными в предыдущих кратких руководствах, вы сможете создать рефакторинг в командной строке!

Инструкции по установке — Visual Studio Installer

Найти SDK-**пакет** .NET Compiler Platform в Visual Studio Installer можно двумя способами:

Установка с помощью Visual Studio Installer — представление "Рабочие нагрузки"

SDK-пакет .NET Compiler Platform не выбирается автоматически в рамках рабочей нагрузки разработки расширений Visual Studio. Его необходимо выбрать как дополнительный компонент.

1. Запустите Visual Studio Installer.
2. Выберите **Изменить**.
3. Отметьте рабочую нагрузку **Разработка расширений** Visual Studio.
4. Откройте узел **Разработка расширений** Visual Studio в дереве сводки.
5. Установите флажок SDK-**пакет** .NET Compiler Platform. Нужный пакет будет представлен последним в списке дополнительных компонентов.

Кроме того, вы можете настроить **редактор** DGML для отображения диаграмм в средстве визуализации:

1. Откройте узел **Отдельные компоненты** в дереве сводки.
2. Установите флажок **Редактор** DGML

Установка с помощью Visual Studio Installer — вкладка "Отдельные компоненты"

1. Запустите Visual Studio Installer.
2. Выберите **Изменить**.
3. Откройте вкладку **Отдельные компоненты**.
4. Установите флажок SDK-**пакет** .NET Compiler Platform. Нужный пакет будет представлен в разделе **Компиляторы, средства сборки и среды выполнения** в самом начале.

Кроме того, вы можете настроить **редактор** DGML для отображения диаграмм в средстве визуализации:

1. Установите флажок **Редактор** DGML. Нужный пакет будет представлен в разделе **Средства для работы с кодом**.

Неизменность и платформа компилятора .NET

Неизменность является фундаментальным принципом платформы компилятора .NET. Неизменяемые структуры данных невозможно изменить после их создания. Такие структуры данных могут безопасно

анализироваться и совместно использоваться несколькими объектами-получателями одновременно. Не возникает опасности, что один объект-получатель непредсказуемо повлияет на работу других. Это означает, что анализатору не нужно применять блокировки или другие меры для поддержки параллелизма. Это правило применяется к деревьям синтаксиса, компиляциям, символам, семантическим моделям и любым другим структурам данных, с которыми вы можете работать. Вместо того чтобы изменять существующие структуры, интерфейсы API создают новые объекты с учетом требуемых изменений. Чтобы применить этот принцип к деревьям синтаксиса, после преобразования следует создавать новые деревья.

Создание и преобразование деревьев

Для синтаксических преобразований можно выбрать одну из двух стратегий. **Фабричные методы** лучше всего использовать при поиске конкретных узлов для замены или конкретных расположений, в которые нужно вставить новый код. **Модули записи** лучше подходят, когда нужно просканировать весь проект и найти структуры кода, которые нужно заменить.

Создание узлов с помощью фабричных методов

В первом синтаксическом преобразовании демонстрируется использование фабричных методов. Здесь вы замените инструкцию `using System.Collections;` инструкцией `using System.Collections.Generic;`. В этом примере показано, как создать объекты Microsoft.CodeAnalysis.CSharp.CSharpSyntaxNode с помощью фабричных методов Microsoft.CodeAnalysis.CSharp.SyntaxFactory. Для каждого типа **узла**, **маркера** или **trivia** имеется фабричный метод, создающий экземпляр этого типа. Чтобы создать синтаксическое дерево, нужно иерархически создавать его узлы, начиная снизу. Затем преобразуйте существующую программу, заменив существующие узлы только что созданным деревом.

Запустите Visual Studio и создайте проект C# для **автономного средства анализа кода**. В Visual Studio последовательно выберите **Файл** > **Создать** > **Проект**, чтобы открыть диалоговое окно "Новый проект". В разделе **Visual C#** > **Расширяемость** выберите **Stand-Alone Code Analysis Tool** (Автономное средство анализа кода). Это краткое руководство содержит два примера проектов. Присвойте решению имя **SyntaxTransformationQuickStart**, а проекту — имя **ConstructionCS**. Нажмите кнопку **ОК**.

Этот проект использует методы класса Microsoft.CodeAnalysis.CSharp.SyntaxFactory для создания объекта Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax, представляющего пространство имен `System.Collections.Generic`.

Добавьте следующую директиву в начало `Program.cs`.

```
using static Microsoft.CodeAnalysis.CSharp.SyntaxFactory;
using static System.Console;
```

Создайте **узлы синтаксиса имени** для построения дерева, представляющего инструкцию `using System.Collections.Generic;`. NameSyntax является базовым классом для четырех типов имен, применяющихся в C#. Сочетания этих четырех типов имен позволяют создать любое имя, которое может присутствовать в языке C#:

- Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax представляет имена простых одиночных идентификаторов, например `System` и `Microsoft`;
- Microsoft.CodeAnalysis.CSharp.Syntax.GenericNameSyntax представляет имя универсального типа или метода, например `List<int>`;
- Microsoft.CodeAnalysis.CSharp.Syntax.QualifiedNameSyntax представляет полное имя в формате `<left-name>.<right-identifier-or-generic-name>`, например `System.IO`;
- Microsoft.CodeAnalysis.CSharp.Syntax.AliasQualifiedNameSyntax представляет выражение имени через

внешний псевдоним сборки, например `LibraryV2::Foo`.

Используйте метод IdentifierName(String), чтобы создать узел NameSyntax. Добавьте приведенный ниже код в метод `Main` объекта `Program.cs`:

```
NameSyntax name = IdentifierName("System");
WriteLine($"\tCreated the identifier {name}");
```

В представленном выше коде создается объект IdentifierNameSyntax и присваивается его переменной `name`. Многие API-интерфейсы Roslyn возвращают базовые классы, что упрощает работу со связанными типами. Переменную `name` класса NameSyntax можно использовать многократно при создании QualifiedNameSyntax. Не используйте определение типа при создании образца. Этот шаг для проекта вы сделаете автоматическим.

Итак, вы создали имя. Теперь нужно добавить в дерево больше узлов, создав QualifiedNameSyntax. Новое дерево использует `name` в качестве левой части имени, а новый объект IdentifierNameSyntax в качестве пространства имен `Collections`, то есть в правой части QualifiedNameSyntax. Добавьте следующий код в файл `program.cs`:

```
name = QualifiedName(name, IdentifierName("Collections"));
WriteLine(name.ToString());
```

Снова выполните этот код и проверьте результаты. Вы создаете дерево узлов, представляющее код. Точно так же вы создадите QualifiedNameSyntax для пространства имен `System.Collections.Generic`. Добавьте следующий код в файл `Program.cs`:

```
name = QualifiedName(name, IdentifierName("Generic"));
WriteLine(name.ToString());
```

Запустите программу еще раз и убедитесь, что вы создаете дерево для добавления кода.

Создание измененного дерева

На текущий момент вы создали небольшое синтаксическое дерево, которое содержит одну инструкцию. API-интерфейсы для создания узлов отлично подходят для создания блоков небольшого размера, например для одного оператора. Но для более крупных блоков кода следует использовать методы, которые заменяют или добавляют узлы в существующее дерево. Не забывайте, что синтаксические деревья являются неизменяемыми. **API синтаксиса** не предоставляет механизмов для изменения существующего синтаксического дерева после его создания. Но зато в нем есть методы, которые позволяют создавать новые деревья на основе существующих с определенными изменениями. Методы `With*` определяются в конкретных классах, производных от SyntaxNode, или в методах расширения, объявленных в классе SyntaxNodeExtensions. Эти методы позволяют создать узел, применяя изменения к дочерним свойствам существующего узла. Кроме того, можно использовать метод расширения ReplaceNode для замены узлов-потомков в поддереве. Этот метод также позволяет обновить родительский узел, чтобы он указывал на новый дочерний элемент, и повторяет этот процесс вверх по всему дереву. Этот процесс называется *повторным прохождением*.

На следующем шаге вы создадите дерево, которое представляет целую (небольшую) программу и измените его. Добавьте следующий код в начало класса `Program`:

```
        private const string sampleCode =
@"using System;
using System.Collections;
using System.Linq;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(""Hello, World!"");
        }
    }
}";
```

NOTE

В примере кода используется пространство имен `System.Collections`, а не `System.Collections.Generic`.

Добавьте следующий код в нижнюю часть метода `Main`, чтобы выполнить синтаксический анализ текста и создать дерево:

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(sampleCode);
var root = (CompilationUnitSyntax)tree.GetRoot();
```

В этом примере используется метод WithName(NameSyntax), чтобы заменить имя в узле UsingDirectiveSyntax тем именем, созданном в предыдущем фрагменте кода.

Создайте узел UsingDirectiveSyntax с помощью метода WithName(NameSyntax), чтобы заменить имя `System.Collections` тем именем, которое создано в предыдущем фрагменте кода. Добавьте следующий код в конец метода `Main`:

```
var oldUsing = root.Usings[1];
var newUsing = oldUsing.WithName(name);
WriteLine(root.ToString());
```

Запустите программу и внимательно изучите выходные данные. `newUsing` не помещается в корневой узел дерева. Исходное дерево не изменяется.

Добавьте следующий код, используя метод расширения ReplaceNode, чтобы создать новое дерево. Новое дерево формируется путем замены существующего импорта обновленным узлом `newUsing`. Назначьте это новое дерево существующей переменной `root`:

```
root = root.ReplaceNode(oldUsing, newUsing);
WriteLine(root.ToString());
```

Запустите программу еще раз. На этот раз дерево правильно импортирует пространство имен `System.Collections.Generic`.

Преобразование деревьев с помощью `SyntaxRewriters`

Методы `With*` и ReplaceNode удобны для преобразования отдельных ветвей в синтаксическом дереве. Класс Microsoft.CodeAnalysis.CSharp.CSharpSyntaxRewriter выполняет множественные преобразования

синтаксического дерева. Класс Microsoft.CodeAnalysis.CSharp.CSharpSyntaxRewriter является подклассом Microsoft.CodeAnalysis.CSharp.CSharpSyntaxVisitor<TResult>. CSharpSyntaxRewriter применяет преобразование к конкретному типу объекта SyntaxNode. Преобразования можно применить к нескольким типам объектов SyntaxNode везде, где они встречаются в синтаксическом дереве. В втором проекте в этом кратком руководстве выполняется рефакторинг в командной строке, при котором удаляются явные типы из объявлений локальных переменных везде, где используются определения типа.

Создайте новый проект C# для **автономного средства анализа кода**. В Visual Studio щелкните правой кнопкой мыши узел решения `SyntaxTransformationQuickStart`. Выберите **Добавить** > **Новый проект**, чтобы открыть диалоговое окно **Новый проект**. В разделе Visual C# > **Расширяемость** выберите **Автономное средство анализа кода**. Присвойте проекту имя `TransformationCS` и нажмите кнопку "OK".

На первом шаге создается класс, производный от CSharpSyntaxRewriter, чтобы выполнять преобразования. Добавьте в проект новый файл класса. В Visual Studio выберите **Проект** > **Добавить класс...** . В диалоговом окне **Добавление нового элемента** введите имя файла `TypeInferenceRewriter.cs`.

В файл `TypeInferenceRewriter.cs` добавьте следующие директивы using:

```
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;
```

Теперь укажите класс `TypeInferenceRewriter` как расширение класса CSharpSyntaxRewriter:

```
public class TypeInferenceRewriter : CSharpSyntaxRewriter
```

Добавьте следующий код, чтобы объявить частное поле только для чтения, в котором будет храниться объект SemanticModel, и инициализируйте его в конструкторе. Это поле потребуется вам позже, чтобы указать, где можно использовать определение типа:

```
private readonly SemanticModel SemanticModel;

public TypeInferenceRewriter(SemanticModel semanticModel) => SemanticModel = semanticModel;
```

Переопределите метод VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax):

```
public override SyntaxNode VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax node)
{

}
```

> **NOTE**
> Многие API-интерфейсы Roslyn объявляют типы возвращаемых значений, которые являются базовыми классами для фактически возвращаемых типов среды выполнения. Во многих сценариях один тип узла может быть заменен другим типом или даже удален. В этом примере метод VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax) возвращает SyntaxNode, а не тип, производный от LocalDeclarationStatementSyntax. Этот модуль записи возвращает новый узел LocalDeclarationStatementSyntax, основанный на существующем узле.

В этом кратком руководстве рассматриваются объявления локальных переменных. Вы можете добавить в

пример кода другие объявления, например циклов `foreach`, циклов `for`, выражений LINQ и лямбда-выражений. Также этот модуль записи преобразует объявления только в простейшем формате:

```
Type variable = expression;
```

Если вы хотите поэкспериментировать самостоятельно, в готовый пример попробуйте добавить объявления переменных следующих типов:

```
// Multiple variables in a single declaration.
Type variable1 = expression1,
     variable2 = expression2;
// No initializer.
Type variable;
```

Добавьте следующий код в текст метода `VisitLocalDeclarationStatement`, чтобы пропускать перезапись этих форм объявления:

```
if (node.Declaration.Variables.Count > 1)
{
    return node;
}
if (node.Declaration.Variables[0].Initializer == null)
{
    return node;
}
```

Этот метод указывает, что перезапись не происходит, возвращая неизмененное значение параметра `node`. Если ни одно из выражений `if` не принимает значение true, этот узел представляет возможное объявление с инициализацией. Добавьте следующие инструкции, чтобы извлечь имя типа, указанное в объявлении, и связать его с помощью поля SemanticModel для получения символа типа:

```
var declarator = node.Declaration.Variables.First();
var variableTypeName = node.Declaration.Type;

var variableType = (ITypeSymbol)SemanticModel
    .GetSymbolInfo(variableTypeName)
    .Symbol;
```

Теперь, чтобы связать выражение инициализатора, добавьте такое выражение:

```
var initializerInfo = SemanticModel.GetTypeInfo(declarator.Initializer.Value);
```

Наконец, добавьте следующую инструкцию `if`, чтобы заменять существующее имя типа ключевым словом `var`, если тип выражения инициализатора соответствует указанному типу:

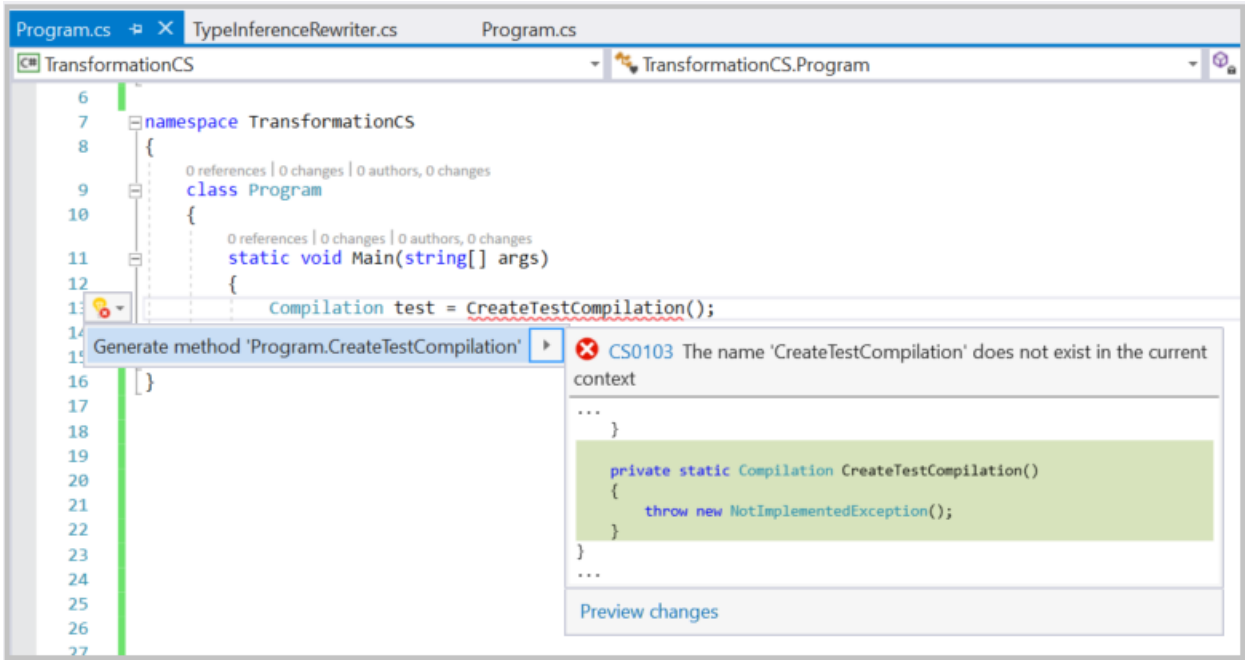
```
if (SymbolEqualityComparer.Default.Equals(variableType, initializerInfo.Type))
{
    TypeSyntax varTypeName = SyntaxFactory.IdentifierName("var")
        .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())
        .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());

    return node.ReplaceNode(variableTypeName, varTypeName);
}
else
{
    return node;
}
```

Условная запись нужна потому, что объявление может приводить выражение инициализатора к базовому классу или интерфейсу. Если вам потребуется такое поведение, типы слева и справа от оператора назначения не будут совпадать. Удаление явного указания типа в таких случаях изменит семантику программы. `var` задается как идентификатор, а не ключевое слово, так как `var` является контекстным ключевым словом. Начальные и конечные элементы trivia (пробелы) передаются из старого имени типа в ключевое слово `var`, чтобы сохранить вертикальные пробелы и отступы. Гораздо проще использовать `ReplaceNode` вместо `With*` для преобразования LocalDeclarationStatementSyntax, так как имя типа по сути является внучатым элементом для оператора объявления.

Итак, вы завершили создание `TypeInferenceRewriter`. Теперь вернитесь к файлу `Program.cs`, чтобы завершить работу с этим примером. Создайте тест Compilation и получите из него SemanticModel. Используйте этот SemanticModel для проверки `TypeInferenceRewriter`. Это будет последний шаг процедуры. Одновременно с этим объявите переменную, которая будет выполнять роль заполнителя для тестовой компиляции:

```
Compilation test = CreateTestCompilation();
```

Подождите пару секунд, и в этой строке появится волнистая линия, которая обозначает ошибку: метод `CreateTestCompilation` не существует. Нажмите клавиши CTRL+**точка**, чтобы открыть меню лампочки. Затем нажмите клавишу "ВВОД", чтобы вызвать команду **Сгенерировать заглушку метода**. Эта команда создаст заглушку для метода `CreateTestCompilation` в классе `Program`. Этот метод вы заполните позднее:



Создайте следующий код, который последовательно перебирает все SyntaxTree в тесте Compilation. Для

каждого из них инициализируйте новый `TypeInferenceRewriter` с классом SemanticModel для этого дерева:

```
foreach (SyntaxTree sourceTree in test.SyntaxTrees)
{
    SemanticModel model = test.GetSemanticModel(sourceTree);

    TypeInferenceRewriter rewriter = new TypeInferenceRewriter(model);

    SyntaxNode newSource = rewriter.Visit(sourceTree.GetRoot());

    if (newSource != sourceTree.GetRoot())
    {
        File.WriteAllText(sourceTree.FilePath, newSource.ToFullString());
    }
}
```

В созданной инструкции `foreach` добавьте приведенный ниже код, чтобы выполнить преобразование для каждого исходного дерева. Этот код проверяет, были ли внесены правки, и если были — записывает преобразованное дерево. Модуль записи должен изменять дерево только в том случае, если он встречает одно или несколько объявлений локальных переменных, которые можно упростить с помощью определения типа:

```
SyntaxNode newSource = rewriter.Visit(sourceTree.GetRoot());

if (newSource != sourceTree.GetRoot())
{
    File.WriteAllText(sourceTree.FilePath, newSource.ToFullString());
}
```

Вы увидите волнистые линии под фрагментом кода `File.WriteAllText`. Выберите значок лампочки и добавьте необходимую инструкцию `using System.IO;`.

Вы почти закончили! Осталось только последнее действие: создать тест Compilation. Так как вы еще не использовали определение типа в этом кратком руководстве, это будет отличным тестовым случаем. К сожалению, создание компиляции из файла проекта C# не входит в это пошаговое руководство. Но вы можете применить обходной путь, если внимательно выполняли все инструкции. Замените содержимое метода `CreateTestCompilation` следующим кодом. Этот код позволяет создать компиляцию теста, которая очень удачно совпадает с проектом, описанным в этом кратком руководстве:

```
String programPath = @"..\..\..\Program.cs";
String programText = File.ReadAllText(programPath);
SyntaxTree programTree =
               CSharpSyntaxTree.ParseText(programText)
                               .WithFilePath(programPath);

String rewriterPath = @"..\..\..\TypeInferenceRewriter.cs";
String rewriterText = File.ReadAllText(rewriterPath);
SyntaxTree rewriterTree =
               CSharpSyntaxTree.ParseText(rewriterText)
                               .WithFilePath(rewriterPath);

SyntaxTree[] sourceTrees = { programTree, rewriterTree };

MetadataReference mscorlib =
        MetadataReference.CreateFromFile(typeof(object).Assembly.Location);
MetadataReference codeAnalysis =
        MetadataReference.CreateFromFile(typeof(SyntaxTree).Assembly.Location);
MetadataReference csharpCodeAnalysis =
        MetadataReference.CreateFromFile(typeof(CSharpSyntaxTree).Assembly.Location);

MetadataReference[] references = { mscorlib, codeAnalysis, csharpCodeAnalysis };

return CSharpCompilation.Create("TransformationCS",
    sourceTrees,
    references,
    new CSharpCompilationOptions(OutputKind.ConsoleApplication));
```

Скрестите пальцы и запустите проект. В Visual Studio выберите **Отладка** > **Начать отладку**. В Visual Studio появится сообщение о том, что файлы в проекте были изменены. Щелкните **Да для всех**, чтобы повторно загрузить все измененные файлы. Изучите их и поразитесь, что вам удалось сделать. Обратите особое внимание, насколько чище выглядит код без ненужных явных описателей типов.

Поздравляем! С помощью API-**интерфейсов компилятора** вы создали собственное средство для рефакторинга, которое ищет определенные синтаксические выражения во всех файлах проекта C#, анализирует и преобразует семантику исходного кода по указанным шаблонам. Теперь вы официально считаетесь разработчиком средства рефакторинга!

Учебник. Создание средства для анализа и исправления кода

18.12.2021 • 20 minutes to read

Пакет SDK для .NET Compiler Platform предоставляет средства, необходимые для создания пользовательской диагностики (анализаторов), исправления и рефакторинга кода, а также подавления диагностики для кода на C# или Visual Basic. **Анализатор** содержит код, который распознает нарушения правила. **Исправление кода** содержит код, который исправляет эти нарушения. Правилами, которые вы реализуете, может быть что угодно — от структуры кода до его стиля или соглашений об именовании и многое другое. .NET Compiler Platform предоставляет платформу для выполнения анализа во время написания кода, а также все функции пользовательского интерфейса Visual Studio для отладки, включая отображение волнистых линий в редакторе, вывод списка ошибок в Visual Studio и отображение значка лампочки, указывающего на наличие предложений и предлагаемых исправлений.

В этом руководстве описано, как создать **анализатор** и соответствующее **исправление кода** с помощью API Roslyn. Анализатор выполняет анализ исходного кода и сообщает о проблеме пользователю. При необходимости исправление кода можно связать с анализатором, чтобы представить изменения в исходном коде пользователя. В этом руководстве описано, как создать анализатор, ищущий локальные переменные, которые можно объявить с помощью модификатора `const`, но которые не объявлены. Сопутствующее исправление кода добавляет модификатор `const` в эти объявления.

Предварительные требования

- Visual Studio 2019 версии 16.8 или более поздней.

Необходимо установить **пакет SDK для .NET Compiler Platform** через Visual Studio Installer:

Инструкции по установке — Visual Studio Installer

Найти SDK-**пакет .NET Compiler Platform** в **Visual Studio Installer** можно двумя способами:

Установка с помощью Visual Studio Installer — представление "Рабочие нагрузки"

SDK-пакет .NET Compiler Platform не выбирается автоматически в рамках рабочей нагрузки разработки расширений Visual Studio. Его необходимо выбрать как дополнительный компонент.

1. Запустите **Visual Studio Installer**.
2. Выберите **Изменить**.
3. Отметьте рабочую нагрузку **Разработка расширений Visual Studio**.
4. Откройте узел **Разработка расширений Visual Studio** в дереве сводки.
5. Установите флажок **SDK-пакет .NET Compiler Platform**. Нужный пакет будет представлен последним в списке дополнительных компонентов.

Кроме того, вы можете настроить **редактор** DGML для отображения диаграмм в средстве визуализации:

1. Откройте узел **Отдельные компоненты** в дереве сводки.
2. Установите флажок **Редактор** DGML

Установка с помощью Visual Studio Installer — вкладка "Отдельные компоненты"

1. Запустите **Visual Studio Installer**.

2. Выберите **Изменить**.
3. Откройте вкладку **Отдельные компоненты**.
4. Установите флажок SDK-**пакет** .NET Compiler Platform. Нужный пакет будет представлен в разделе **Компиляторы, средства сборки и среды выполнения** в самом начале.

Кроме того, вы можете настроить **редактор** DGML для отображения диаграмм в средстве визуализации:

1. Установите флажок **Редактор** DGML. Нужный пакет будет представлен в разделе **Средства для работы с кодом**.

Создать и проверить анализатор можно несколькими способами:

1. Создайте решение.
2. Зарегистрируйте имя и описание анализатора.
3. Создайте отчет анализатора о предупреждениях и рекомендациях.
4. Внедрите исправление кода, чтобы принимать рекомендации.
5. Улучшите анализ с помощью модульных тестов.

Создание решения

- В Visual Studio последовательно выберите **Файл** > **Создать** > **Проект**, чтобы открыть диалоговое окно "Новый проект".
- В разделе **Visual C#** > **Расширяемость** выберите **Analyzer with code fix (.NET Standard)** (Анализатор с исправлением кода (.NET Standard)).
- Присвойте проекту имя **MakeConst** и нажмите кнопку "OK".

Изучение шаблона анализатора

Анализатор с шаблоном исправления кода создает пять проектов:

- **MakeConst** — включает анализатор;
- **MakeConst.CodeFixes** — включает исправление кода;
- **MakeConst.Package** — используется для создания пакета NuGet для анализатора и исправления кода;
- **MakeConst.Test** — проект модульного теста;
- **MakeConst.Vsix** — проект запуска по умолчанию, который запускает второй экземпляр Visual Studio с загруженным новым анализатором. Нажмите клавишу F5, чтобы запустить проект VSIX.

> **NOTE**
>
> Анализаторам следует использовать .NET Standard 2.0, так как они могут запускаться в средах .NET Core (сборки командной строки) и .NET Framework (Visual Studio).

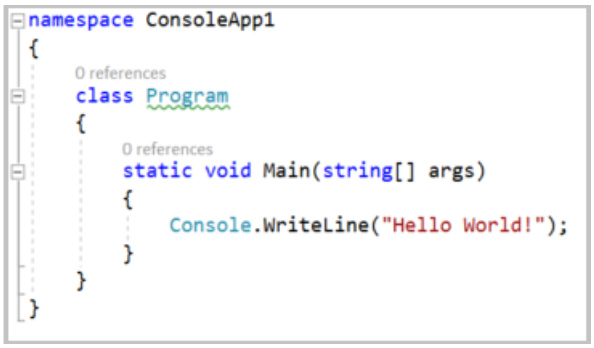
TIP

Когда вы запустите анализатор, откроется вторая копия Visual Studio. Эта вторая копия использует другой куст реестра для хранения параметров, что позволяет различить параметры визуальных элементов в обоих копиях Visual Studio. Вы можете выбрать другую тему для экспериментального запуска Visual Studio. Кроме того, не следует перемещать параметры или выполнять вход в учетную запись Visual Studio в экспериментальном экземпляре Visual Studio. Так параметры останутся разными.

В состав куста входит не только разрабатываемый анализатор, но и все открытые ранее. Чтобы сбросить куст Roslyn, необходимо вручную удалить его из раздела *LocalAppData%\Microsoft\VisualStudio*. Имя папки куста Roslyn будет заканчиваться на `Roslyn`, например `16.0_9ae182f9Roslyn`. Обратите внимание, что после удаления куста может потребоваться очистить решение и перестроить его.

Во втором экземпляре Visual Studio, который вы только что запустили, создайте проект C# консольного приложения (целевая платформа может быть любой, так как анализаторы работают на уровне исходного кода). Наведите указатель мыши на токен с волнистым подчеркиванием. Появится текст предупреждения от анализатора.

Шаблон создает анализатор, который выдает предупреждение на каждое объявление типа, где имя типа состоит из букв нижнего регистра, как показано ниже:



Также шаблон содержит исправление кода, которое меняет любые буквы нижнего регистра в имени типа на буквы верхнего регистра. Предлагаемые исправления можно просмотреть, щелкнув значок лампочки рядом с предупреждением. После принятия изменений имя типа и все ссылки на этот тип будут обновлены. Теперь, когда вы увидели работу начального анализатора, закройте второй экземпляр Visual Studio и вернитесь к проекту анализатора.

Для тестирования изменений в анализаторе не требуется каждый раз запускать вторую копию Visual Studio, так как шаблон создает проект модульного теста. В этом проекте содержатся два теста. `TestMethod1` показывает обычный формат теста, при котором анализ кода происходит без активации диагностики. `TestMethod2` — формат, при котором сначала активируется диагностика, а затем применяется исправление кода. Во время сборки анализатора и исправления кода вы напишете тесты для проверки разных структур. Модульные тесты проводятся гораздо быстрее, чем интерактивное тестирование анализаторов в Visual Studio.

TIP

Модульные тесты анализатора — отличный инструмент, если вы знаете, какие конструкции кода должны и не должны запускать анализатор. В свою очередь запуск анализатора в другой копии Visual Studio позволяет определить и найти конструкции, о которых вы еще не задумывались.

В этом руководстве показано, как написать анализатор, информирующий пользователя о любых объявлениях локальной переменной, которые можно преобразовать в локальные константы. Рассмотрим следующий пример кода:

```
int x = 0;
Console.WriteLine(x);
```

В приведенном выше коде `x` присваивается значение константы, которое никогда не меняется. Ее можно объявить с помощью модификатора `const`:

```
const int x = 0;
Console.WriteLine(x);
```

Чтобы определить, можно ли изменить переменную на константу, используется синтаксический анализ, анализ константы из выражения инициализатора, а также анализ потока данных, чтобы убедиться, что переменная не записана. .NET Compiler Platform предоставляет API для упрощения такого анализа.

Регистрация анализатора

В файле *MakeConstAnalyzer.cs* с помощью шаблона создается начальный класс `DiagnosticAnalyzer`. В этом начальном анализаторе отображены два важных свойства каждого анализатора.

- В каждом диагностическом анализаторе должен быть указан атрибут `[DiagnosticAnalyzer]`, который описывает язык, на котором он работает.
- Каждый диагностический анализатор должен быть производным (прямо или косвенно) от класса DiagnosticAnalyzer.

Также в шаблоне отображены базовые функции любого анализатора:

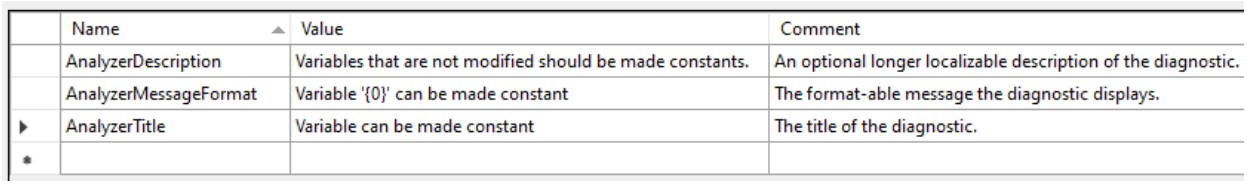
1. Регистрация действий. Действия представляют изменения кода, которые запускают анализатор для проверки нарушений. Когда Visual Studio обнаруживает изменения в коде, которые соответствуют зарегистрированному действию, происходит вызов зарегистрированного метода.
2. Создание диагностики. Обнаружив нарушения, анализатор создает объект диагностики, с помощью которого Visual Studio уведомляет пользователя об этом нарушении.

Действия регистрируются в переопределении метода DiagnosticAnalyzer.Initialize(AnalysisContext). При работе с этим руководстве вы просмотрите **синтаксические узлы** локальных объявлений и узнаете, какие из них имеют значения констант. Если объявление может быть константой, анализатор создаст диагностику и сформирует отчет.

Сначала обновите константы регистрации и метод `Initialize`, так как константы определяют ваш анализатор MakeConst. Большинство строковых констант определены в файле строковых ресурсов. Используйте их, чтобы упростить локализацию. Откройте файл *Resources.resx* для проекта анализатора **MakeConst**. Откроется редактор ресурсов. Измените строковые ресурсы, как показано ниже:

- Измените `AnalyzerDescription` на Variables that are not modified should be made constants..
- Измените `AnalyzerMessageFormat` на Variable '{0}' can be made constant.
- Измените `AnalyzerTitle` на Variable can be made constant.

После настройки редактор ресурсов будет выглядеть как на рисунке ниже:



	Name	Value	Comment
	AnalyzerDescription	Variables that are not modified should be made constants.	An optional longer localizable description of the diagnostic.
	AnalyzerMessageFormat	Variable '{0}' can be made constant	The format-able message the diagnostic displays.
▶	AnalyzerTitle	Variable can be made constant	The title of the diagnostic.
*			

Остальные изменения будут в файле анализатора. Откройте файл *MakeConstAnalyzer.cs* в Visual Studio. Измените зарегистрированное действие на символы на действие на синтаксис. В методе

`MakeConstAnalyzerAnalyzer.Initialize` найдите строку с действием на символы:

```
context.RegisterSymbolAction(AnalyzeSymbol, SymbolKind.NamedType);
```

Замените ее приведенным ниже кодом:

```
context.RegisterSyntaxNodeAction(AnalyzeNode, SyntaxKind.LocalDeclarationStatement);
```

После этого метод `AnalyzeSymbol` можно удалить. Этот анализатор проверяет SyntaxKind.LocalDeclarationStatement, а не операторы SymbolKind.NamedType. Обратите внимание на то, что `AnalyzeNode` подчеркивается красной волнистой линией, так как код, который вы только что вставили, ссылается на метод `AnalyzeNode`, который еще не объявлен. Объявите этот метод, используя приведенный ниже код:

```
private void AnalyzeNode(SyntaxNodeAnalysisContext context)
{
}
```

Измените `Category` на Usage в файле *MakeConstAnalyzer.cs*, как показано в следующем коде:

```
private const string Category = "Usage";
```

Поиск локальных объявлений, которые могут быть константами

Теперь мы напишем первую версию метода `AnalyzeNode`. Он должен найти одно локальное объявление, которое может быть `const`, но таковым не является. Пример приведен ниже:

```
int x = 0;
Console.WriteLine(x);
```

Сначала найдите локальные объявления. Добавьте приведенный ниже код в `AnalyzeNode` в файле *MakeConstAnalyzer.cs*:

```
var localDeclaration = (LocalDeclarationStatementSyntax)context.Node;
```

Это приведение всегда завершается успешно, так как ваш анализатор зарегистрирован для отслеживания изменений только локальных объявлений. Другие типы узлов не вызывают метод `AnalyzeNode`. Затем проверьте объявление на наличие модификаторов `const`. Если они есть, сразу же выполните возврат. Приведенный ниже код ищет модификаторы `const` в локальном объявлении:

```
// make sure the declaration isn't already const:
if (localDeclaration.Modifiers.Any(SyntaxKind.ConstKeyword))
{
    return;
}
```

В конце проверьте, может ли переменная быть `const`. Это означает, что она не может быть назначена после инициализации.

Выполните семантический анализ с помощью SyntaxNodeAnalysisContext. Используйте аргумент `context`,

чтобы определить, можно ли объявление локальной переменной сделать `const`. Microsoft.CodeAnalysis.SemanticModel представляет все семантические сведения в одном исходном файле. См. дополнительные сведения о семантических моделях. С помощью Microsoft.CodeAnalysis.SemanticModel выполните анализ потока данных на операторе локального объявления. Затем, используя результаты этого анализа потока данных, убедитесь, что в локальную переменную не записывается новое значение где-либо еще. Вызовите метод расширения GetDeclaredSymbol, чтобы извлечь ILocalSymbol переменной и убедиться, что она не содержится в коллекции DataFlowAnalysis.WrittenOutside из анализа потока данных. Добавьте в конце метода `AnalyzeNode` приведенный ниже код:

```
// Perform data flow analysis on the local declaration.
DataFlowAnalysis dataFlowAnalysis = context.SemanticModel.AnalyzeDataFlow(localDeclaration);

// Retrieve the local symbol for each variable in the local declaration
// and ensure that it is not written outside of the data flow analysis region.
VariableDeclaratorSyntax variable = localDeclaration.Declaration.Variables.Single();
ISymbol variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable, context.CancellationToken);
if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
{
    return;
}
```

Этот код гарантирует, что переменная не изменена и может быть `const`. Теперь мы активируем диагностику. Добавьте приведенный ниже код в последнюю строку метода `AnalyzeNode`:

```
context.ReportDiagnostic(Diagnostic.Create(Rule, context.Node.GetLocation(),
localDeclaration.Declaration.Variables.First().Identifier.ValueText));
```

Чтобы проверить состояние, запустите анализатор, нажав клавишу F5. Загрузите консольное приложение, созданное ранее, и добавьте приведенный ниже код теста:

```
int x = 0;
Console.WriteLine(x);
```

Появится лампочка и анализатор выдаст отчет с диагностическими сведениями. При этом поведение лампочки по-прежнему основано на исправлении кода, сгенерированном шаблоном, указывая на то, что можно использовать буквы верхнего регистра. В следующем разделе показано, как написать исправление кода.

Написание исправления кода

Анализатор поддерживает одно или несколько исправлений кода. Исправление кода определяет, какие правки нужно внести для решения обнаруженной проблемы. Исправление кода вашего анализатора предоставляет код с ключевым словом const:

```
- int x = 0;
+ const int x = 0;
Console.WriteLine(x);
```

Пользователь выбирает исправление в интерфейсе лампочки в редакторе, а Visual Studio изменяет код.

Откройте файл *CodeFixResources.resx* и измените `CodeFixTitle` на Make constant.

Откройте файл *MakeConstCodeFixProvider.cs*, добавленный шаблоном. Это исправление уже привязано к идентификатору диагностики вашего анализатора, но оно пока не реализует преобразование кода.

Затем удалите метод `MakeUppercaseAsync`. Он больше не применяется.

Все поставщики исправлений кода являются производными от CodeFixProvider. и переопределяют CodeFixProvider.RegisterCodeFixesAsync(CodeFixContext) на сообщение о доступных исправлениях. В LocalDeclarationStatementSyntax измените тип узла-предка на `RegisterCodeFixesAsync` в соответствии с диагностикой:

```
var declaration =
root.FindToken(diagnosticSpan.Start).Parent.AncestorsAndSelf().OfType<LocalDeclarationStatementSyntax>
().First();
```

Затем, чтобы зарегистрировать исправление кода, измените последнюю строку. Исправление создаст документ, полученный после добавления модификатора `const` к существующему объявлению:

```
// Register a code action that will invoke the fix.
context.RegisterCodeFix(
    CodeAction.Create(
        title: CodeFixResources.CodeFixTitle,
        createChangedDocument: c => MakeConstAsync(context.Document, declaration, c),
        equivalenceKey: nameof(CodeFixResources.CodeFixTitle)),
    diagnostic);
```

Под символом `MakeConstAsync` появятся красные волнистые линии. Добавьте объявление в `MakeConstAsync`, например как указано ниже:

```
private static async Task<Document> MakeConstAsync(Document document,
    LocalDeclarationStatementSyntax localDeclaration,
    CancellationToken cancellationToken)
{
}
```

Новый метод `MakeConstAsync` преобразует Document исходного файла пользователя в новый экземпляр Document, содержащий объявление `const`.

Создайте новый токен ключевого слова `const` и вставьте его перед оператором объявления. Не забудьте сначала удалить все элементы trivia из первого оператора объявления и подключить к токену `const`. Добавьте следующий код в метод `MakeConstAsync`:

```
// Remove the leading trivia from the local declaration.
SyntaxToken firstToken = localDeclaration.GetFirstToken();
SyntaxTriviaList leadingTrivia = firstToken.LeadingTrivia;
LocalDeclarationStatementSyntax trimmedLocal = localDeclaration.ReplaceToken(
    firstToken, firstToken.WithLeadingTrivia(SyntaxTriviaList.Empty));

// Create a const token with the leading trivia.
SyntaxToken constToken = SyntaxFactory.Token(leadingTrivia, SyntaxKind.ConstKeyword,
SyntaxFactory.TriviaList(SyntaxFactory.ElasticMarker));
```

Затем добавьте токен `const` к объявлению с помощью приведенного ниже кода:

```
// Insert the const token into the modifiers list, creating a new modifiers list.
SyntaxTokenList newModifiers = trimmedLocal.Modifiers.Insert(0, constToken);
// Produce the new local declaration.
LocalDeclarationStatementSyntax newLocal = trimmedLocal
    .WithModifiers(newModifiers)
    .WithDeclaration(localDeclaration.Declaration);
```

Отформатируйте новое объявление в соответствии с правилами форматирования C#. Форматирование изменений в соответствии с существующим кодом упрощает работу. Добавьте приведенный ниже оператор сразу после существующего кода:

```
// Add an annotation to format the new local declaration.
LocalDeclarationStatementSyntax formattedLocal = newLocal.WithAdditionalAnnotations(Formatter.Annotation);
```

Для выполнения этого кода требуется новое пространство имен. Добавьте следующую директиву `using` в начало файла.

```
using Microsoft.CodeAnalysis.Formatting;
```

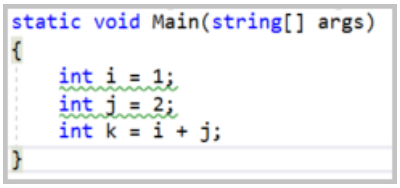
В конце нужно внести правки. Для этого выполните эти три шага:

1. Получите дескриптор существующего документа.
2. Создайте документ, заменив существующее объявление на новое.
3. Верните новый документ.

Добавьте в конце метода `MakeConstAsync` приведенный ниже код:

```
// Replace the old local declaration with the new local declaration.
SyntaxNode oldRoot = await document.GetSyntaxRootAsync(cancellationToken).ConfigureAwait(false);
SyntaxNode newRoot = oldRoot.ReplaceNode(localDeclaration, formattedLocal);

// Return document with transformed tree.
return document.WithSyntaxRoot(newRoot);
```

Ваше исправление кода готово. Нажмите клавишу F5, чтобы запустить проект анализатора во втором экземпляре Visual Studio. Во втором экземпляре Visual Studio создайте проект C# консольного приложения и добавьте несколько объявлений локальной переменной, инициализированных со значениями константы в методе main. Они будут отмечены как предупреждения. См. пример ниже.



Мы уже много сделали. Объявления, которые могут быть `const`, подчеркнуты волнистой линией. Но нам еще нужно с ними поработать. Здесь следует добавить `const` в объявления `i`, затем `j` и `k`. Но если вы добавите модификатор `const` в обратном порядке, начиная с `k`, анализатор выдаст ошибки: `k` не может быть объявлен как `const`, пока `i` и `j` не являются `const`. Нужно выполнить дополнительный анализ, чтобы убедиться, что переменные можно объявить и инициализировать различными путями.

Создание модульных тестов

Анализатор и исправление кода работают на простом примере одного объявления, которое может быть константой. Есть множество возможных операторов объявления, где они выдают ошибки. В этих ситуациях можно обратиться к библиотеке модульных тестов, созданной шаблоном. Это гораздо быстрее, чем каждый раз запускать вторую копию Visual Studio.

Откройте файл *MakeConstUnitTests.cs* в проекте модульного теста. В нем созданы два теста по общим шаблонам для анализатора и для модульного теста исправления кода. Метод `TestMethod1` гарантирует, что анализатор не выдаст отчет о диагностике, когда это не нужно. Метод `TestMethod2` выдает отчет о

диагностике и запускает исправление кода.

Этот шаблон использует пакеты Microsoft.CodeAnalysis.Testing для работы с модульными тестами.

> **TIP**
> Библиотека тестирования поддерживает специальный синтаксис разметки, в том числе следующие элементы:
>
> - `[|text|]` — указывает, что для `text` предоставляются данные диагностики. По умолчанию эта форма может использоваться только для анализаторов тестирования с одним экземпляром `DiagnosticDescriptor`, предоставляемым `DiagnosticAnalyzer.SupportedDiagnostics`.
> - `{|ExpectedDiagnosticId:text|}` — указывает, что для `text` предоставляются данные диагностики с Id `ExpectedDiagnosticId`.

Замените тесты шаблона в классе `MakeConstUnitTest` следующим методом теста.

```
        [TestMethod]
        public async Task LocalIntCouldBeConstant_Diagnostic()
        {
            await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
        [|int i = 0;|]
        Console.WriteLine(i);
    }
}
", @"
using System;

class Program
{
    static void Main()
    {
        const int i = 0;
        Console.WriteLine(i);
    }
}
");
        }
```

Запустите этот тест, чтобы убедиться, что он проходит. Откройте **обозреватель тестов** в Visual Studio, последовательно выбрав **Тест** > **Windows** > **Обозреватель тестов**. Затем выберите **Выполнить все**.

Создание тестов для допустимых объявлений

Как правило, анализаторы должны завершать работу как можно быстрее, выполняя минимум операций. Когда пользователь правит код, Visual Studio вызывает зарегистрированные анализаторы. Основным требованием здесь является скорость реагирования. Существует несколько тестовых случаев для кода, который не должен вызывать диагностику. Анализатор уже обрабатывает один из них — тот, при котором переменная присваивается после инициализации. Добавьте следующий метод теста, чтобы представить этот случай:

```
        [TestMethod]
        public async Task VariableIsAssigned_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int i = 0;
        Console.WriteLine(i++);
    }
}
");
        }
```

Этот тест также проходит. Далее добавьте методы теста для условий, которые еще не обработаны:

- Объявления, которые представляют `const`, так как они уже являются константами:

```
        [TestMethod]
        public async Task VariableIsAlreadyConst_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        const int i = 0;
        Console.WriteLine(i);
    }
}
");
        }
```

- Объявления, у которых нет инициализатора, так как нет соответствующего значения:

```
        [TestMethod]
        public async Task NoInitializer_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int i;
        i = 0;
        Console.WriteLine(i);
    }
}
");
        }
```

- Объявления, у которых инициализатор не является константой, так как они не могут быть константами времени компиляции:

```
        [TestMethod]
        public async Task InitializerIsNotConstant_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int i = DateTime.Now.DayOfYear;
        Console.WriteLine(i);
    }
}
");
        }
```

Трудность заключается еще в том, что в C# допускается несколько объявлений, работающих как один оператор. Рассмотрите приведенную ниже строковую константу тестового случая:

```
        [TestMethod]
        public async Task MultipleInitializers_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int i = 0, j = DateTime.Now.DayOfYear;
        Console.WriteLine(i);
        Console.WriteLine(j);
    }
}
");
        }
```

Переменная `i` может быть константой, а переменная `j` — нет. Поэтому этот оператор не может быть объявлением константы.

Снова запустите тесты. Произойдет сбой в новых тестовых случаях.

Обновление анализатора для пропуска верных объявлений

Чтобы отфильтровать код, который соответствует условиям, необходимо преобразовать метод `AnalyzeNode` анализатора. Эти условия связаны между собой, и изменения в одном из них будут применены ко всем. Внесите следующие изменения в `AnalyzeNode`:

- Для объявления одной переменной был выполнен семантический анализ. Этот код должен находиться в цикле `foreach`, который проверяет все переменные, объявленные в одном и том же операторе.
- Каждая объявленная переменная должна иметь инициализатор.
- Каждый инициализатор переменной должен быть константой времени компиляции.

В методе `AnalyzeNode` замените исходный семантический анализ:

```
// Perform data flow analysis on the local declaration.
DataFlowAnalysis dataFlowAnalysis = context.SemanticModel.AnalyzeDataFlow(localDeclaration);

// Retrieve the local symbol for each variable in the local declaration
// and ensure that it is not written outside of the data flow analysis region.
VariableDeclaratorSyntax variable = localDeclaration.Declaration.Variables.Single();
ISymbol variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable, context.CancellationToken);
if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
{
    return;
}
```

приведенным ниже фрагментом кода:

```
// Ensure that all variables in the local declaration have initializers that
// are assigned with constant values.
foreach (VariableDeclaratorSyntax variable in localDeclaration.Declaration.Variables)
{
    EqualsValueClauseSyntax initializer = variable.Initializer;
    if (initializer == null)
    {
        return;
    }

    Optional<object> constantValue = context.SemanticModel.GetConstantValue(initializer.Value,
context.CancellationToken);
    if (!constantValue.HasValue)
    {
        return;
    }
}

// Perform data flow analysis on the local declaration.
DataFlowAnalysis dataFlowAnalysis = context.SemanticModel.AnalyzeDataFlow(localDeclaration);

foreach (VariableDeclaratorSyntax variable in localDeclaration.Declaration.Variables)
{
    // Retrieve the local symbol for each variable in the local declaration
    // and ensure that it is not written outside of the data flow analysis region.
    ISymbol variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable, context.CancellationToken);
    if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
    {
        return;
    }
}
```

Первый цикл `foreach` проверяет каждое объявление переменной с помощью синтаксического анализа. В первой проверке подтверждается наличие инициализатора. Во второй — что этот инициализатор является константой. Во втором цикле запускается исходный семантический анализ. Семантические проверки проходят в отдельных циклах, так как они серьезно влияют на производительность. Снова запустите тест. Все проверки должны быть пройдены.

Финальные штрихи

Вы почти у цели. Анализатор должен обработать еще несколько условий. Пока пользователь пишет код, Visual Studio вызывает анализаторы. Часто бывает так, что анализатор вызван для кода, который не компилируется. Метод `AnalyzeNode` диагностического анализатора не проверяет, можно ли преобразовать значение константы в тип переменной. Поэтому в текущей реализации все неправильные объявления, такие как `int i = "abc"`, преобразуются в локальные константы. Добавьте метод теста для этого случая:

```
        [TestMethod]
        public async Task DeclarationIsInvalid_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int x = {|CS0029:""abc""|};
    }
}
");
        }
```

Кроме того, ссылочные типы обрабатываются неправильно. Единственное допустимое значение константы для ссылочного типа — `null`, кроме случаев с использованием System.String, когда допускаются строковые литералы. Другими словами, `const string s = "abc"` является допустимым, а `const object s = "abc"` — нет. Этот фрагмент кода проверяет это условие:

```
        [TestMethod]
        public async Task DeclarationIsNotString_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        object s = ""abc"";
    }
}
");
        }
```

Чтобы убедиться в том, что можно создать объявление константы для строки, добавьте еще один тест. В приведенным ниже фрагменте кода определен как код, вызывающий диагностику, так и код после исправления:

```
        [TestMethod]
        public async Task StringCouldBeConstant_Diagnostic()
        {
            await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
        [|string s = ""abc"";|]
    }
}
", @"
using System;

class Program
{
    static void Main()
    {
        const string s = ""abc"";
    }
}
");
        }
```

Если переменная объявлена с ключевым словом `var`, исправление выдает объявление `const var`, которое не поддерживается в C#. Чтобы исправить эту ошибку, исправление должно заменить ключевое слово `var` именем выведенного типа:

```
        [TestMethod]
        public async Task VarIntDeclarationCouldBeConstant_Diagnostic()
        {
            await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
        [|var item = 4;|]
    }
}
", @"
using System;

class Program
{
    static void Main()
    {
        const int item = 4;
    }
}
");
        }

        [TestMethod]
        public async Task VarStringDeclarationCouldBeConstant_Diagnostic()
        {
            await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
        [|var item = ""abc"";|]
    }
}
", @"
using System;

class Program
{
    static void Main()
    {
        const string item = ""abc"";
    }
}
");
        }
```

К счастью, все вышеперечисленные ошибки можно устранить с помощью методов, которые вы только что изучили.

Чтобы исправить первую ошибку, сначала откройте файл *MakeConstAnalyzer.cs* и найдите цикл foreach, в котором проверяются инициализаторы локального объявления. Им должны быть назначены значения констант. Сразу же, *до* выполнения цикла foreach, вызовите `context.SemanticModel.GetTypeInfo()`, чтобы извлечь подробные сведения об объявленном типе из локального объявления:

```
TypeSyntax variableTypeName = localDeclaration.Declaration.Type;
ITypeSymbol variableType = context.SemanticModel.GetTypeInfo(variableTypeName,
context.CancellationToken).ConvertedType;
```

Затем проверьте каждый инициализатор внутри цикла `foreach`, чтобы убедиться в том, что его можно преобразовать в тип переменной. Убедившись в том, что инициализатор является константой, добавьте приведенную ниже проверку:

```
// Ensure that the initializer value can be converted to the type of the
// local declaration without a user-defined conversion.
Conversion conversion = context.SemanticModel.ClassifyConversion(initializer.Value, variableType);
if (!conversion.Exists || conversion.IsUserDefined)
{
    return;
}
```

Следующее изменение основано на последнем. Перед закрывающей фигурной скобкой первого цикла foreach добавьте приведенный ниже код. Он проверит тип локального объявления, когда константа является строкой или имеет значение null.

```
// Special cases:
//  * If the constant value is a string, the type of the local declaration
//    must be System.String.
//  * If the constant value is null, the type of the local declaration must
//    be a reference type.
if (constantValue.Value is string)
{
    if (variableType.SpecialType != SpecialType.System_String)
    {
        return;
    }
}
else if (variableType.IsReferenceType && constantValue.Value != null)
{
    return;
}
```

Необходимо заменить ключевое слово `var` правильным именем типа в вашем поставщике исправлений кода. Вернитесь к *MakeConstCodeFixProvider.cs*. Код, который вы добавите, выполняет следующие шаги:

- Проверяет, является ли объявление `var`, если да:
- Создает тип для выводимого типа.
- Проверяет, что объявление типа не является псевдонимом. Если это так, можно объявить `const var`.
- Проверяет, что `var` не является именем типа в программе (если это так, `const var` является допустимым).
- Упрощает полное имя типа.

Кажется, что здесь довольно много кода. Но это не так. Замените строку, которая объявляет и инициализирует `newLocal` приведенным ниже кодом. Он выполняется сразу после инициализации `newModifiers`:

```
// If the type of the declaration is 'var', create a new type name
// for the inferred type.
VariableDeclarationSyntax variableDeclaration = localDeclaration.Declaration;
TypeSyntax variableTypeName = variableDeclaration.Type;
if (variableTypeName.IsVar)
{
    SemanticModel semanticModel = await
document.GetSemanticModelAsync(cancellationToken).ConfigureAwait(false);

    // Special case: Ensure that 'var' isn't actually an alias to another type
    // (e.g. using var = System.String).
    IAliasSymbol aliasInfo = semanticModel.GetAliasInfo(variableTypeName, cancellationToken);
    if (aliasInfo == null)
    {
        // Retrieve the type inferred for var.
        ITypeSymbol type = semanticModel.GetTypeInfo(variableTypeName, cancellationToken).ConvertedType;

        // Special case: Ensure that 'var' isn't actually a type named 'var'.
        if (type.Name != "var")
        {
            // Create a new TypeSyntax for the inferred type. Be careful
            // to keep any leading and trailing trivia from the var keyword.
            TypeSyntax typeName = SyntaxFactory.ParseTypeName(type.ToDisplayString())
                .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())
                .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());

            // Add an annotation to simplify the type name.
            TypeSyntax simplifiedTypeName = typeName.WithAdditionalAnnotations(Simplifier.Annotation);

            // Replace the type in the variable declaration.
            variableDeclaration = variableDeclaration.WithType(simplifiedTypeName);
        }
    }
}
// Produce the new local declaration.
LocalDeclarationStatementSyntax newLocal = trimmedLocal.WithModifiers(newModifiers)
                        .WithDeclaration(variableDeclaration);
```

Чтобы использовать тип Simplifier, потребуется добавить одну директиву `using`:

```
using Microsoft.CodeAnalysis.Simplification;
```

Запустите тесты. Они все должны быть пройдены. Можете поздравить себя, запустив готовый анализатор. Чтобы запустить проект анализатора во втором экземпляре Visual Studio с загруженным расширением Roslyn (предварительная версия), нажмите клавиши CTRL+F5.

- Во втором экземпляре Visual Studio создайте новый проект C# консольного приложения и добавьте `int x = "abc";` в метод main. Так как первая ошибка была исправлена, для объявления локальной переменной не должно выдаваться предупреждение (хотя есть ошибка компилятора, как и предполагалось).
- Затем добавьте `object s = "abc";` в метод main. Так как вторая ошибка была исправлена, не должно быть никаких предупреждений.
- Наконец, добавьте другую локальную переменную, использующую ключевое слово `var`. Внизу слева появится предупреждение и предложение исправлений.
- Наведите курсор редактора на волнистую линию и нажмите клавиши CTRL+. Отобразятся предложенные исправления. Обратите внимание, что после выбора исправления ключевое слово `var` теперь обрабатывается правильно.

В конце добавьте приведенный ниже код:

```
int i = 2;
int j = 32;
int k = i + j;
```

После этого только две переменные будут подчеркнуты красными волнистыми линиями. Добавьте `const` в `i` и `j`. Появится предупреждение, указывающее на то, что `k` может быть `const`.

Поздравляем! Вы создали свое первое расширение .NET Compiler Platform, которое выполняет анализ кода в режиме реального времени, находит проблемы и быстро их исправляет. Кроме того, вы изучили множество API, входящих в пакет SDK .NET Compiler Platform (API Roslyn). Вы можете сравнить результат своей работы с готовым примером в нашем репозитории в GitHub.

Другие источники

- Начало работы с функциями синтаксического анализа
- Начало работы с семантическим анализом

Руководство по программированию на C#

18.12.2021 • 2 minutes to read

Этот раздел содержит подробные сведения о ключевых функциях языка C# и возможностях, которые платформа .NET предоставляет для C#.

В текстах этого раздела обычно предполагается, что вы уже знакомы с языком C# и общими концепциями программирования. Если вы раньше никогда не программировали или не работали с языком C#, рекомендуем посмотреть руководства Знакомство с C# или Руководство в браузере по .NET, которые не требуют знаний программирования.

Сведения о конкретных ключевых словах, операторах и директивах препроцессора см. в справочнике по C#. Сведения о спецификации языка C# размещены в статье Спецификации языка C#.

Разделы о программе

Структура программы C#

Main() и аргументы командной строки

Разделы о языке

Инструкции, выражения и операторы

Типы

Объектно-ориентированное программирование

Интерфейсы

Делегаты

Массивы

Строки

Свойства

Индексаторы

События

Универсальные шаблоны

Итераторы

Выражения запросов LINQ

Пространства имен

Небезопасный код и указатели

Комментарии XML-документации

Разделы о платформе

Домены приложений

Сборки в .NET

Атрибуты

Коллекции

Исключения и обработка исключений

Файловая система и реестр (руководство по программированию на C#)

Взаимодействие

Отражение

См. также

- Справочник по C#

Основные понятия программирования (C#)

18.12.2021 • 2 minutes to read

Этот раздел описывает концепции программирования на языке C#.

В этом разделе

ЗАГОЛОВОК	ОПИСАНИЕ
Сборки в .NET	Описывает создание и использование сборок.
Асинхронное программирование с использованием ключевых слов async и await (C#)	Описывает создание асинхронных решений в C# с помощью ключевых слов async и await. Содержит пошаговое руководство.
Атрибуты (C#)	Описывает, как с помощью атрибутов предоставлять дополнительные сведения об элементах программы, таких как типы, поля, методы и свойства.
Коллекции (C#)	Описывает некоторые типы коллекций, предоставляемых платформой .NET. Демонстрирует использование простых коллекций и коллекций с парами "ключ — значение".
Ковариация и контравариантность (C#)	Демонстрирует, как активировать неявное преобразование параметров универсального типа в интерфейсах и делегатах.
Деревья выражений (C#)	Объясняет, как использовать деревья выражений для динамического изменения исполняемого кода.
Итераторы (C#)	Описывает итераторы, которые используются для пошагового перебора элементов коллекций.
LINQ (C#)	Описывает возможности работы с запросами в синтаксисе языка C#, а также примеры запросов к реляционными базам данных, XML-документам, наборам данных и коллекциям в памяти.
Отражение (C#)	Объясняет, как использовать отражение для динамического создания экземпляра типа, привязки типа к существующему объекту, а также получения типа из существующего объекта и вызова его методов или доступа к его полям и свойствам.
Сериализация (C#)	Описывает основные концепции двоичной сериализации, XML-сериализации и SOAP-сериализации.

Связанные разделы

- Советы по производительности .NET

  Описывает нескольких основных правил, которые помогают повысить производительность

приложения.

Асинхронное программирование с использованием ключевых слов async и await

18.12.2021 • 14 minutes to read

Модель асинхронного программирования на основе задач (TAP) предоставляет абстракцию асинхронного кода. Вы пишете код как последовательность операторов, как обычно. Вы можете читать этот код, как если бы каждая инструкция завершалась до начала следующей. Компилятор выполняет множество преобразований, так как некоторые из этих инструкций могут начать работу и вернуть Task, представляющий текущую работу.

Это и есть цель такого синтаксиса: сделать возможным код, который читается как последовательность операторов, но выполняется в гораздо более сложном порядке на основе выделения внешних ресурсов и при завершении задач. Это аналогично тому, как люди дают инструкции для процессов, которые включают асинхронные задачи. В этой статье вы будете использовать пример инструкции для приготовления завтрака, чтобы увидеть, как ключевые слова `async` и `await` упрощают понимание кода, который включает в себя серию асинхронных инструкций. Можно написать инструкции аналогично следующему списку, чтобы объяснить, как приготовить завтрак.

1. Налить чашку кофе.
2. Нагреть сковородку, а затем поджарить два яйца.
3. Поджарить три куска бекона.
4. Сделать два тоста.
5. Намазать тосты маслом и джемом.
6. Налить стакан апельсинового сока.

Если у вас есть кулинарный опыт, вы бы выполняли эти инструкции **асинхронно**. Сначала вы бы поставили сковородку на огонь, а затем занялись бы беконом. Потом бы поставили тосты, а вслед за этим принялись бы за яичницу. На каждом этапе процесса необходимо запустить задачу, а затем обратить внимание на другие задачи, которые требуют вашего внимания.

Приготовление завтрака представляет собой хороший пример асинхронной непараллельной работы. Один пользователь (или поток) может обрабатывать все эти задачи. Продолжая аналогию с завтраком, один человек может приготовить завтрак асинхронно путем запуска очередной задачи до завершения предыдущей. Готовка продолжается вне зависимости от того, следит ли за ней кто-либо. Как только вы начали греть сковороду для яичницы, можно заняться обжаркой бекона. Когда бекон будет жариться, можно поместить хлеб в тостер.

Для параллельного алгоритма потребовалось бы несколько поваров (или потоков). Один готовит яйца, один — бекон и т. д. Каждый из них будет заниматься только одной задачей. Каждый повар (или поток) будет заблокирован синхронным ожиданием готовности бекона или тостов.

Теперь рассмотрим эти же инструкции, написанные на C#.

```
using System;
using System.Threading.Tasks;

namespace AsyncBreakfast
{
    class Program
    {
        static void Main(string[] args)
        {
```

```
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");

    Egg eggs = FryEggs(2);
    Console.WriteLine("eggs are ready");

    Bacon bacon = FryBacon(3);
    Console.WriteLine("bacon is ready");

    Toast toast = ToastBread(2);
    ApplyButter(toast);
    ApplyJam(toast);
    Console.WriteLine("toast is ready");

    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");
}

private static Juice PourOJ()
{
    Console.WriteLine("Pouring orange juice");
    return new Juice();
}

private static void ApplyJam(Toast toast) =>
    Console.WriteLine("Putting jam on the toast");

private static void ApplyButter(Toast toast) =>
    Console.WriteLine("Putting butter on the toast");

private static Toast ToastBread(int slices)
{
    for (int slice = 0; slice < slices; slice++)
    {
        Console.WriteLine("Putting a slice of bread in the toaster");
    }
    Console.WriteLine("Start toasting...");
    Task.Delay(3000).Wait();
    Console.WriteLine("Remove toast from toaster");

    return new Toast();
}

private static Bacon FryBacon(int slices)
{
    Console.WriteLine($"putting {slices} slices of bacon in the pan");
    Console.WriteLine("cooking first side of bacon...");
    Task.Delay(3000).Wait();
    for (int slice = 0; slice < slices; slice++)
    {
        Console.WriteLine("flipping a slice of bacon");
    }
    Console.WriteLine("cooking the second side of bacon...");
    Task.Delay(3000).Wait();
    Console.WriteLine("Put bacon on plate");

    return new Bacon();
}

private static Egg FryEggs(int howMany)
{
    Console.WriteLine("Warming the egg pan...");
    Task.Delay(3000).Wait();
    Console.WriteLine($"cracking {howMany} eggs");
    Console.WriteLine("cooking the eggs ...");
    Task.Delay(3000).Wait();
    Console.WriteLine("Put eggs on plate");
```

```
            return new Egg();
        }

        private static Coffee PourCoffee()
        {
            Console.WriteLine("Pouring coffee");
            return new Coffee();
        }
    }
}
```

Синхронное приготовление завтрака заняло примерно 30 минут, так как общее время является суммой времен выполнения каждой задачи.

> **NOTE**
> Классы `Coffee`, `Egg`, `Bacon`, `Toast` и `Juice` пусты. Они просто являются классами меток, используемыми в целях демонстрации, не содержат свойств и не используются для выполнения других задач.

Компьютеры не рассматривают эти инструкции так же, как люди. Компьютер будет задерживаться над каждой инструкцией до момента, когда работа будет завершена, прежде чем перейдет к следующему оператору. Вряд ли такой завтрак вас устроит. Более поздние задачи не будут начаты до завершения предыдущих. Потребуется гораздо больше времени для приготовления завтрака, к тому же часть уже остынет еще до подачи.

Если требуется, чтобы компьютер асинхронно выполнил инструкции выше, необходимо писать асинхронный код.

Эти проблемы важны для программ, которые вы пишете уже сегодня. При написании клиентских программ требуется, чтобы пользовательский интерфейс реагировал на ввод данных пользователем. Приложения не должны блокировать телефон при скачивании данных из Интернета. При написании серверных программ не стоит блокировать потоки. Эти потоки могут обслуживать другие запросы. Использование синхронного кода в ситуации, когда существуют асинхронные альтернативы, мешает масштабированию с минимальными затратами. Вы платите за эти заблокированные потоки.

Успешные современные приложения требуют использования асинхронного кода. Без поддержки языком при написании асинхронного кода требуются обратные вызовы, события завершения или другие способы, заслоняющие исходное назначение кода. Преимуществом синхронного кода является то, что эти пошаговые действия упрощают проверку и анализ. Традиционные асинхронные модели заставляют сосредоточиваться на асинхронности кода, а не на фундаментальных действиях в нем.

Не блокировать, а использовать await

Приведенный выше код демонстрирует дурную практику: использование синхронного кода для выполнения асинхронных операций. В таком виде код блокирует выполняющий поток, не позволяя делать другие действия. Он не будет прерван, пока задачи выполняются. Все равно что стоять и смотреть на тостер, пока поджаривается хлеб. Пока тост не готов, вы всех игнорируете.

Давайте начнем менять этот код, чтобы не блокировать поток во время выполнения задачи. Ключевое слово `await` позволяет обойтись без блокировки для запуска задачи, а затем продолжить выполнение, когда задача завершается. Простая асинхронная версия кода для приготовления завтрака будет выглядеть так:

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");

    Egg eggs = await FryEggsAsync(2);
    Console.WriteLine("eggs are ready");

    Bacon bacon = await FryBaconAsync(3);
    Console.WriteLine("bacon is ready");

    Toast toast = await ToastBreadAsync(2);
    ApplyButter(toast);
    ApplyJam(toast);
    Console.WriteLine("toast is ready");

    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");
}
```

IMPORTANT

Общее затраченное время примерно такое же, как у начальной синхронной версии. Этот код можно улучшить, используя ряд ключевых возможностей асинхронного программирования.

TIP

Тексты методов `FryEggsAsync`, `FryBaconAsync` и `ToastBreadAsync` были обновлены так, чтобы возвращать `Task<Egg>`, `Task<Bacon>` и `Task<Toast>`, соответственно. Методы переименованы и теперь содержат суффикс "Async". Их реализации показаны в составе окончательной версии далее в этой статье.

Этот код не блокируется при приготовлении яиц или бекона. Этот код, однако, не запускает других задач. По-прежнему придется поместить тост в тостер и смотреть на него, пока он не выскочит. Но по крайней мере можно отвечать всем, кто хочет вашего внимания. В ресторане, где будет размещаться несколько заказов, повар сможет начать готовить другой завтрак, пока первый готовится.

Теперь поток завтрака не блокируется в ожидании любой запущенной задачи, которая еще не

завершена. Для некоторых приложений это изменение — все, что требуется. Приложение с графическим интерфейсом будет отвечать пользователю после этого изменения. Тем не менее в этом сценарии нам нужно больше. Нам не требуется последовательное выполнение каждой из задач компонента. Лучше запускать каждую из задач компонента, не ожидая завершения предыдущей задачи.

Одновременный запуск задач

Во многих случаях требуется запускать сразу несколько независимых задач. Затем, когда каждая задача завершается, можно продолжить другую работу, которая уже готова к этому. В нашей аналогии — так завтрак готовится быстрее. Вы также приготовите все примерно в одно и то же время. Вы получите горячий завтрак.

System.Threading.Tasks.Task и связанные типы — это классы, позволяющие делать выводы о задачах, которые находятся в процессе выполнения. Это позволяет писать код, который точнее определяет, как будет фактически готовиться завтрак. Вы начинаете готовить яйца, бекон и тосты примерно в одно и то же время. По мере необходимости вы обращаете внимание на отдельные задачи, переходите к другим, а затем ждете третьих, которые нуждаются в обработке.

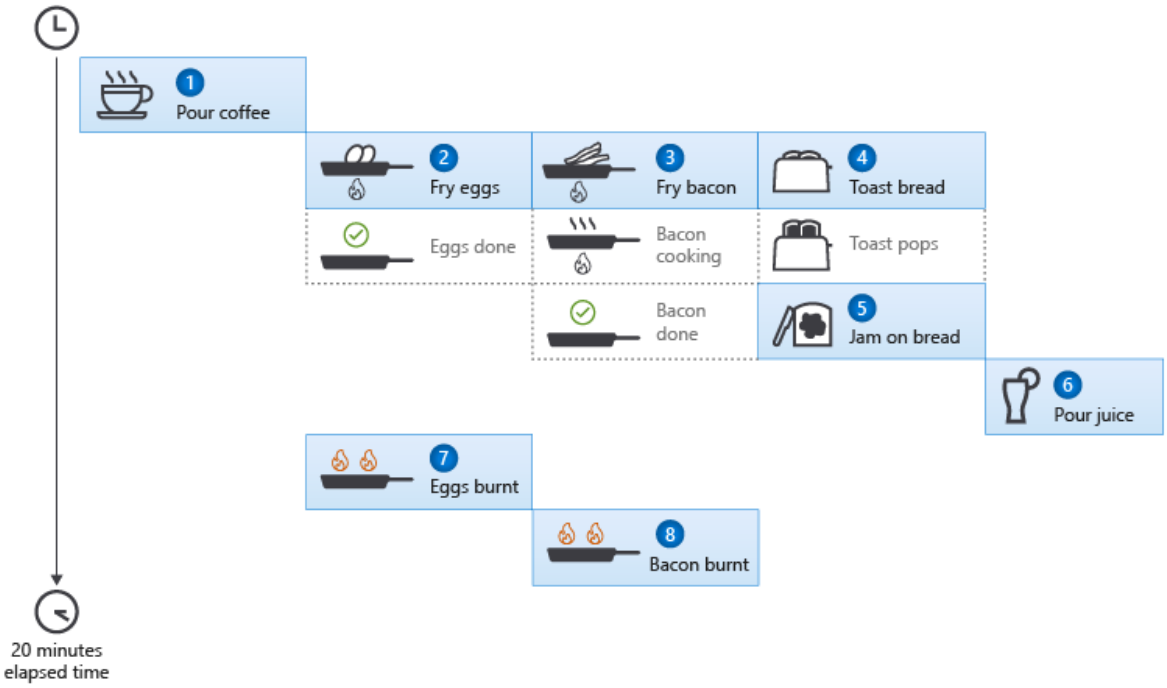
Вы начинаете задачу и удерживаете объект Task, представляющий работу. Вы вызываете `await` для каждой задачи, прежде чем начать работу с ее результатами.

Давайте внесем эти изменения в код для приготовления завтрака. Первым делом сохраним задачи для отдельных операций при их запуске, чтобы не ждать их:

```
Coffee cup = PourCoffee();
Console.WriteLine("Coffee is ready");

Task<Egg> eggsTask = FryEggsAsync(2);
Egg eggs = await eggsTask;
Console.WriteLine("Eggs are ready");

Task<Bacon> baconTask = FryBaconAsync(3);
Bacon bacon = await baconTask;
Console.WriteLine("Bacon is ready");

Task<Toast> toastTask = ToastBreadAsync(2);
Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("Toast is ready");

Juice oj = PourOJ();
Console.WriteLine("Oj is ready");
Console.WriteLine("Breakfast is ready!");
```

Затем вы можете переместить инструкции `await` для бекона и яиц в конец метода, сразу перед подачей завтрака:

```
Coffee cup = PourCoffee();
Console.WriteLine("Coffee is ready");

Task<Egg> eggsTask = FryEggsAsync(2);
Task<Bacon> baconTask = FryBaconAsync(3);
Task<Toast> toastTask = ToastBreadAsync(2);

Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("Toast is ready");
Juice oj = PourOJ();
Console.WriteLine("Oj is ready");

Egg eggs = await eggsTask;
Console.WriteLine("Eggs are ready");
Bacon bacon = await baconTask;
Console.WriteLine("Bacon is ready");

Console.WriteLine("Breakfast is ready!");
```



Асинхронное приготовление завтрака заняло примерно 20 минут. Это позволило сэкономить время, так как некоторые задачи можно было выполнять параллельно.

Предыдущий код работает лучше. Запуск всех асинхронных задач выполняется за один раз. Вы ожидаете каждую задачу только в том случае, когда требуются результаты. Приведенный выше код может быть похож на код в веб-приложении, который отправляет запросы для разных микрослужб, а затем объединяет результаты в одну страницу. Вы отправляете все запросы сразу, а затем вызываете `await`, чтобы соединить все задачи и создать веб-страницу.

Сочетаемость задач

У вас все готово для завтрака в одно и то же время, за исключением тостов. Приготовление тоста — композиция асинхронной операции (поджарить хлеб) и синхронной операции (добавить масло и джем). Обновление этого кода иллюстрирует важную концепцию:

> **IMPORTANT**
> композиция асинхронной операции, за которой следует синхронная задача, является асинхронной операцией. Говоря иначе, если какая-либо часть операции является асинхронной, то и вся операция является асинхронной.

Приведенный выше код показал, что можно использовать объекты Task или Task<TResult> для хранения выполняемых задач. Вы вызываете `await` для каждой задачи, прежде чем использовать ее результат. Следующим шагом является создание методов, которые представляют сочетание другой работы. Перед подачей завтрака требуется дождаться задачи, представляющей поджарку хлеба перед добавлением масла и джема. Вы можете представить эту работу следующим кодом:

```
static async Task<Toast> MakeToastWithButterAndJamAsync(int number)
{
    var toast = await ToastBreadAsync(number);
    ApplyButter(toast);
    ApplyJam(toast);

    return toast;
}
```

Предыдущий метод имеет `async` модификатор в сигнатуре. Он сообщает компилятору, что этот метод содержит инструкцию `await`; она содержит асинхронные операции. Этот метод представляет задачу, в рамках которой поджаривается хлеб, а затем добавляется масло и джем. Этот метод возвращает Task<TResult>, представляющий сочетание этих трех операций. Теперь вид основного блока кода будет таким:

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");

    var eggsTask = FryEggsAsync(2);
    var baconTask = FryBaconAsync(3);
    var toastTask = MakeToastWithButterAndJamAsync(2);

    var eggs = await eggsTask;
    Console.WriteLine("eggs are ready");

    var bacon = await baconTask;
    Console.WriteLine("bacon is ready");

    var toast = await toastTask;
    Console.WriteLine("toast is ready");

    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");
}
```

Предыдущее изменение показывает важную методику для работы с асинхронным кодом. Составные задачи можно создавать, разделяя операции в новом методе, который возвращает задачу. Вы можете выбрать, когда следует ожидать выполнения созданной задачи. Одновременно можно запускать другие задачи.

Асинхронные исключения

До этого момента вы неявно предполагали, что все эти задачи были выполнены успешно. Асинхронные

методы создают исключения, точно так же, как и синхронные методы. В целом поддержка исключений и обработки ошибок в асинхронном коде преследует те же цели, что и поддержка асинхронного кода в целом: необходимо написать код, который выглядит как последовательность синхронных инструкций. Когда задачи не могут быть успешно завершены, они выдают исключения. Клиентский код может перехватывать эти исключения, когда запущенная задача ожидается (`awaited`). Например, предположим, что тостер загорается во время приготовления тоста. Это можно смоделировать, изменив метод `ToastBreadAsync` следующим образом:

```
private static async Task<Toast> ToastBreadAsync(int slices)
{
    for (int slice = 0; slice < slices; slice++)
    {
        Console.WriteLine("Putting a slice of bread in the toaster");
    }
    Console.WriteLine("Start toasting...");
    await Task.Delay(2000);
    Console.WriteLine("Fire! Toast is ruined!");
    throw new InvalidOperationException("The toaster is on fire");
    await Task.Delay(1000);
    Console.WriteLine("Remove toast from toaster");

    return new Toast();
}
```

> **NOTE**
> При компиляции предыдущего кода будет выдано предупреждение о наличии недостижимого кода. Это сделано намеренно, поскольку после того, как тостер загорится, дальнейшие операции не будут выполняться обычным образом.

Запустите приложение после внесения этих изменений, и вы получите следующий результат:

```
Pouring coffee
Coffee is ready
Warming the egg pan...
putting 3 slices of bacon in the pan
Cooking first side of bacon...
Putting a slice of bread in the toaster
Putting a slice of bread in the toaster
Start toasting...
Fire! Toast is ruined!
Flipping a slice of bacon
Flipping a slice of bacon
Flipping a slice of bacon
Cooking the second side of bacon...
Cracking 2 eggs
Cooking the eggs ...
Put bacon on plate
Put eggs on plate
Eggs are ready
Bacon is ready
Unhandled exception. System.InvalidOperationException: The toaster is on fire
   at AsyncBreakfast.Program.ToastBreadAsync(Int32 slices) in Program.cs:line 65
   at AsyncBreakfast.Program.MakeToastWithButterAndJamAsync(Int32 number) in Program.cs:line 36
   at AsyncBreakfast.Program.Main(String[] args) in Program.cs:line 24
   at AsyncBreakfast.Program.<Main>(String[] args)
```

Обратите внимание, что между возгоранием тостера и выдачей исключения выполняется довольно много задач. Если задача, которая выполняется асинхронно, выдает исключение, эта задача завершается ***с ошибкой***. Созданное исключение находится в свойстве Task.Exception объекта Task. Задачи,

завершившиеся с ошибкой, выдают исключение, когда эти задачи ожидаются.

Существует два важных механизма, работу которых нужно понимать: как исключение хранится в задаче, завершившейся с ошибкой, и как оно распаковывается и выдается повторно, когда код ожидает задачу, завершившуюся с ошибкой.

Когда асинхронный код выдает исключение, это исключение хранится в `Task`. Свойство Task.Exception имеет значение System.AggregateException, потому что во время асинхронного выполнения может быть выдано несколько исключений. Все выданные исключения добавляются в коллекцию AggregateException.InnerExceptions. Если это свойство `Exception` имеет значение null, то создается новое исключение `AggregateException`, и выданное исключение становится первым элементом в коллекции.

Чаще всего для задачи, завершившейся с ошибкой, свойство `Exception` содержит только одно исключение. Когда код ожидает (`awaits`) задачу, завершившуюся сбоем, первое исключение в коллекции AggregateException.InnerExceptions выдается повторно. Поэтому в выходных данных этого примера мы видим `InvalidOperationException` вместо `AggregateException`. Извлечение первого внутреннего исключения делает работу с асинхронными методами настолько похожей на работу с синхронными методами, насколько это возможно. Если ваш сценарий может создать несколько исключений, вы можете проверить свойство `Exception` в коде.

Перед тем как продолжить, закомментируйте эти две строки в методе `ToastBreadAsync`. Ведь вы не хотите, чтобы у вас появилась еще одна проблема:

```
Console.WriteLine("Fire! Toast is ruined!");
throw new InvalidOperationException("The toaster is on fire");
```

Эффективное ожидание задач

Ряд инструкций `await` в конце приведенного выше кода можно улучшить с помощью методов класса `Task`. Один из этих API — WhenAll, который возвращает Task; она завершается после завершения всех задач в списке аргументов, как показано в следующем коде:

```
await Task.WhenAll(eggsTask, baconTask, toastTask);
Console.WriteLine("Eggs are ready");
Console.WriteLine("Bacon is ready");
Console.WriteLine("Toast is ready");
Console.WriteLine("Breakfast is ready!");
```

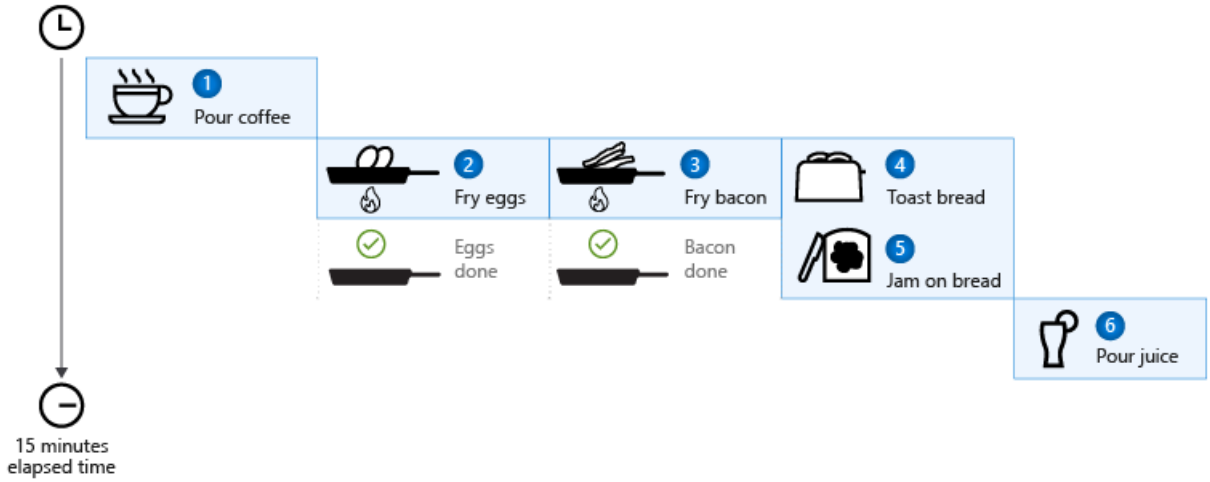
Другой вариант — использовать WhenAny, который возвращает `Task<Task>`, выполняемый по завершении любого из своих аргументов. Можно ожидать возвращенной задачи, зная, что она уже завершена. В следующем коде показано, как использовать WhenAny для ожидания первой задачи, чтобы затем обработать ее результат. После обработки результата завершенной задачи удалим ее из списка задач, передаваемого в `WhenAny`.

```
var breakfastTasks = new List<Task> { eggsTask, baconTask, toastTask };
while (breakfastTasks.Count > 0)
{
    Task finishedTask = await Task.WhenAny(breakfastTasks);
    if (finishedTask == eggsTask)
    {
        Console.WriteLine("Eggs are ready");
    }
    else if (finishedTask == baconTask)
    {
        Console.WriteLine("Bacon is ready");
    }
    else if (finishedTask == toastTask)
    {
        Console.WriteLine("Toast is ready");
    }
    breakfastTasks.Remove(finishedTask);
}
```

После всех этих изменений окончательная версия кода выглядит так:

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace AsyncBreakfast
{
    class Program
    {
        static async Task Main(string[] args)
        {
            Coffee cup = PourCoffee();
            Console.WriteLine("coffee is ready");

            var eggsTask = FryEggsAsync(2);
            var baconTask = FryBaconAsync(3);
            var toastTask = MakeToastWithButterAndJamAsync(2);

            var breakfastTasks = new List<Task> { eggsTask, baconTask, toastTask };
            while (breakfastTasks.Count > 0)
            {
                Task finishedTask = await Task.WhenAny(breakfastTasks);
                if (finishedTask == eggsTask)
                {
                    Console.WriteLine("eggs are ready");
                }
                else if (finishedTask == baconTask)
                {
                    Console.WriteLine("bacon is ready");
                }
                else if (finishedTask == toastTask)
                {
                    Console.WriteLine("toast is ready");
                }
                breakfastTasks.Remove(finishedTask);
            }

            Juice oj = PourOJ();
            Console.WriteLine("oj is ready");
            Console.WriteLine("Breakfast is ready!");
        }

        static async Task<Toast> MakeToastWithButterAndJamAsync(int number)
        {
            var toast = await ToastBreadAsync(number);
            ApplyButter(toast);
```

```
            ApplyJam(toast);

            return toast;
        }

        private static Juice PourOJ()
        {
            Console.WriteLine("Pouring orange juice");
            return new Juice();
        }

        private static void ApplyJam(Toast toast) =>
            Console.WriteLine("Putting jam on the toast");

        private static void ApplyButter(Toast toast) =>
            Console.WriteLine("Putting butter on the toast");

        private static async Task<Toast> ToastBreadAsync(int slices)
        {
            for (int slice = 0; slice < slices; slice++)
            {
                Console.WriteLine("Putting a slice of bread in the toaster");
            }
            Console.WriteLine("Start toasting...");
            await Task.Delay(3000);
            Console.WriteLine("Remove toast from toaster");

            return new Toast();
        }

        private static async Task<Bacon> FryBaconAsync(int slices)
        {
            Console.WriteLine($"putting {slices} slices of bacon in the pan");
            Console.WriteLine("cooking first side of bacon...");
            await Task.Delay(3000);
            for (int slice = 0; slice < slices; slice++)
            {
                Console.WriteLine("flipping a slice of bacon");
            }
            Console.WriteLine("cooking the second side of bacon...");
            await Task.Delay(3000);
            Console.WriteLine("Put bacon on plate");

            return new Bacon();
        }

        private static async Task<Egg> FryEggsAsync(int howMany)
        {
            Console.WriteLine("Warming the egg pan...");
            await Task.Delay(3000);
            Console.WriteLine($"cracking {howMany} eggs");
            Console.WriteLine("cooking the eggs ...");
            await Task.Delay(3000);
            Console.WriteLine("Put eggs on plate");

            return new Egg();
        }

        private static Coffee PourCoffee()
        {
            Console.WriteLine("Pouring coffee");
            return new Coffee();
        }
    }
}
```



Окончательная версия асинхронного приготовления завтрака заняла примерно 15 минут, так как в этом случае некоторые задачи выполнялись параллельно. Также одновременно отслеживалось несколько задач из кода, и действия выполнялись только тогда, когда это было необходимо.

Этот итоговый код выполняется асинхронно. Он более точно отражает, как пользователь будет готовить завтрак. Сравните предыдущий код с первым примером кода в этой статье. Основные действия по-прежнему очевидны при прочтении. Этот код можно прочитать так же, как указания по приготовлению завтрака в начале этой статьи. Возможности языка для `async` и `await` делают возможными преобразования, которые любой человек производит, выполняя эти инструкции: запуск задач без блокировки в ожидании их завершения.

Следующие шаги

Исследование реальных сценариев для асинхронных программ

Асинхронное программирование

18.12.2021 • 9 minutes to read

Для решения задач, связанных с вводом-выводом (например, запрос данных из сети, доступ к базе данных или чтение и запись в файловой системе), желательно использовать асинхронное программирование. Если у вас есть код, ограниченный ресурсами процессора, например выполняющий сложные вычисления, то это также подходящий сценарий для асинхронного программирования.

В C# есть модель асинхронного программирования, реализованная на уровне языка, которая позволяет легко писать асинхронный код, не прибегая к обратным вызовам или библиотекам, которые поддерживают асинхронность. Она строится на принципах асинхронной модели на основе задач (TAP).

Обзор асинхронной модели

В основе асинхронного программирования лежат объекты `Task` и `Task<T>`, которые моделируют асинхронные операции. Они поддерживаются ключевыми словами `async` и `await`. В большинстве случаев модель достаточно проста.

- В коде, ограниченном производительностью ввода-вывода, выполняйте await для операции, которая возвращает `Task` или `Task<T>`, внутри метода `async`.
- В коде, ограниченном ресурсами процессора, выполняйте await для операции, которая запускается в фоновом потоке методом Task.Run.

Именно с помощью ключевого слова `await` творится вся магия. Оно передает управление вызывающему объекту метода, который выполнил `await`, позволяя, таким образом, пользовательскому интерфейсу или службе отвечать на запросы. Хотя существуют и другие способы реализации асинхронного кода, кроме `async` и `await`, в этой статье рассматриваются только конструкции уровня языка.

Пример кода, ограниченного производительностью ввода-вывода: скачивание данных из веб-службы

Предположим, вам нужно скачать некоторые данные из веб-службы по нажатию кнопки, не блокируя поток пользовательского интерфейса. Это можно сделать так:

```
private readonly HttpClient _httpClient = new HttpClient();

downloadButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI as the request
    // from the web service is happening.
    //
    // The UI thread is now free to perform other work.
    var stringData = await _httpClient.GetStringAsync(URL);
    DoSomethingWithData(stringData);
};
```

В этом коде намерение (скачивание данных в асинхронном режиме) выражается без запутанных операций с объектами `Task`.

Пример кода, ограниченного ресурсами процессора: выполнение вычислений для игры

Предположим, вы разрабатываете игру для мобильных устройств, в которой при нажатии кнопки может наноситься урон множеству противников на экране. Расчет урона может потреблять много ресурсов. Если производить его в потоке пользовательского интерфейса, то на это время игра может приостанавливаться!

Оптимальный способ — запустить фоновый поток, который выполняет задачу с помощью `Task.Run`, а затем ожидать ее результат с помощью `await`. Это обеспечит плавность работы пользовательского интерфейса в процессе вычисления.

```
private DamageResult CalculateDamageDone()
{
    // Code omitted:
    //
    // Does an expensive calculation and returns
    // the result of that calculation.
}

calculateButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI while CalculateDamageDone()
    // performs its work. The UI thread is free to perform other work.
    var damageResult = await Task.Run(() => CalculateDamageDone());
    DisplayDamage(damageResult);
};
```

В этом коде четко выражается назначение события нажатия кнопки, фоновым потоком не требуется управлять вручную, и он выполняется без блокировки.

Что происходит на внутреннем уровне

Выполнение асинхронных операций связано со множеством моментов. Если вы хотите знать внутренние принципы работы объектов `Task` и `Task<T>`, прочтите статью Подробный обзор асинхронного программирования.

С точки зрения C#, компилятор преобразовывает код в конечный автомат, который контролирует такие моменты, как передача выполнения при достижении `await` и возобновление выполнения после завершения фонового задания.

Если вас интересует теория, это реализация модели асинхронности на основе обещаний.

Ключевые моменты для понимания

- Асинхронный код можно использовать как при ограниченной производительности ввода-вывода, так и при ограниченных ресурсах процессора, но по-разному в каждом случае.
- В асинхронном коде используются конструкции `Task<T>` и `Task`, которые служат для моделирования задач, выполняемых в фоновом режиме.
- Ключевое слово `async` делает метод асинхронным, что позволяет использовать в его теле ключевое слово `await`.
- Когда применяется ключевое слово `await`, оно приостанавливает выполнение вызывающего метода и передает управление обратно вызывающему объекту, пока не будет завершена ожидаемая задача.
- `await` можно использовать только внутри асинхронного метода.

Различия задач, ограниченных ресурсами процессора и производительностью ввода-вывода

В первых двух примерах этого руководства было показано, как можно использовать `async` и `await` для выполнения задач, ограниченных производительностью ввода-вывода и ресурсами процессора. Крайне важно уметь идентифицировать такие задачи, так как они могут существенно повлиять на производительность кода и привести к неправильному использованию некоторых конструкций.

Перед написанием любого кода нужно ответить на два вопроса.

1. Будет ли код "ожидать" чего-либо, например данных из базы данных?

   Если ответ утвердительный, то ваша задача **ограничена производительностью ввода-вывода**.

2. Будет ли код выполнять сложные вычисления?

   Если ответ утвердительный, то задача **ограничена ресурсами процессора**.

Если ваша задача **ограничена производительностью ввода-вывода**, используйте `async` и `await` *без* `Task.Run`. Библиотеку параллельных задач использовать *не следует*. Причина этого указана в статье Подробный обзор асинхронного программирования.

Если ваша задача **ограничена ресурсами процессора** и вам важна скорость реагирования, используйте `async` и `await`, но перенесите выполнение задачи в дополнительный поток, у которого *есть* `Task.Run`. Если к задаче применим параллелизм, рассмотрите возможность использования библиотеки параллельных задач.

Кроме того, всегда следует оценивать выполнение кода. Например, затраты на выполнение задачи, ограниченной ресурсами процессора, могут оказаться не столь высокими, как накладные расходы, связанные с переключениями контекста при многопоточности. Каждый вариант имеет свои недостатки, поэтому следует выбрать наиболее компромиссный вариант в вашей ситуации.

Дополнительные примеры

В приведенных ниже примерах демонстрируются различные способы написания асинхронного кода на C#. Они охватывают несколько сценариев, с которыми вы можете столкнуться.

Извлечение данных из сети

Этот фрагмент кода скачивает код HTML с главной страницы сайта https://dotnetfoundation.org и подсчитывает в нем число вхождений строки ".NET". С помощью ASP.NET он определяет метод контроллера веб-API, который выполняет эту задачу и возвращает число.

> **NOTE**
> Если вы планируете проанализировать HTML в рабочем коде, не используйте регулярные выражения. Используйте библиотеку анализа.

```
private readonly HttpClient _httpClient = new HttpClient();

[HttpGet, Route("DotNetCount")]
public async Task<int> GetDotNetCount()
{
    // Suspends GetDotNetCount() to allow the caller (the web server)
    // to accept another request, rather than blocking on this one.
    var html = await _httpClient.GetStringAsync("https://dotnetfoundation.org");

    return Regex.Matches(html, @"\.NET").Count;
}
```

Вот аналогичный код для универсального приложения для Windows, который выполняет ту же задачу при нажатии кнопки:

```
private readonly HttpClient _httpClient = new HttpClient();

private async void OnSeeTheDotNetsButtonClick(object sender, RoutedEventArgs e)
{
    // Capture the task handle here so we can await the background task later.
    var getDotNetFoundationHtmlTask = _httpClient.GetStringAsync("https://dotnetfoundation.org");

    // Any other work on the UI thread can be done here, such as enabling a Progress Bar.
    // This is important to do here, before the "await" call, so that the user
    // sees the progress bar before execution of this method is yielded.
    NetworkProgressBar.IsEnabled = true;
    NetworkProgressBar.Visibility = Visibility.Visible;

    // The await operator suspends OnSeeTheDotNetsButtonClick(), returning control to its caller.
    // This is what allows the app to be responsive and not block the UI thread.
    var html = await getDotNetFoundationHtmlTask;
    int count = Regex.Matches(html, @"\.NET").Count;

    DotNetCountLabel.Text = $"Number of .NETs on dotnetfoundation.org: {count}";

    NetworkProgressBar.IsEnabled = false;
    NetworkProgressBar.Visibility = Visibility.Collapsed;
}
```

Ожидание выполнения нескольких задач

Может возникнуть ситуация, когда несколько фрагментов данных должны извлекаться одновременно. API-интерфейс `Task` содержит два метода, Task.WhenAll и Task.WhenAny, которые позволяют писать асинхронный код, выполняющий неблокирующее ожидание нескольких фоновых заданий.

В этом примере показано, как можно получить данные `User` для набора `userId`.

```
public async Task<User> GetUserAsync(int userId)
{
    // Code omitted:
    //
    // Given a user Id {userId}, retrieves a User object corresponding
    // to the entry in the database with {userId} as its Id.
}

public static async Task<IEnumerable<User>> GetUsersAsync(IEnumerable<int> userIds)
{
    var getUserTasks = new List<Task<User>>();
    foreach (int userId in userIds)
    {
        getUserTasks.Add(GetUserAsync(userId));
    }

    return await Task.WhenAll(getUserTasks);
}
```

Вот более лаконичный вариант этого кода, написанный с использованием LINQ:

```
public async Task<User> GetUserAsync(int userId)
{
    // Code omitted:
    //
    // Given a user Id {userId}, retrieves a User object corresponding
    // to the entry in the database with {userId} as its Id.
}

public static async Task<User[]> GetUsersAsync(IEnumerable<int> userIds)
{
    var getUserTasks = userIds.Select(id => GetUserAsync(id));
    return await Task.WhenAll(getUserTasks);
}
```

Хотя размер кода будет меньше, соблюдайте осторожность при использовании LINQ в сочетании с асинхронным кодом. Так как в LINQ используется отложенное выполнение, асинхронные вызовы будут выполняться не немедленно, как в цикле `foreach`, если только вы не производите принудительную итерацию созданной последовательности с помощью вызова `.ToList()` или `.ToArray()`.

Важные сведения и советы

При создании асинхронного кода необходимо учитывать ряд моментов, которые позволят избежать непредвиденного поведения.

- **В методах `async` должно присутствовать ключевое слово `await`. В противном случае результат не будет получен.**

  Это важно помнить. Если в теле метода `async` не используется ключевое слово `await`, компилятор C# выдаст предупреждение, но код скомпилируется и будет выполняться, как обычный метод. Это крайне неэффективно, так как созданный компилятором C# конечный автомат для асинхронного метода не будет выполнять никакой работы.

- **К имени каждого создаваемого асинхронного метода следует добавлять суффикс** Async.

  Это соглашение применяется в .NET для удобной дифференциации синхронных и асинхронных методов. Это не всегда применимо к некоторым методам, которые не вызываются в коде явным образом (например, к обработчикам событий или методам веб-контроллеров). Так как они не вызываются в коде явно, требования к их именованию не так строги.

- `async void` **следует использовать только для обработчиков событий.**

  `async void` — это единственный способ обеспечить работу асинхронных обработчиков событий, так как у событий нет типов возвращаемых значений (поэтому они не могут использовать `Task` и `Task<T>`). Любые иные способы применения `async void` не предусмотрены моделью TAP и могут создавать указанные ниже проблемы.

  - Исключения, вызываемые в методе `async void`, невозможно перехватывать вне этого метода.
  - Методы `async void` очень трудно тестировать.
  - Методы `async void` могут иметь негативные побочные эффекты, если вызывающий объект не ожидает, что они будут асинхронными.

- **Будьте осторожны при использовании асинхронных лямбда-выражений в выражениях** LINQ

  Для лямбда-выражений в LINQ применяется отложенное выполнение. Это означает, что код может выполняться в произвольный момент, когда вы этого не ожидаете. Неправильное использование блокирующих задач при этом может привести к взаимоблокировке. Кроме того,

вложение такого асинхронного кода может усложнить анализ выполнения кода. Асинхронное выполнение и LINQ — эффективные средства, но использовать их следует с максимальной осторожностью и ясным пониманием того, что вы делаете.

- **При написании кода ожидание задач следует реализовывать без блокирования**

  Блокирование текущего потока для ожидания завершения `Task` может привести к взаимоблокировкам и блокированию потоков контекста, что потребует более сложной обработки ошибок. В приведенной ниже таблице даются рекомендации по реализации ожидания задач без блокировки.

РЕКОМЕНДУЕМЫЙ СПОСОБ	НЕРЕКОМЕНДУЕМЫЙ СПОСОБ	ЗАДАЧА
`await`	`Task.Wait` или `Task.Result`	Получение результата фоновой задачи
`await Task.WhenAny`	`Task.WaitAny`	Ожидание завершения выполнения любой задачи
`await Task.WhenAll`	`Task.WaitAll`	Ожидание завершения выполнения всех задач
`await Task.Delay`	`Thread.Sleep`	Ожидание в течение заданного времени

- **Рекомендуется использовать `ValueTask` во всех возможных случаях**

  В некоторых случаях возврат объекта `Task` из асинхронных методов может вызывать сложности. `Task` — это тип ссылки, поэтому его применение означает распределение объекта. В случаях, когда метод с модификатором `async` возвращает кэшированный результат или завершается синхронно, лишние распределения могут вызывать серьезные потери времени при выполнении фрагментов кода, зависящих от производительности. Эта проблема встает серьезно, если распределения происходят в коротких циклах. Дополнительные сведения см. в разделе Обобщенные асинхронные типы возвращаемых значений.

- **Рекомендуется использовать `ConfigureAwait(false)`**

  Часто возникает вопрос: "Когда же нужно использовать метод Task.ConfigureAwait(Boolean)?" Этот метод позволяет экземпляру `Task` настроить ожидающий объект. Это важный элемент, неправильная настройка которого может привести к снижению производительности и даже к взаимоблокировкам. Дополнительные сведения о `ConfigureAwait` см. в статье с вопросами и ответами по ConfigureAwait.

- **Пишите код с менее строгим отслеживанием состояния**

  Старайтесь, чтобы выполнение кода не зависело от состояния глобальных объектов или выполнения определенных методов. Оно должно зависеть только от возвращаемых методами значений. Почему?

  - Код будет проще анализировать.
  - Код будет проще тестировать.
  - Гораздо проще будет сочетать асинхронный и синхронный код.
  - Как правило, можно полностью избежать состояний гонки.
  - Зависимость от возвращаемых значений упрощает согласование асинхронного кода.
  - Дополнительным преимуществом является то, что такой код хорошо работает с внедрением зависимостей.

Следует стремиться к достижению полной или почти полной ссылочной прозрачности в коде. Результатом будет предсказуемость базы кода, а также ее пригодность для тестирования и обслуживания.

Другие источники

- В статье Подробный обзор асинхронного программирования приводятся дополнительные сведения о принципах работы задач.
- Асинхронная модель программирования задач (C#).
- Эпизод Six Essential Tips for Async (Шесть важных советов по асинхронному программированию) с Лусианом Вышиком (Lucian Wischik) — это отличный ресурс по асинхронному программированию.

Асинхронная модель программирования

18.12.2021 • 11 minutes to read

Асинхронное программирование позволяет избежать появления узких мест производительности и увеличить общую скорость реагирования приложения. Однако традиционные методы создания асинхронных приложений могут оказаться сложными, как в плане написания кода, так и в плане отладки и обслуживания.

C# 5 вводит упрощенный подход асинхронного программирования, который использует асинхронные возможности .NET Framework 4.5 и более поздних версий, .NET Core и среды выполнения Windows. Компилятор выполняет сложную работу, которую раньше делал разработчик, при этом логическая структура кода приложения похожа на синхронный код. То есть можно пользоваться всеми преимуществами асинхронного программирования с гораздо меньшими, чем обычно, трудозатратами.

В этом разделе рассматриваются области и методы использования асинхронного программирования, а также приводятся ссылки на вспомогательные разделы с дополнительными сведениями и примерами.

Async повышает скорость реагирования

Асинхронность необходимо использовать при наличии потенциально блокирующих работу действий, например при осуществлении доступа к Интернету. Доступ к веб-ресурсу иногда осуществляется медленно или с задержкой. Если такое действие блокируется в синхронном процессе, все приложение вынуждено ожидать. В асинхронном процессе приложение может перейти к следующей операции, не зависящей от веб-ресурса, до завершения блокирующей задачи.

В следующей таблице показаны стандартные области, в которых асинхронное программирование повышает скорость реагирования. Перечисленные API-интерфейсы платформы .NET и среды выполнения Windows содержат методы, поддерживающие асинхронное программирование.

ОБЛАСТЬ ПРИЛОЖЕНИЯ	ТИПЫ .NET С МЕТОДАМИ ASYNC	ТИПЫ СРЕДЫ ВЫПОЛНЕНИЯ WINDOWS С МЕТОДАМИ ASYNC
Веб-доступ	HttpClient	Windows.Web.Http.HttpClient SyndicationClient
Работа с файлами	JsonSerializer StreamReader StreamWriter XmlReader XmlWriter	StorageFile
Работа с образами		MediaCapture BitmapEncoder BitmapDecoder
Программирование с использованием WCF	Синхронные и асинхронные операции	

Асинхронность особенно полезна в приложениях, которые обращаются к потоку пользовательского интерфейса, поскольку все связанные с пользовательским интерфейсом действия обычно используют один поток. В синхронном приложении блокировка одного процесса приводит к блокировке всех процессов. Приложение перестает отвечать, и это выглядит как сбой, а не как ожидание.

При использовании асинхронных методов приложение продолжает реагировать на действия в пользовательском интерфейсе. Например, можно изменить размер окна или свернуть его, либо закрыть приложение, если не требуется ждать завершения работы.

Асинхронный подход добавляет эквивалент автоматической передачи в список параметров, выбранных при создании асинхронных операций. То есть разработчик может пользоваться всеми преимуществами традиционного асинхронного программирования, прикладывая гораздо меньше усилий.

Методы async проще создавать

В C# основой асинхронного программирования. являются ключевые слова async и await. Они позволяют использовать ресурсы платформы .NET Framework, .NET Core или среды выполнения Windows для создания асинхронных методов, и это почти так же просто, как создавать синхронные методы. Асинхронные методы, которые определяются с помощью ключевого слова `async`, называются *методами async*.

Ниже приводится пример асинхронного метода. Почти все элементы кода должны быть вам знакомы.

Полный пример Windows Presentation Foundation (WPF), доступный для загрузки, см. в статье Асинхронное программирование с использованием ключевых слов async и await в C#.

```
public async Task<int> GetUrlContentLengthAsync()
{
    var client = new HttpClient();

    Task<string> getStringTask =
        client.GetStringAsync("https://docs.microsoft.com/dotnet");

    DoIndependentWork();

    string contents = await getStringTask;

    return contents.Length;
}

void DoIndependentWork()
{
    Console.WriteLine("Working...");
}
```

Из предыдущего примера вы можете узнать несколько полезных методик. Начните с сигнатуры метода. Она включает модификатор `async`. Типом возвращаемого значения является `Task<int>` (дополнительные параметры см. в разделе "Типы возвращаемых значений"). Имя метода заканчивается на `Async`. В теле метода `GetStringAsync` возвращает `Task<string>`. Это означает, что когда вы ожидаете (`await`) задачу, то получаете `string` (`contents`). Прежде чем ожидать задачу, можно сделать работу, которая не зависит от `string` из `GetStringAsync`.

Обратите внимание на оператор `await`. Он приостанавливает `GetUrlContentLengthAsync`.

- `GetUrlContentLengthAsync` не может продолжить выполнение до завершения `getStringTask`.
- На это время управление возвращается вызывающему объекту метода `GetUrlContentLengthAsync`.
- Управление возобновляется после завершения `getStringTask`.
- Оператор `await` извлекает результат `string` из `getStringTask`.

Оператор return указывает целочисленный результат. Все методы, ожидающие `GetUrlContentLengthAsync`, получают значение длины.

Если метод `GetUrlContentLengthAsync` не выполняет никакие операции между вызовом метода

`GetStringAsync` и его завершением, можно упростить код, описав вызов и ожидание с помощью следующего простого оператора.

```
string contents = await client.GetStringAsync("https://docs.microsoft.com/dotnet");
```

Далее поясняется, почему код предыдущего примера является асинхронным методом:

- Сигнатура метода включает модификатор `async`.
- Имя асинхронного метода, как правило, оканчивается суффиксом Async.
- Возвращаемое значение имеет один из следующих типов:
  - Task<TResult>, если метод включает оператор return с операндом типа `TResult`.
  - Task, если метод не имеет оператора Return или имеет оператор Return без операнда.
  - `void`, если вы создаете асинхронный обработчик событий.
  - Любой другой тип, имеющий метод `GetAwaiter` (начиная с C# 7.0).

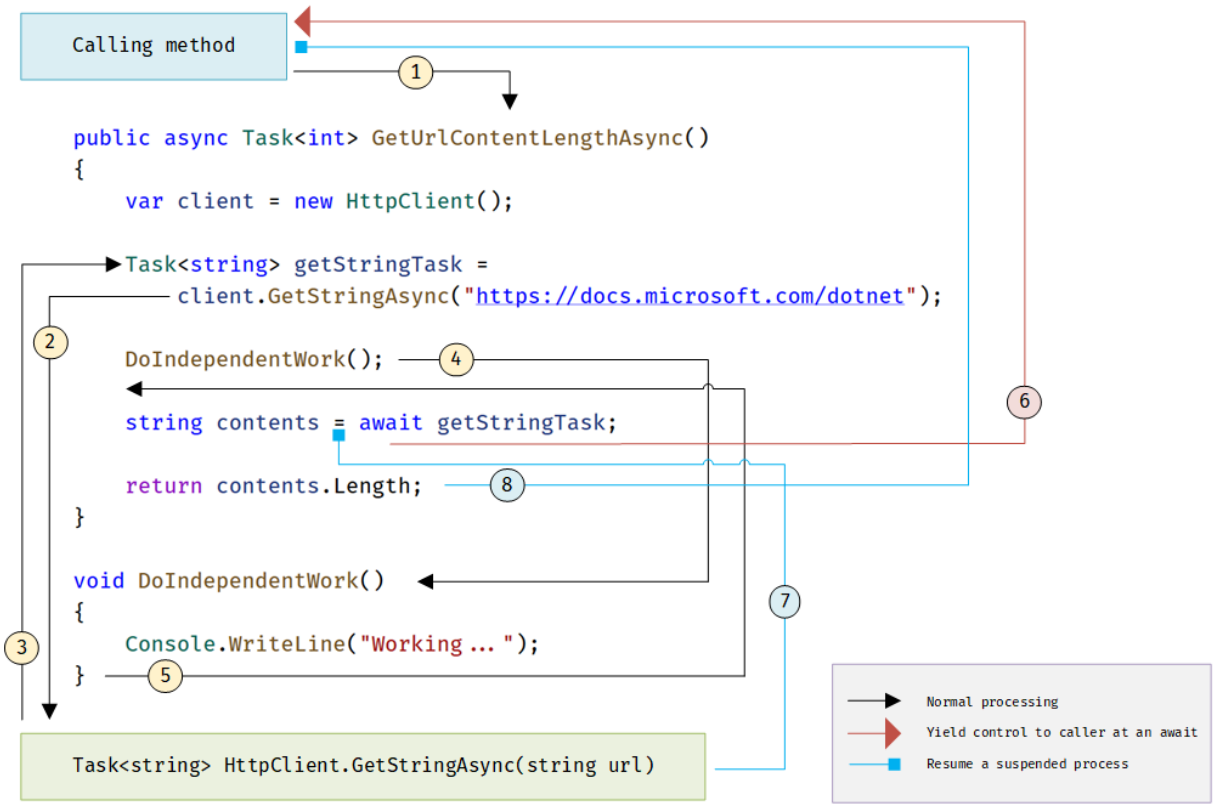
  Дополнительные сведения см. в описании типов возвращаемого значения и параметров.
- Метод обычно содержит по меньшей мере одно выражение `await`, отмечающее точку, в которой метод не может продолжить выполнение до завершения асинхронной операции. На это время метод приостанавливается и управление возвращается к вызывающему объекту метода. В следующем подразделе показано, что происходит в точке приостановки.

В асинхронных методах с помощью соответствующих ключевых слов и типов указывается, что требуется сделать, а компилятор выполняет остальные задачи, в том числе отслеживает обязательные действия, когда управление возвращается в точку await в приостановленном методе. Некоторые программные процессы, такие как циклы и обработка исключений, сложно добавлять в традиционный асинхронный код. В асинхронном методе эти элементы пишутся почти так же, как в синхронном решении, что очень удобно.

Дополнительные сведения об асинхронности в предыдущих версиях платформы .NET Framework см. в статье Библиотека параллельных задач и традиционное асинхронное программирование .NET Framework.

Что происходит в методе Async

В асинхронном программировании важнее всего понимать, как поток управления перемещается из метода в метод. Следующая схема описывает этот процесс.



Числа на схеме соответствуют следующим шагам, которые запускаются, когда вызывающий метод вызывает асинхронный метод.

1. Вызывающий метод вызывает и ожидает асинхронный метод `GetUrlContentLengthAsync`.

2. `GetUrlContentLengthAsync` создает экземпляр HttpClient и вызывает асинхронный метод GetStringAsync, чтобы загрузить содержимое веб-сайта в виде строки.

3. В `GetStringAsync` происходит событие, которое приостанавливает ход выполнения. Например, методу необходимо подождать завершения загрузки или произошло другое блокирующее действие. Чтобы избежать блокировки ресурсов, `GetStringAsync` передает управление вызывающему объекту `GetUrlContentLengthAsync`.

   `GetStringAsync` возвращает Task<TResult>, где `TResult` — строка, а `GetUrlContentLengthAsync` присваивает задачу переменной `getStringTask`. Задача представляет собой непрерывный процесс для вызова `GetStringAsync` с обязательством создать фактическое значение строки, когда работа будет завершена.

4. Поскольку значение из процесса `getStringTask` еще не получено, метод `GetUrlContentLengthAsync` может перейти к другим операциям, не зависящим от конечного результата `GetStringAsync`. Эти операции представлены вызовом синхронного метода `DoIndependentWork`.

5. `DoIndependentWork` — это синхронный метод, который выполняет свой код и возвращает управление вызывающему объекту.

6. Метод `GetUrlContentLengthAsync` выполнил все операции, для которых не требуется результат процесса `getStringTask`. Далее метод `GetUrlContentLengthAsync` должен вычислить длину загруженной строки и возвратить ее, но не может этого сделать, пока нет строки.

   Поэтому `GetUrlContentLengthAsync` использует оператор await, чтобы приостановить свою работу и передать управление методу, вызвавшему `GetUrlContentLengthAsync`. `GetUrlContentLengthAsync` возвращает вызывающему объекту `Task<int>`. Задача представляет собой обещание создать целочисленный результат, являющийся длиной загруженной строки.

> **NOTE**
> Если метод `GetStringAsync` (и, следовательно, процесс `getStringTask`) завершается прежде, чем этого дождется `GetUrlContentLengthAsync`, управление остается у метода `GetUrlContentLengthAsync`. На приостановку метода `GetUrlContentLengthAsync` и последующий возврат к нему были бы потрачены лишние ресурсы, если вызываемый асинхронный процесс `getStringTask` уже завершен и `GetUrlContentLengthAsync` не нужно ждать окончательного результата.

В вызывающем методе сохраняется шаблон обработки. Вызывающий объект может выполнять другие операции, не зависящие от результата `GetUrlContentLengthAsync`, во время ожидания этого результата, или сразу ожидать результата. Вызывающий метод ожидает `GetUrlContentLengthAsync`, а `GetUrlContentLengthAsync` ожидает `GetStringAsync`.

7. `GetStringAsync` завершается и создает строковый результат. Вызов возвращает строковый результат в метод `GetStringAsync`, но не так, как, возможно, ожидалось. (Помните, что метод уже возвратил задачу в шаге 3.) Вместо этого строковый результат хранится в задаче `getStringTask`, которая представляет собой завершение метода. Оператор await извлекает результат из `getStringTask`. Оператор присваивания назначает извлеченный результат `contents`.

8. Если `GetUrlContentLengthAsync` содержит строковый результат, метод может вычислить длину строки. Затем работа `GetUrlContentLengthAsync` также завершена, и ожидающий обработчик событий может возобновить работу. В полном примере в конце этого раздела видно, что обработчик событий извлекает значение длины и выводит результат. Если вы недавно занимаетесь асинхронным программированием, рекомендуем обратить внимание на различия между синхронным и асинхронным поведением. Синхронный метод возвращает управление, когда его работа завершается (шаг 5), тогда как асинхронный метод возвращает значение задачи, когда его работа приостанавливается (шаги 3 и 6). Когда асинхронный метод в конечном счете завершает работу, задача помечается как завершенная и результат, при его наличии, сохраняется в задаче.

Async-методы API-интерфейсов

Где же найти методы для асинхронного программирования (такие как `GetStringAsync`)? Платформа .NET Framework 4.5 и более поздние версии, а также .NET Core содержат большое количество элементов, совместимых с `async` и `await`. Они содержат суффикс Async в имени элемента и возвращают тип Task или Task<TResult>. Например, класс `System.IO.Stream` имеет такие методы, как CopyToAsync, ReadAsync и WriteAsync, наряду с синхронными методами CopyTo, Read и Write.

Среда выполнения Windows также содержит множество методов, которые можно использовать в сочетании с `async` и `await` в приложениях Windows. См. дополнительные сведения о потоковом и асинхронном программировании для разработки UWP, асинхронном программировании (приложения Магазина Windows), а также вызове асинхронных API в C# или Visual Basic, если вы используете предыдущие версии среды выполнения Windows.

Потоки

Асинхронные методы используются для неблокирующих операций. Выражение `await` в асинхронном методе не блокирует текущий поток на время выполнения ожидаемой задачи. Вместо этого выражение регистрирует остальную часть метода как продолжение и возвращает управление вызывающему объекту асинхронного метода.

Ключевые слова `async` и `await` не вызывают создания дополнительных потоков. Асинхронные методы не требуют многопоточности, поскольку асинхронный метод не выполняется в собственном потоке.

Метод выполняется в текущем контексте синхронизации и использует время в потоке, только когда метод активен. Метод Task.Run можно применять для перемещения операций, использующих ресурсы ЦП, в фоновый поток, однако фоновый поток не имеет смысла применять для процесса, который просто ждет результата.

Асинхронный подход к асинхронному программированию практически по всем параметрам имеет преимущество перед другими подходами. В частности, такой подход эффективнее, чем использование класса BackgroundWorker для привязанных к вводу-выводу операций, так как используется более простой код и вам не нужно специально предотвращать состояние гонки. В сочетании с методом Task.Run асинхронное программирование лучше BackgroundWorker для операций, использующих ресурсы процессора, так как оно отделяет координацию выполнения кода от действий, которые `Task.Run` перемещает в пул потоков.

Async и Await

Если с помощью модификатора async указать, что метод является асинхронным, у вас появятся следующие две возможности.

- Асинхронный метод сможет использовать await для обозначения точек приостановки. Оператор `await` сообщает компилятору, что асинхронный метод не может выполняться после этой точки до завершения ожидаемого асинхронного процесса. На это время управление возвращается вызывающему объекту асинхронного метода.

  Приостановка асинхронного метода на выражении `await` не считается выходом из метода, и блоки `finally` не выполняются.

- Сам обозначенный асинхронный метод может ожидаться вызывающими его методами.

Асинхронный метод обычно содержит одно или несколько вхождений оператора `await`, но отсутствие выражений `await` не вызывает ошибок компилятора. Если асинхронный метод не использует оператор `await` для обозначения точки приостановки, метод выполняется как синхронный независимо от наличия модификатора `async`. При компиляции таких методов выдается предупреждение.

`async` и `await` являются контекстными ключевыми словами. Дополнительные сведения и примеры см. в следующих разделах:

- async
- await

Типы и параметры возвращаемого значения

В результате работы асинхронного метода обычно возвращается Task или Task<TResult>. Внутри асинхронного метода оператор `await` применяется к задаче, возвращаемой из вызова другого асинхронного метода.

В качестве возвращаемого типа указывается Task<TResult>, если метод содержит оператор `return`, который задает операнд типа `TResult`.

Используйте Task в качестве возвращаемого типа, если метод не содержит операторов return или содержит оператор return, который не возвращает операнд.

В C# начиная с версии 7.0 также можно указывать любые другие типы операторов return, при условии, что тип содержит метод `GetAwaiter`. Пример такого типа — ValueTask<TResult>. Он доступен в NuGet-пакете System.Threading.Tasks.Extension.

В следующем примере показано объявление и вызов метода, который возвращает Task<TResult> или Task:

```
async Task<int> GetTaskOfTResultAsync()
{
    int hours = 0;
    await Task.Delay(0);

    return hours;
}


Task<int> returnedTaskTResult = GetTaskOfTResultAsync();
int intResult = await returnedTaskTResult;
// Single line
// int intResult = await GetTaskOfTResultAsync();

async Task GetTaskAsync()
{
    await Task.Delay(0);
    // No return statement needed
}

Task returnedTask = GetTaskAsync();
await returnedTask;
// Single line
await GetTaskAsync();
```

Каждая возвращенная задача представляет выполняющуюся работу. Задача инкапсулирует информацию о состоянии асинхронного процесса и, в итоге, либо конечный результат процесса, либо исключение, созданное процессом в случае его сбоя.

Асинхронный метод может иметь тип возвращаемого значения `void`. Возвращаемый тип в основном используется для определения обработчиков событий, где требуется возвращать тип `void`. Асинхронные обработчики событий часто служат в качестве отправной точки для асинхронных программ.

Асинхронный метод, который имеет тип возвращаемого значения `void`, невозможно ожидать методом await. Вызывающий объект не может перехватывать исключения, которые выдает такой метод.

Асинхронный метод не может объявлять параметры in, ref или out, но может вызывать методы с этими параметрами. Аналогичным образом, асинхронный метод не может возвращать значение по ссылке, несмотря на то, что он может вызывать методы с возвращаемыми значениями ref.

Дополнительные сведения и примеры см. в статье Типы возвращаемых значений асинхронных операций (C#). Дополнительные сведения о перехвате исключений в асинхронных методах см. в описании механизма try-catch.

При программировании в среде выполнения Windows асинхронные API-интерфейсы имеют один из следующих возвращаемых типов, которые похожи на задачи.

- IAsyncOperation<TResult>, что соответствует Task<TResult>
- IAsyncAction, что соответствует Task
- IAsyncActionWithProgress<TProgress>
- IAsyncOperationWithProgress<TResult,TProgress>

Соглашение об именовании

По соглашению имена методов, возвращающих обычно поддерживающие ожидание типы (например, `Task`, `Task<T>`, `ValueTask` и `ValueTask<T>`), должны заканчиваться на Async. Имена методов, которые запускают асинхронную операцию, но не возвращают поддерживающий ожидание тип, не должны заканчиваться на Async, но могут начинаться с Begin, Start или другой команды, предполагающей, что

метод не возвращает и не выдает результат операции.

Соглашение можно игнорировать в тех случаях, когда событие, базовый класс или контракт интерфейса предлагает другое имя. Например, не следует переименовывать общие обработчики событий, такие как `OnButtonClick`.

Связанные статьи (Visual Studio)

ЗАГОЛОВОК	ОПИСАНИЕ
Практическое руководство. Параллельное выполнение нескольких веб-запросов с использованием Async и Await (C#)	Иллюстрирует, как запустить несколько задач одновременно.
Типы возвращаемых значений асинхронных операций (C#)	Иллюстрирует типы, которые могут возвращать асинхронные методы, и поясняет, когда следует использовать каждый из этих типов.
Отмените задачи с маркером отмены в качестве механизма сигнализации.	Иллюстрирует добавление следующих функциональных возможностей в асинхронное решение: - Отмена списка задач (C#) - Отмена задач после определенного периода времени (C#) - Обработка асинхронных задач по мере завершения (C#)
Использование метода async для доступа к файлам (C#)	Иллюстрирует преимущества использования асинхронности и ожидания для доступа к файлам.
Асинхронный шаблон, основанный на задачах (TAP)	Описывает асинхронную модель, шаблон основан на типах Task и Task<TResult>.
Видеоролики об async на канале Channel 9	Предоставляет ссылки на различные видеоролики об асинхронном программировании.

См. также

- async
- await
- Асинхронное программирование
- Общие сведения об асинхронной модели

Типы возвращаемых значений асинхронных операций (C#)

18.12.2021 • 8 minutes to read

Асинхронные методы могут иметь следующие типы возвращаемых значений:

- Task для асинхронного метода, который выполняет операцию, но не возвращает значение.
- Task<TResult> для асинхронного метода, возвращающего значение.
- `void` для обработчика событий.
- Начиная с версии 7.0 в языке C# поддерживаются любые типы с доступным методом `GetAwaiter`. Объект, возвращаемый методом `GetAwaiter`, должен реализовывать интерфейс System.Runtime.CompilerServices.ICriticalNotifyCompletion.
- Начиная с версии 8.0, в языке C# поддерживается интерфейс IAsyncEnumerable<T> для асинхронного метода, который возвращает *асинхронный поток*.

Дополнительные сведения об асинхронных методах см. в разделе Асинхронное программирование с использованием ключевых слов async и await (C#).

Существуют также некоторые другие типы, характерные для рабочих нагрузок Windows.

- DispatcherOperation для асинхронных операций, ограниченных Windows.
- IAsyncAction для асинхронных действий в UWP, которые не возвращают значение.
- IAsyncActionWithProgress<TProgress> для асинхронных действий в UWP, которые сообщают о ходе выполнения, но не возвращают значение.
- IAsyncOperation<TResult> для асинхронных операций в UWP, возвращающих значение.
- IAsyncOperationWithProgress<TResult,TProgress> для асинхронных операций в UWP, которые сообщают о ходе выполнения и возвращают значение.

Тип возвращаемого значения Task

Асинхронные методы, не содержащие инструкцию `return` или содержащие инструкцию `return`, которая не возвращает операнд, обычно имеют тип возвращаемого значения Task. При синхронном выполнении такие методы возвращают `void`. Если для асинхронного метода вы используете тип возвращаемого значения Task, вызывающий метод может использовать оператор `await` для приостановки выполнения вызывающего объекта до завершения вызванного асинхронного метода.

В следующем примере метод `WaitAndApologizeAsync` не содержит инструкцию `return`, в связи с чем он возвращает объект Task. Возврат `Task` позволяет реализовать ожидание `WaitAndApologizeAsync`. Тип Task не имеет возвращаемого значения и, соответственно, не содержит свойство `Result`.

```
public static async Task DisplayCurrentInfoAsync()
{
    await WaitAndApologizeAsync();

    Console.WriteLine($"Today is {DateTime.Now:D}");
    Console.WriteLine($"The current time is {DateTime.Now.TimeOfDay:t}");
    Console.WriteLine("The current temperature is 76 degrees.");
}

static async Task WaitAndApologizeAsync()
{
    await Task.Delay(2000);

    Console.WriteLine("Sorry for the delay...\n");
}
// Example output:
//    Sorry for the delay...
//
// Today is Monday, August 17, 2020
// The current time is 12:59:24.2183304
// The current temperature is 76 degrees.
```

`WaitAndApologizeAsync` вызывается и ожидается с помощью инструкции await (вместо выражения await), похожей на инструкцию вызова для синхронного метода, возвращающего значение void. Применение оператора await в этом случае не возвращает значение. Если правый операнд `await` имеет значение Task<TResult>, `await` выражение возвращает результат для `T`. Если же правый операнд `await` имеет значение Task, `await` и его операнд являются оператором.

Можно отделить вызов `WaitAndApologizeAsync` от применения инструкции await, как показывает следующий код. Однако следует помнить, что `Task` не содержит свойство `Result`, и при применении оператора await к `Task` никакое значение не создается.

В следующем коде вызов метода `WaitAndApologizeAsync` отделяется от ожидания задачи, которую возвращает этот метод.

```
Task waitAndApologizeTask = WaitAndApologizeAsync();

string output =
    $"Today is {DateTime.Now:D}\n" +
    $"The current time is {DateTime.Now.TimeOfDay:t}\n" +
    "The current temperature is 76 degrees.\n";

await waitAndApologizeTask;
Console.WriteLine(output);
```

Тип возвращаемого значения Task<TResult>

Тип возвращаемого значения Task<TResult> используется для асинхронного метода, содержащего инструкцию return с операндом типа `TResult`.

В следующем примере метод `GetLeisureHoursAsync` содержит инструкцию `return`, которая возвращает целое число. В объявлении метода должен указываться тип возвращаемого значения `Task<int>`. Асинхронный метод FromResult представляет собой заполнитель для операции, которая возвращает DayOfWeek.

```
public static async Task ShowTodaysInfoAsync()
{
    string message =
        $"Today is {DateTime.Today:D}\n" +
        "Today's hours of leisure: " +
        $"{await GetLeisureHoursAsync()}";

    Console.WriteLine(message);
}

static async Task<int> GetLeisureHoursAsync()
{
    DayOfWeek today = await Task.FromResult(DateTime.Now.DayOfWeek);

    int leisureHours =
        today is DayOfWeek.Saturday || today is DayOfWeek.Sunday
        ? 16 : 5;

    return leisureHours;
}
// Example output:
//    Today is Wednesday, May 24, 2017
//    Today's hours of leisure: 5
```

При вызове `GetLeisureHoursAsync` из выражения await в методе `ShowTodaysInfo` это выражение await извлекает целочисленное значение (значение `leisureHours`), хранящееся в задаче, которая возвращается методом `GetLeisureHours`. Дополнительные сведения о выражениях await см. в разделе await.

Чтобы лучше понять, как `await` получает результат из `Task<T>`, отделите вызов метода `GetLeisureHoursAsync` от применения `await`, как показано в следующем коде. Вызов метода `GetLeisureHoursAsync`, который не ожидается немедленно, возвращает `Task<int>`, как и следовало ожидать из объявления метода. В данном примере эта задача назначается переменной `getLeisureHoursTask`. Поскольку `getLeisureHoursTask` является Task<TResult>, она содержит свойство Result типа `TResult`. В этом примере `TResult` представляет собой целочисленный тип. Если выражение `await` применяется к `getLeisureHoursTask`, выражение await вычисляется как содержимое свойства Result объекта `getLeisureHoursTask`. Это значение присваивается переменной `ret`.

> **IMPORTANT**
>
> Свойство Result является блокирующим свойством. При попытке доступа к нему до завершения его задачи поток, который в текущий момент активен, блокируется до того момента, пока задача не будет завершена, а ее значение не станет доступным. В большинстве случаев следует получать доступ к этому значению с помощью `await` вместо прямого обращения к свойству.
>
> В предыдущем примере извлекалось значение свойства Result для блокировки основного потока. Это позволяет методу `Main` вывести `message` в окно консоли до того, как завершится работа приложения.

```
var getLeisureHoursTask = GetLeisureHoursAsync();

string message =
    $"Today is {DateTime.Today:D}\n" +
    "Today's hours of leisure: " +
    $"{await getLeisureHoursTask}";

Console.WriteLine(message);
```

Тип возвращаемого значения Void

Тип возвращаемого значения `void` используется в асинхронных обработчиках событий, для которых требуется тип возвращаемого значения `void`. Поскольку методы, не являющиеся обработчиками событий, не возвращают значения, вместо этого необходимо вернуть Task. Это вызвано тем, что для асинхронных методов, возвращающих значение `void`, ожидание невозможно. Любой вызывающий объект такого метода должен иметь возможность завершить свою работу, не дожидаясь завершения вызванного асинхронного метода. Вызывающий объект не должен зависеть ни от каких значений и исключений, создаваемых асинхронным методом.

Вызывающий объект асинхронного метода, возвращающего void, не может перехватывать создаваемые методом исключения. Такие необработанные исключения, скорее всего, приведут к сбою приложения. Если метод, возвращающий Task или Task<TResult>, создает исключение, оно хранится в возвращенной задаче. Исключение повторно вызывается при ожидании задачи. Убедитесь, что любой асинхронный метод, который может вызвать исключение, имеет тип возвращаемого значения Task или Task<TResult> и что вызовы метода являются ожидаемыми.

Дополнительные сведения о перехвате исключений в асинхронных методах см. в разделе Исключения в асинхронных методах в статье о try-catch.

В следующем примере показано поведение асинхронного обработчика событий. В примере кода асинхронный обработчик событий должен сообщить основному потоку о завершении своей работы. Основной поток может ожидать завершения работы асинхронного обработчика событий перед выходом из программы.

```
using System;
using System.Threading.Tasks;

public class NaiveButton
{
    public event EventHandler? Clicked;

    public void Click()
    {
        Console.WriteLine("Somebody has clicked a button. Let's raise the event...");
        Clicked?.Invoke(this, EventArgs.Empty);
        Console.WriteLine("All listeners are notified.");
    }
}

public class AsyncVoidExample
{
    static readonly TaskCompletionSource<bool> s_tcs = new TaskCompletionSource<bool>();

    public static async Task MultipleEventHandlersAsync()
    {
        Task<bool> secondHandlerFinished = s_tcs.Task;

        var button = new NaiveButton();

        button.Clicked += OnButtonClicked1;
        button.Clicked += OnButtonClicked2Async;
        button.Clicked += OnButtonClicked3;

        Console.WriteLine("Before button.Click() is called...");
        button.Click();
        Console.WriteLine("After button.Click() is called...");

        await secondHandlerFinished;
    }

    private static void OnButtonClicked1(object? sender, EventArgs e)
    {
        Console.WriteLine("   Handler 1 is starting...");
        Task.Delay(100).Wait();
```

```
        Console.WriteLine("   Handler 1 is done.");
    }

    private static async void OnButtonClicked2Async(object? sender, EventArgs e)
    {
        Console.WriteLine("   Handler 2 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("   Handler 2 is about to go async...");
        await Task.Delay(500);
        Console.WriteLine("   Handler 2 is done.");
        s_tcs.SetResult(true);
    }

    private static void OnButtonClicked3(object? sender, EventArgs e)
    {
        Console.WriteLine("   Handler 3 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("   Handler 3 is done.");
    }
}
// Example output:
//
// Before button.Click() is called...
// Somebody has clicked a button. Let's raise the event...
//    Handler 1 is starting...
//    Handler 1 is done.
//    Handler 2 is starting...
//    Handler 2 is about to go async...
//    Handler 3 is starting...
//    Handler 3 is done.
// All listeners are notified.
// After button.Click() is called...
//    Handler 2 is done.
```

Обобщенные асинхронные типы возвращаемых значений и ValueTask<TResult>

Начиная с C# 7.0, асинхронные методы могут возвращать любой тип с доступным методом `GetAwaiter`, который возвращает экземпляр *типа awaiter*. Также тип, возвращаемый методом `GetAwaiter`, должен иметь атрибут System.Runtime.CompilerServices.AsyncMethodBuilderAttribute. Дополнительные сведения см. в статье, посвященной атрибутам, считываемым компилятором, или в спецификациях функций для типов возвращаемого значения, подобных задаче.

Эта функция является дополнением к выражениям с поддержкой await с описанием требований для операнда `await`. Обобщенные асинхронные типы возвращаемых значений позволяют компилятору создавать методы `async`, возвращающие различные типы. Благодаря использованию обобщенных асинхронных типов возвращаемых значений производительность в библиотеках .NET повысилась. Поскольку Task и Task<TResult> являются ссылочными типами, выделение памяти во влияющих на производительность сегментах (особенно при выделении памяти в ограниченных циклах) может серьезно снизить производительность. Поддержка обобщенных типов возвращаемых значений позволяет возвращать небольшой тип значения вместо ссылочного типа, благодаря чему удается предотвратить избыточное выделение памяти.

На платформе .NET представлена структура System.Threading.Tasks.ValueTask<TResult>, которая является упрощенной реализацией обобщенного значения, возвращающего задачу. Чтобы использовать тип System.Threading.Tasks.ValueTask<TResult>, необходимо добавить в проект пакет NuGet `System.Threading.Tasks.Extensions`. В следующем примере структура ValueTask<TResult> используется для извлечения значений двух игральных костей.

```
using System;
using System.Threading.Tasks;

class Program
{
    static readonly Random s_rnd = new Random();

    static async Task Main() =>
        Console.WriteLine($"You rolled {await GetDiceRollAsync()}");

    static async ValueTask<int> GetDiceRollAsync()
    {
        Console.WriteLine("Shaking dice...");

        int roll1 = await RollAsync();
        int roll2 = await RollAsync();

        return roll1 + roll2;
    }

    static async ValueTask<int> RollAsync()
    {
        await Task.Delay(500);

        int diceRoll = s_rnd.Next(1, 7);
        return diceRoll;
    }
}
// Example output:
//    Shaking dice...
//    You rolled 8
```

Создание обобщенного асинхронного типа возвращаемого значения — сложный сценарий, который используется в специализированных средах. Вместо этого можно применять типы `Task`, `Task<T>` и `ValueTask<T>`, которые используются в большинстве сценариев для асинхронного кода.

В C# 10 и более поздних версиях к асинхронному методу можно применить атрибут `AsyncMethodBuilder` (вместо объявления асинхронного типа возвращаемого значения), чтобы переопределить построитель для этого типа. Обычно этот атрибут применяется для использования другого построителя, предоставленного в среде выполнения .NET.

Асинхронные потоки с IAsyncEnumerable<T>

Начиная с C# 8.0, асинхронный метод может возвращать *асинхронный поток*, представленный интерфейсом IAsyncEnumerable<T>. Асинхронный поток позволяет перечислять элементы, считываемые из потока, при создании блоков элементов с помощью повторяющихся асинхронных вызовов. В следующем примере показан асинхронный метод, создающий асинхронный поток.

```
static async IAsyncEnumerable<string> ReadWordsFromStreamAsync()
{
    string data =
        @"This is a line of text.
          Here is the second line of text.
          And there is one more for good measure.
          Wait, that was the penultimate line.";

    using var readStream = new StringReader(data);

    string line = await readStream.ReadLineAsync();
    while (line != null)
    {
        foreach (string word in line.Split(' ', StringSplitOptions.RemoveEmptyEntries))
        {
            yield return word;
        }

        line = await readStream.ReadLineAsync();
    }
}
```

В предыдущем примере показано асинхронное считывание строк. После считывания каждой строки код перечисляет каждое слово в строке. Вызывающие объекты будут перечислять каждое слово с помощью оператора `await foreach`. Метод ожидает, когда необходимо асинхронно считать следующую строку из исходной строки.

См. также

- FromResult
- Обработка асинхронных задач по мере завершения
- Асинхронное программирование с использованием ключевых слов Async и Await (C#)
- async
- await

Отмена списка задач (C#)

18.12.2021 • 4 minutes to read

Вы можете отменить асинхронное консольное приложение, если не хотите дожидаться его завершения. Выполнив код в приведенных ниже примерах, вы сможете добавить в приложение возможность отмены, загружающую содержимое списка веб-сайтов. Можно отменить множество задач, связав экземпляр CancellationTokenSource с каждой задачей. В этом случае нажатие клавиши ВВОД отменяет сразу все незавершенные задачи.

Темы, рассматриваемые в этом руководстве:

- Создание консольного приложения .NET
- Создание асинхронного приложения, поддерживающего отмену
- Демонстрация отмены сигнализации

Предварительные требования

Для работы с данным учебником требуется следующее:

- Пакет SDK для .NET 5 или более поздней версии
- Интегрированная среда разработки (IDE)
  - Мы рекомендуем Visual Studio, Visual Studio Code или Visual Studio для Mac

Создание примера приложения

Создайте новое консольное приложение .NET Core. Его можно создать с помощью команды `dotnet new console` или в Visual Studio. Откройте файл *Program.cs* в любом редакторе кода.

Замена операторов using

Замените существующие операторы using следующими объявлениями.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;
```

Добавить поля

В определении класса `Program` добавьте следующие три поля.

```
static readonly CancellationTokenSource s_cts = new CancellationTokenSource();

static readonly HttpClient s_client = new HttpClient
{
    MaxResponseContentBufferSize = 1_000_000
};

static readonly IEnumerable<string> s_urlList = new string[]
{
    "https://docs.microsoft.com",
    "https://docs.microsoft.com/aspnet/core",
    "https://docs.microsoft.com/azure",
    "https://docs.microsoft.com/azure/devops",
    "https://docs.microsoft.com/dotnet",
    "https://docs.microsoft.com/dynamics365",
    "https://docs.microsoft.com/education",
    "https://docs.microsoft.com/enterprise-mobility-security",
    "https://docs.microsoft.com/gaming",
    "https://docs.microsoft.com/graph",
    "https://docs.microsoft.com/microsoft-365",
    "https://docs.microsoft.com/office",
    "https://docs.microsoft.com/powershell",
    "https://docs.microsoft.com/sql",
    "https://docs.microsoft.com/surface",
    "https://docs.microsoft.com/system-center",
    "https://docs.microsoft.com/visualstudio",
    "https://docs.microsoft.com/windows",
    "https://docs.microsoft.com/xamarin"
};
```

CancellationTokenSource используется для сигнализации запрошенной отмены CancellationToken. `HttpClient` предоставляет возможность отправлять HTTP-запросы и получать HTTP-ответы. `s_urlList` содержит все URL-адреса, которые планируется обработать приложением.

Обновление точки входа приложения

Главной точкой входа в консольное приложение является метод `Main`. Замените существующий метод следующим кодом.

```
static async Task Main()
{
    Console.WriteLine("Application started.");
    Console.WriteLine("Press the ENTER key to cancel...\n");

    Task cancelTask = Task.Run(() =>
    {
        while (Console.ReadKey().Key != ConsoleKey.Enter)
        {
            Console.WriteLine("Press the ENTER key to cancel...");
        }

        Console.WriteLine("\nENTER key pressed: cancelling downloads.\n");
        s_cts.Cancel();
    });

    Task sumPageSizesTask = SumPageSizesAsync();

    await Task.WhenAny(new[] { cancelTask, sumPageSizesTask });

    Console.WriteLine("Application ending.");
}
```

Обновленный метод `Main` теперь считается асинхронным методом main, который позволяет использовать асинхронную точку входа в исполняемом файле. Он записывает в консоль несколько сообщений с инструкциями, а затем объявляет экземпляр Task с именем `cancelTask`, который будет считывать нажатия клавиш в консоли. Если нажата клавиша ВВОД, выполняется вызов CancellationTokenSource.Cancel(). Это приведет к отмене сигнала. Затем переменная `sumPageSizesTask` назначается методом `SumPageSizesAsync`. Затем обе задачи передаются в Task.WhenAny(Task[]), что будет продолжено после завершения любой из этих задач.

Создание метода асинхронного суммирования размеров страниц

Добавьте метод `Main` под методом `SumPageSizesAsync`.

```
static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    int total = 0;
    foreach (string url in s_urlList)
    {
        int contentLength = await ProcessUrlAsync(url, s_client, s_cts.Token);
        total += contentLength;
    }

    stopwatch.Stop();

    Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
    Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
}
```

Метод начинается с создания экземпляра и запуска Stopwatch. Затем он просматривает каждый URL-адрес в `s_urlList` и вызывает `ProcessUrlAsync`. При каждой итерации в метод `ProcessUrlAsync` передается `s_cts.Token`, а код возвращает Task<TResult>, где `TResult` является целым числом.

```
int total = 0;
foreach (string url in s_urlList)
{
    int contentLength = await ProcessUrlAsync(url, s_client, s_cts.Token);
    total += contentLength;
}
```

Добавление метода обработки

Добавьте приведенный ниже метод `ProcessUrlAsync` после метода `SumPageSizesAsync`.

```
static async Task<int> ProcessUrlAsync(string url, HttpClient client, CancellationToken token)
{
    HttpResponseMessage response = await client.GetAsync(url, token);
    byte[] content = await response.Content.ReadAsByteArrayAsync(token);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}
```

Для любого заданного URL-адреса метод будет использовать экземпляр `client`, предоставленный для получения ответа в качестве `byte[]`. CancellationToken экземпляр передается в методы HttpClient.GetAsync(String, CancellationToken) и HttpContent.ReadAsByteArrayAsync(). `token` используется для регистрации запрошенной отмены. Длина возвращается после того, как URL-адрес и длина

записываются в консоль.

Пример выходных данных приложения

```
Application started.
Press the ENTER key to cancel...

https://docs.microsoft.com                                 37,357
https://docs.microsoft.com/aspnet/core                     85,589
https://docs.microsoft.com/azure                          398,939
https://docs.microsoft.com/azure/devops                    73,663
https://docs.microsoft.com/dotnet                          67,452
https://docs.microsoft.com/dynamics365                     48,582
https://docs.microsoft.com/education                       22,924

ENTER key pressed: cancelling downloads.

Application ending.
```

Полный пример

Приведенный ниже код — это полный текст файла *Program.cs* для примера.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;

class Program
{
    static readonly CancellationTokenSource s_cts = new CancellationTokenSource();

    static readonly HttpClient s_client = new HttpClient
    {
        MaxResponseContentBufferSize = 1_000_000
    };

    static readonly IEnumerable<string> s_urlList = new string[]
    {
            "https://docs.microsoft.com",
            "https://docs.microsoft.com/aspnet/core",
            "https://docs.microsoft.com/azure",
            "https://docs.microsoft.com/azure/devops",
            "https://docs.microsoft.com/dotnet",
            "https://docs.microsoft.com/dynamics365",
            "https://docs.microsoft.com/education",
            "https://docs.microsoft.com/enterprise-mobility-security",
            "https://docs.microsoft.com/gaming",
            "https://docs.microsoft.com/graph",
            "https://docs.microsoft.com/microsoft-365",
            "https://docs.microsoft.com/office",
            "https://docs.microsoft.com/powershell",
            "https://docs.microsoft.com/sql",
            "https://docs.microsoft.com/surface",
            "https://docs.microsoft.com/system-center",
            "https://docs.microsoft.com/visualstudio",
            "https://docs.microsoft.com/windows",
            "https://docs.microsoft.com/xamarin"
    };

    static async Task Main()
    {
        Console.WriteLine("Application started.");
```

```
        Console.WriteLine("Press the ENTER key to cancel...\n");

        Task cancelTask = Task.Run(() =>
        {
            while (Console.ReadKey().Key != ConsoleKey.Enter)
            {
                Console.WriteLine("Press the ENTER key to cancel...");
            }

            Console.WriteLine("\nENTER key pressed: cancelling downloads.\n");
            s_cts.Cancel();
        });

        Task sumPageSizesTask = SumPageSizesAsync();

        await Task.WhenAny(new[] { cancelTask, sumPageSizesTask });

        Console.WriteLine("Application ending.");
    }

    static async Task SumPageSizesAsync()
    {
        var stopwatch = Stopwatch.StartNew();

        int total = 0;
        foreach (string url in s_urlList)
        {
            int contentLength = await ProcessUrlAsync(url, s_client, s_cts.Token);
            total += contentLength;
        }

        stopwatch.Stop();

        Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
        Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
    }

    static async Task<int> ProcessUrlAsync(string url, HttpClient client, CancellationToken token)
    {
        HttpResponseMessage response = await client.GetAsync(url, token);
        byte[] content = await response.Content.ReadAsByteArrayAsync(token);
        Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

        return content.Length;
    }
}
```

См. также раздел

- CancellationToken
- CancellationTokenSource
- Асинхронное программирование с использованием ключевых слов Async и Await (C#)

Дальнейшие действия

Отмена асинхронных задач после определенного периода времени (C#)

Отмена асинхронных задач после определенного периода времени (C#)

18.12.2021 • 2 minutes to read

Если не нужно дожидаться, пока завершится выполнение асинхронной операции, ее можно отменить по истечении определенного периода времени с помощью метода CancellationTokenSource.CancelAfter. Этот метод планирует отмену всех связанных задач, не завершенных в течение времени, установленного выражением `CancelAfter`.

В этом примере добавляется код, составленный в разделе Отмена списка задач (C#), для загрузки списка веб-сайтов и отображения длины содержимого каждого из них.

Темы, рассматриваемые в этом руководстве:

- Обновление существующего консольного приложения .NET
- Планирование отмены

Предварительные требования

Для работы с данным учебником требуется следующее:

- Предполагается, что вы создали приложение в учебнике Отмена списка задач (C#).
- Пакет SDK для .NET 5 или более поздней версии
- Интегрированная среда разработки (IDE)
  - Мы рекомендуем Visual Studio, Visual Studio Code или Visual Studio для Mac

Обновление точки входа приложения

Замените существующий метод `Main` следующим кодом.

```
static async Task Main()
{
    Console.WriteLine("Application started.");

    try
    {
        s_cts.CancelAfter(3500);

        await SumPageSizesAsync();
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine("\nTasks cancelled: timed out.\n");
    }
    finally
    {
        s_cts.Dispose();
    }

    Console.WriteLine("Application ending.");
}
```

Обновленный метод `Main` записывает в консоль несколько инструкций. В операторе try-catch вызов CancellationTokenSource.CancelAfter(Int32) позволяет запланировать отмену. Эта операция будет

сообщать об отмене по истечении определенного периода времени.

Затем ожидается метод `SumPageSizesAsync`. Если обработка всех URL-адресов выполняется быстрее запланированной отмены, приложение завершается. Однако если запланированная отмена запускается до обработки всех URL-адресов, создается OperationCanceledException.

Пример выходных данных приложения

```
Application started.

https://docs.microsoft.com                                    37,357
https://docs.microsoft.com/aspnet/core                        85,589
https://docs.microsoft.com/azure                             398,939
https://docs.microsoft.com/azure/devops                       73,663

Tasks cancelled: timed out.

Application ending.
```

Полный пример

Приведенный ниже код — это полный текст файла *Program.cs* для примера.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;

class Program
{
    static readonly CancellationTokenSource s_cts = new CancellationTokenSource();

    static readonly HttpClient s_client = new HttpClient
    {
        MaxResponseContentBufferSize = 1_000_000
    };

    static readonly IEnumerable<string> s_urlList = new string[]
    {
            "https://docs.microsoft.com",
            "https://docs.microsoft.com/aspnet/core",
            "https://docs.microsoft.com/azure",
            "https://docs.microsoft.com/azure/devops",
            "https://docs.microsoft.com/dotnet",
            "https://docs.microsoft.com/dynamics365",
            "https://docs.microsoft.com/education",
            "https://docs.microsoft.com/enterprise-mobility-security",
            "https://docs.microsoft.com/gaming",
            "https://docs.microsoft.com/graph",
            "https://docs.microsoft.com/microsoft-365",
            "https://docs.microsoft.com/office",
            "https://docs.microsoft.com/powershell",
            "https://docs.microsoft.com/sql",
            "https://docs.microsoft.com/surface",
            "https://docs.microsoft.com/system-center",
            "https://docs.microsoft.com/visualstudio",
            "https://docs.microsoft.com/windows",
            "https://docs.microsoft.com/xamarin"
    };

    static async Task Main()
    {
```

```
        Console.WriteLine("Application started.");

        try
        {
            s_cts.CancelAfter(3500);

            await SumPageSizesAsync();
        }
        catch (OperationCanceledException)
        {
            Console.WriteLine("\nTasks cancelled: timed out.\n");
        }
        finally
        {
            s_cts.Dispose();
        }

        Console.WriteLine("Application ending.");
    }

    static async Task SumPageSizesAsync()
    {
        var stopwatch = Stopwatch.StartNew();

        int total = 0;
        foreach (string url in s_urlList)
        {
            int contentLength = await ProcessUrlAsync(url, s_client, s_cts.Token);
            total += contentLength;
        }

        stopwatch.Stop();

        Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
        Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
    }

    static async Task<int> ProcessUrlAsync(string url, HttpClient client, CancellationToken token)
    {
        HttpResponseMessage response = await client.GetAsync(url, token);
        byte[] content = await response.Content.ReadAsByteArrayAsync(token);
        Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

        return content.Length;
    }
}
```

См. также раздел

- CancellationToken
- CancellationTokenSource
- Асинхронное программирование с использованием ключевых слов Async и Await (C#)
- Отмена списка задач (C#)

Обработка асинхронных задач по мере завершения (C#)

18.12.2021 • 8 minutes to read

С помощью Task.WhenAny можно запускать несколько задач одновременно и обрабатывать их по одной по мере завершения, а не в порядке их запуска.

В следующем примере используется запрос для создания коллекции задач. Каждая задача загружает содержимое указанного веб-сайта. В каждой итерации цикла while ожидаемый вызов WhenAny возвращает задачу из коллекции задач, которая первой завершает свою загрузку. Эта задача удаляется из коллекции и обрабатывается. Цикл выполняется до тех пор, пока в коллекции еще есть задачи.

Предварительные требования

Этот учебник можно выполнить с помощью одного из следующих вариантов:

- Visual Studio 2022, предварительная версия 17.0.0 с установленной рабочей нагрузкой **Разработка классических приложений .NET**. Пакет SDK для .NET 6.0 устанавливается автоматически при выборе этой рабочей нагрузки.
- Пакет SDK для .NET 6.0 с любым редактором кода, например Visual Studio Code.

Создание примера приложения

Создайте новое консольное приложение .NET Core для .NET 6.0. Его можно создать с помощью команды dotnet new console или в Visual Studio.

Откройте *Program.cs* в любом редакторе кода и замените существующий код следующим:

```
using System.Diagnostics;

namespace ProcessTasksAsTheyFinish;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```

Добавить поля

Добавьте в определение класса `Program` следующие два поля.

```
static readonly HttpClient s_client = new HttpClient
{
    MaxResponseContentBufferSize = 1_000_000
};

static readonly IEnumerable<string> s_urlList = new string[]
{
    "https://docs.microsoft.com",
    "https://docs.microsoft.com/aspnet/core",
    "https://docs.microsoft.com/azure",
    "https://docs.microsoft.com/azure/devops",
    "https://docs.microsoft.com/dotnet",
    "https://docs.microsoft.com/dynamics365",
    "https://docs.microsoft.com/education",
    "https://docs.microsoft.com/enterprise-mobility-security",
    "https://docs.microsoft.com/gaming",
    "https://docs.microsoft.com/graph",
    "https://docs.microsoft.com/microsoft-365",
    "https://docs.microsoft.com/office",
    "https://docs.microsoft.com/powershell",
    "https://docs.microsoft.com/sql",
    "https://docs.microsoft.com/surface",
    "https://docs.microsoft.com/system-center",
    "https://docs.microsoft.com/visualstudio",
    "https://docs.microsoft.com/windows",
    "https://docs.microsoft.com/xamarin"
};
```

`HttpClient` предоставляет возможность отправлять HTTP-запросы и получать HTTP-ответы. `s_urlList` содержит все URL-адреса, которые планируется обработать приложением.

Обновление точки входа приложения

Главной точкой входа в консольное приложение является метод `Main`. Замените существующий метод следующим кодом.

```
static Task Main() => SumPageSizesAsync();
```

Обновленный метод `Main` теперь считается асинхронным методом main, который позволяет использовать асинхронную точку входа в исполняемом файле. Он выражается вызовом `SumPageSizesAsync`.

Создание метода асинхронного суммирования размеров страниц

Добавьте метод `Main` под методом `SumPageSizesAsync`.

```
static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    IEnumerable<Task<int>> downloadTasksQuery =
        from url in s_urlList
        select ProcessUrlAsync(url, s_client);

    List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

    int total = 0;
    while (downloadTasks.Any())
    {
        Task<int> finishedTask = await Task.WhenAny(downloadTasks);
        downloadTasks.Remove(finishedTask);
        total += await finishedTask;
    }

    stopwatch.Stop();

    Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
    Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
}
```

Метод начинается с создания экземпляра и запуска Stopwatch. Затем он включает запрос, который при выполнении создает коллекцию задач. Каждый вызов `ProcessUrlAsync` в следующем коде возвращает Task<TResult>, где `TResult` — целое число.

```
IEnumerable<Task<int>> downloadTasksQuery =
    from url in s_urlList
    select ProcessUrlAsync(url, s_client);
```

Из-за отложенного выполнения с помощью LINQ для запуска каждой задачи вызывается Enumerable.ToList.

```
List<Task<int>> downloadTasks = downloadTasksQuery.ToList();
```

Цикл `while` выполняет следующие действия для каждой задачи в коллекции.

1. Ожидает вызов `WhenAny` для определения первой задачи в коллекции, чтобы завершить ее загрузку.

   ```
   Task<int> finishedTask = await Task.WhenAny(downloadTasks);
   ```

2. Удаляет эту задачу из коллекции.

   ```
   downloadTasks.Remove(finishedTask);
   ```

3. Ожидает `finishedTask`, возвращаемый при вызове `ProcessUrlAsync`. Переменная `finishedTask` представляет собой Task<TResult>, где `TResult` — целое число. Задача уже завершена, но она ожидается для получения размера загруженного веб-сайта, как показано в следующем примере. В случае сбоя задачи `await` выдаст первое исключение дочернего элемента, хранящееся в `AggregateException`, в отличие от считывания свойства Task<TResult>.Result, которое выдаст `AggregateException`.

```
    total += await finishedTask;
```

Добавление метода обработки

Добавьте приведенный ниже метод `ProcessUrlAsync` после метода `SumPageSizesAsync`.

```
static async Task<int> ProcessUrlAsync(string url, HttpClient client)
{
    byte[] content = await client.GetByteArrayAsync(url);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}
```

Для любого заданного URL-адреса метод будет использовать экземпляр `client`, предоставленный для получения ответа в качестве `byte[]`. Длина возвращается после того, как URL-адрес и длина записываются в консоль.

Запустите проект несколько раз и убедитесь, что размеры скачанных файлов не всегда отображаются в одном и том же порядке.

Caution

Можно использовать `WhenAny` в цикле, как описано в примере, для решения проблем, которые включают небольшое число задач. Однако когда требуется обработка большого числа задач, другие методы будут более эффективны. Дополнительные сведения и примеры см. в разделе Обработка задач по мере их завершения.

Полный пример

Приведенный ниже код — это полный текст файла *Program.cs* для примера.

```
using System.Diagnostics;

namespace ProcessTasksAsTheyFinish;

class Program
{
    static readonly HttpClient s_client = new HttpClient
    {
        MaxResponseContentBufferSize = 1_000_000
    };

    static readonly IEnumerable<string> s_urlList = new string[]
    {
            "https://docs.microsoft.com",
            "https://docs.microsoft.com/aspnet/core",
            "https://docs.microsoft.com/azure",
            "https://docs.microsoft.com/azure/devops",
            "https://docs.microsoft.com/dotnet",
            "https://docs.microsoft.com/dynamics365",
            "https://docs.microsoft.com/education",
            "https://docs.microsoft.com/enterprise-mobility-security",
            "https://docs.microsoft.com/gaming",
            "https://docs.microsoft.com/graph",
            "https://docs.microsoft.com/microsoft-365",
            "https://docs.microsoft.com/office",
            "https://docs.microsoft.com/powershell",
            "https://docs.microsoft.com/sql",
            "https://docs.microsoft.com/surface",
            "https://docs.microsoft.com/system-center",
```

```
            "https://docs.microsoft.com/visualstudio",
            "https://docs.microsoft.com/windows",
            "https://docs.microsoft.com/xamarin"
    };

    static Task Main() => SumPageSizesAsync();

    static async Task SumPageSizesAsync()
    {
        var stopwatch = Stopwatch.StartNew();

        IEnumerable<Task<int>> downloadTasksQuery =
            from url in s_urlList
            select ProcessUrlAsync(url, s_client);

        List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

        int total = 0;
        while (downloadTasks.Any())
        {
            Task<int> finishedTask = await Task.WhenAny(downloadTasks);
            downloadTasks.Remove(finishedTask);
            total += await finishedTask;
        }

        stopwatch.Stop();

        Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
        Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
    }

    static async Task<int> ProcessUrlAsync(string url, HttpClient client)
    {
        byte[] content = await client.GetByteArrayAsync(url);
        Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

        return content.Length;
    }
}

// Example output:
// https://docs.microsoft.com/windows                              25,513
// https://docs.microsoft.com/gaming                               30,705
// https://docs.microsoft.com/dotnet                               69,626
// https://docs.microsoft.com/dynamics365                          50,756
// https://docs.microsoft.com/surface                              35,519
// https://docs.microsoft.com                                      39,531
// https://docs.microsoft.com/azure/devops                         75,837
// https://docs.microsoft.com/xamarin                              60,284
// https://docs.microsoft.com/system-center                        43,444
// https://docs.microsoft.com/enterprise-mobility-security         28,946
// https://docs.microsoft.com/microsoft-365                        43,278
// https://docs.microsoft.com/visualstudio                         31,414
// https://docs.microsoft.com/office                               42,292
// https://docs.microsoft.com/azure                               401,113
// https://docs.microsoft.com/graph                                46,831
// https://docs.microsoft.com/education                            25,098
// https://docs.microsoft.com/powershell                           58,173
// https://docs.microsoft.com/aspnet/core                          87,763
// https://docs.microsoft.com/sql                                  53,362

// Total bytes returned: 1,249,485
// Elapsed time:          00:00:02.7068725
```

См. также раздел

- WhenAny
- Асинхронное программирование с использованием ключевых слов Async и Await (C#)

С помощью Task.WhenAny можно запускать несколько задач одновременно и обрабатывать их по одной по мере завершения, а не в порядке их запуска.

В следующем примере используется запрос для создания коллекции задач. Каждая задача загружает содержимое указанного веб-сайта. В каждой итерации цикла while ожидаемый вызов WhenAny возвращает задачу из коллекции задач, которая первой завершает свою загрузку. Эта задача удаляется из коллекции и обрабатывается. Цикл выполняется до тех пор, пока в коллекции еще есть задачи.

Предварительные требования

Этот учебник можно выполнить с помощью одного из следующих вариантов:

- Visual Studio с установленной рабочей нагрузкой **Разработка классических приложений** .NET. Пакет SDK для .NET устанавливается автоматически при выборе этой рабочей нагрузки.
- Пакет SDK для .NET 5.0 с любым редактором кода, например Visual Studio Code.

Создание примера приложения

Создайте новое консольное приложение .NET Core для .NET 5.0 или .NET Core 3.1. Его можно создать с помощью команды dotnet new console или в Visual Studio. Откройте файл *Program.cs* в любом редакторе кода.

Замена операторов using

Замените существующие операторы using следующими объявлениями.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;
```

Добавить поля

Добавьте в определение класса `Program` следующие два поля.

```
static readonly HttpClient s_client = new HttpClient
{
    MaxResponseContentBufferSize = 1_000_000
};

static readonly IEnumerable<string> s_urlList = new string[]
{
    "https://docs.microsoft.com",
    "https://docs.microsoft.com/aspnet/core",
    "https://docs.microsoft.com/azure",
    "https://docs.microsoft.com/azure/devops",
    "https://docs.microsoft.com/dotnet",
    "https://docs.microsoft.com/dynamics365",
    "https://docs.microsoft.com/education",
    "https://docs.microsoft.com/enterprise-mobility-security",
    "https://docs.microsoft.com/gaming",
    "https://docs.microsoft.com/graph",
    "https://docs.microsoft.com/microsoft-365",
    "https://docs.microsoft.com/office",
    "https://docs.microsoft.com/powershell",
    "https://docs.microsoft.com/sql",
    "https://docs.microsoft.com/surface",
    "https://docs.microsoft.com/system-center",
    "https://docs.microsoft.com/visualstudio",
    "https://docs.microsoft.com/windows",
    "https://docs.microsoft.com/xamarin"
};
```

`HttpClient` предоставляет возможность отправлять HTTP-запросы и получать HTTP-ответы. `s_urlList` содержит все URL-адреса, которые планируется обработать приложением.

Обновление точки входа приложения

Главной точкой входа в консольное приложение является метод `Main`. Замените существующий метод следующим кодом.

```
static Task Main() => SumPageSizesAsync();
```

Обновленный метод `Main` теперь считается асинхронным методом main, который позволяет использовать асинхронную точку входа в исполняемом файле. Он выражается вызовом `SumPageSizesAsync`.

Создание метода асинхронного суммирования размеров страниц

Добавьте метод `Main` под методом `SumPageSizesAsync`.

```
static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    IEnumerable<Task<int>> downloadTasksQuery =
        from url in s_urlList
        select ProcessUrlAsync(url, s_client);

    List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

    int total = 0;
    while (downloadTasks.Any())
    {
        Task<int> finishedTask = await Task.WhenAny(downloadTasks);
        downloadTasks.Remove(finishedTask);
        total += await finishedTask;
    }

    stopwatch.Stop();

    Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
    Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
}
```

Метод начинается с создания экземпляра и запуска Stopwatch. Затем он включает запрос, который при выполнении создает коллекцию задач. Каждый вызов `ProcessUrlAsync` в следующем коде возвращает Task<TResult>, где `TResult` — целое число.

```
IEnumerable<Task<int>> downloadTasksQuery =
    from url in s_urlList
    select ProcessUrlAsync(url, s_client);
```

Из-за отложенного выполнения с помощью LINQ для запуска каждой задачи вызывается Enumerable.ToList.

```
List<Task<int>> downloadTasks = downloadTasksQuery.ToList();
```

Цикл `while` выполняет следующие действия для каждой задачи в коллекции.

1. Ожидает вызов `WhenAny` для определения первой задачи в коллекции, чтобы завершить ее загрузку.

   ```
   Task<int> finishedTask = await Task.WhenAny(downloadTasks);
   ```

2. Удаляет эту задачу из коллекции.

   ```
   downloadTasks.Remove(finishedTask);
   ```

3. Ожидает `finishedTask`, возвращаемый при вызове `ProcessUrlAsync`. Переменная `finishedTask` представляет собой Task<TResult>, где `TResult` — целое число. Задача уже завершена, но она ожидается для получения размера загруженного веб-сайта, как показано в следующем примере. В случае сбоя задачи `await` выдаст первое исключение дочернего элемента, хранящееся в `AggregateException`, в отличие от считывания свойства Task<TResult>.Result, которое выдаст `AggregateException`.

```
    total += await finishedTask;
```

Добавление метода обработки

Добавьте приведенный ниже метод `ProcessUrlAsync` после метода `SumPageSizesAsync`.

```
static async Task<int> ProcessUrlAsync(string url, HttpClient client)
{
    byte[] content = await client.GetByteArrayAsync(url);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}
```

Для любого заданного URL-адреса метод будет использовать экземпляр `client`, предоставленный для получения ответа в качестве `byte[]`. Длина возвращается после того, как URL-адрес и длина записываются в консоль.

Запустите проект несколько раз и убедитесь, что размеры скачанных файлов не всегда отображаются в одном и том же порядке.

Caution

Можно использовать `WhenAny` в цикле, как описано в примере, для решения проблем, которые включают небольшое число задач. Однако когда требуется обработка большого числа задач, другие методы будут более эффективны. Дополнительные сведения и примеры см. в разделе Обработка задач по мере их завершения.

Полный пример

Приведенный ниже код — это полный текст файла *Program.cs* для примера.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;

namespace ProcessTasksAsTheyFinish
{
    class Program
    {
        static readonly HttpClient s_client = new HttpClient
        {
            MaxResponseContentBufferSize = 1_000_000
        };

        static readonly IEnumerable<string> s_urlList = new string[]
        {
            "https://docs.microsoft.com",
            "https://docs.microsoft.com/aspnet/core",
            "https://docs.microsoft.com/azure",
            "https://docs.microsoft.com/azure/devops",
            "https://docs.microsoft.com/dotnet",
            "https://docs.microsoft.com/dynamics365",
            "https://docs.microsoft.com/education",
            "https://docs.microsoft.com/enterprise-mobility-security",
            "https://docs.microsoft.com/gaming",
            "https://docs.microsoft.com/graph",
            "https://docs.microsoft.com/microsoft-365",
```

```
            "https://docs.microsoft.com/office",
            "https://docs.microsoft.com/powershell",
            "https://docs.microsoft.com/sql",
            "https://docs.microsoft.com/surface",
            "https://docs.microsoft.com/system-center",
            "https://docs.microsoft.com/visualstudio",
            "https://docs.microsoft.com/windows",
            "https://docs.microsoft.com/xamarin"
        };

        static Task Main() => SumPageSizesAsync();

        static async Task SumPageSizesAsync()
        {
            var stopwatch = Stopwatch.StartNew();

            IEnumerable<Task<int>> downloadTasksQuery =
                from url in s_urlList
                select ProcessUrlAsync(url, s_client);

            List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

            int total = 0;
            while (downloadTasks.Any())
            {
                Task<int> finishedTask = await Task.WhenAny(downloadTasks);
                downloadTasks.Remove(finishedTask);
                total += await finishedTask;
            }

            stopwatch.Stop();

            Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
            Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
        }

        static async Task<int> ProcessUrlAsync(string url, HttpClient client)
        {
            byte[] content = await client.GetByteArrayAsync(url);
            Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

            return content.Length;
        }
    }
}
// Example output:
// https://docs.microsoft.com/windows                               25,513
// https://docs.microsoft.com/gaming                                30,705
// https://docs.microsoft.com/dotnet                                69,626
// https://docs.microsoft.com/dynamics365                           50,756
// https://docs.microsoft.com/surface                               35,519
// https://docs.microsoft.com                                       39,531
// https://docs.microsoft.com/azure/devops                          75,837
// https://docs.microsoft.com/xamarin                               60,284
// https://docs.microsoft.com/system-center                         43,444
// https://docs.microsoft.com/enterprise-mobility-security          28,946
// https://docs.microsoft.com/microsoft-365                         43,278
// https://docs.microsoft.com/visualstudio                          31,414
// https://docs.microsoft.com/office                                42,292
// https://docs.microsoft.com/azure                                401,113
// https://docs.microsoft.com/graph                                 46,831
// https://docs.microsoft.com/education                             25,098
// https://docs.microsoft.com/powershell                            58,173
// https://docs.microsoft.com/aspnet/core                           87,763
// https://docs.microsoft.com/sql                                   53,362

// Total bytes returned: 1,249,485
// Elapsed time:          00:00:02.7068725
```

См. также раздел

- WhenAny
- Асинхронное программирование с использованием ключевых слов Async и Await (C#)

Асинхронный доступ к файлам (C#)

18.12.2021 • 4 minutes to read

Для доступа к файлам можно использовать функцию Async. При использовании функции Async вы можете вызывать асинхронные методы без использования обратных вызовов или разделения вашего кода на множество методов или лямбда-выражений. Для выполнения последовательного кода асинхронно просто вызовите асинхронный метод вместо синхронного метода и добавьте несколько ключевых слов в код.

Можно рассмотреть следующие причины для добавления асинхронности для вызовов для доступа к файлам.

- Асинхронность делает приложения пользовательского интерфейса более отзывчивыми, потому что поток пользовательского интерфейса, который запускает операцию, может продолжать выполнять и другую работу. Если поток пользовательского интерфейса должен выполнять код, который занимает много времени (например, более 50 миллисекунд), пользовательский интерфейс можно приостановить до тех пор, пока не будет завершен ввод-вывод, и затем пользовательский интерфейс сможет снова обрабатывать ввод с клавиатуры, мыши и другие события.
- Асинхронность улучшает масштабируемость ASP.NET и других серверных приложений за счет уменьшения необходимости использования потоков. Если приложение использует выделенный поток на ответ и тысяча запросов приходит одновременно, тысяча потоков не потребуется. Асинхронные операции часто не нуждаются в пользовании потоком во время ожидания. Они пользуются существующим потоком завершения ввода-вывода короткое время в конце.
- Задержка операции доступа к файлу может быть очень низкой при текущих условиях, но может значительно увеличиться в будущем. Например, файл может быть перемещен на сервер через Интернет.
- Добавленные издержки при использовании функции Async являются малыми.
- Асинхронные задачи могут легко выполняться параллельно.

Использование соответствующих классов

В простых примерах в этом разделе демонстрируются File.WriteAllTextAsync и File.ReadAllTextAsync. Для конечного контроля над операциями файлового ввода-вывода используйте класс FileStream, содержащий параметр, который вызывает асинхронный ввод-вывод на уровне операционной системы. С помощью этого параметра можно избежать блокирования пула потоков во многих случаях. Чтобы включить этот параметр, необходимо добавить в вызов конструктора аргумент `useAsync=true` или `options=FileOptions.Asynchronous`.

Этот параметр нельзя использовать с классами StreamReader и StreamWriter, если вы открываете их напрямую (указав путь к файлу). При этом параметр можно использовать, если им предоставлен Stream, открытый классом FileStream. Асинхронные вызовы выполняются быстрее в приложениях пользовательского интерфейса, даже если поток в пуле потоков блокирован, поскольку поток пользовательского интерфейса не блокирован во время ожидания.

Запись текста

Следующие примеры записывают текст в файл. На каждой точке await происходит немедленный выход из метода. После завершения файлового ввода-вывода метод возобновляет работу с пункта, следующего за await. Модификатор async в определении методов требует наличия await в теле метода.

Простой пример

```
public async Task SimpleWriteAsync()
{
    string filePath = "simple.txt";
    string text = $"Hello World";

    await File.WriteAllTextAsync(filePath, text);
}
```

Пример конечного элемента управления

```
public async Task ProcessWriteAsync()
{
    string filePath = "temp.txt";
    string text = $"Hello World{Environment.NewLine}";

    await WriteTextAsync(filePath, text);
}

async Task WriteTextAsync(string filePath, string text)
{
    byte[] encodedText = Encoding.Unicode.GetBytes(text);

    using var sourceStream =
        new FileStream(
            filePath,
            FileMode.Create, FileAccess.Write, FileShare.None,
            bufferSize: 4096, useAsync: true);

    await sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
}
```

Первоначальная строка с оператором `await sourceStream.WriteAsync(encodedText, 0, encodedText.Length);` является сокращенной формой записи двух следующих операторов:

```
Task theTask = sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
await theTask;
```

Первый оператор возвращает задачу и вызывает запуск обработки файла. Вторая строка с await немедленно оставляет метод и возвращается в другую задачу. При окончании обработки файла выполнение возвращается в точку выполнения, которая следует за await.

Чтение текста

Следующий пример считывает текст из файла.

Простой пример

```
public async Task SimpleReadAsync()
{
    string filePath = "simple.txt";
    string text = await File.ReadAllTextAsync(filePath);

    Console.WriteLine(text);
}
```

Пример конечного элемента управления

Текст добавляется в буфер обмена, а затем, в данном случае, помещается в StringBuilder. В отличие от

предыдущего примера await выдаёт в результате значение. Метод ReadAsync возвращает Task<Int32>, поэтому по завершении операции оценка await выдает значение `Int32` `numRead`. Дополнительные сведения см. в разделе Асинхронные типы возвращаемых значений (C#).

```
public async Task ProcessReadAsync()
{
    try
    {
        string filePath = "temp.txt";
        if (File.Exists(filePath) != false)
        {
            string text = await ReadTextAsync(filePath);
            Console.WriteLine(text);
        }
        else
        {
            Console.WriteLine($"file not found: {filePath}");
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}

async Task<string> ReadTextAsync(string filePath)
{
    using var sourceStream =
        new FileStream(
            filePath,
            FileMode.Open, FileAccess.Read, FileShare.Read,
            bufferSize: 4096, useAsync: true);

    var sb = new StringBuilder();

    byte[] buffer = new byte[0x1000];
    int numRead;
    while ((numRead = await sourceStream.ReadAsync(buffer, 0, buffer.Length)) != 0)
    {
        string text = Encoding.Unicode.GetString(buffer, 0, numRead);
        sb.Append(text);
    }

    return sb.ToString();
}
```

Параллельный асинхронный ввод-вывод

В следующем примере показана параллельная обработка при записи 10 текстовых файлов.

Простой пример

```
public async Task SimpleParallelWriteAsync()
{
    string folder = Directory.CreateDirectory("tempfolder").Name;
    IList<Task> writeTaskList = new List<Task>();

    for (int index = 11; index <= 20; ++ index)
    {
        string fileName = $"file-{index:00}.txt";
        string filePath = $"{folder}/{fileName}";
        string text = $"In file {index}{Environment.NewLine}";

        writeTaskList.Add(File.WriteAllTextAsync(filePath, text));
    }

    await Task.WhenAll(writeTaskList);
}
```

Пример конечного элемента управления

Для каждого файла метод WriteAsync возвращает задачу, которая затем добавляется в список задач. Оператор `await Task.WhenAll(tasks);` существует и возобновляется в методе, как только завершается обработка файла для всех задач.

Пример закрывает все экземпляры FileStream в блоке `finally` после завершения всех задач. Если бы вместо этого каждый `FileStream` был бы создан в операторе `using`, то `FileStream` можно было бы удалить до завершения задачи.

Любое увеличение производительности зависит почти полностью от параллельной, а не асинхронной обработки. Преимущества асинхронности в том, что она не привязана к количеству потоков и не связана с потоком пользовательского интерфейса.

```
public async Task ProcessMulitpleWritesAsync()
{
    IList<FileStream> sourceStreams = new List<FileStream>();

    try
    {
        string folder = Directory.CreateDirectory("tempfolder").Name;
        IList<Task> writeTaskList = new List<Task>();

        for (int index = 1; index <= 10; ++ index)
        {
            string fileName = $"file-{index:00}.txt";
            string filePath = $"{folder}/{fileName}";

            string text = $"In file {index}{Environment.NewLine}";
            byte[] encodedText = Encoding.Unicode.GetBytes(text);

            var sourceStream =
                new FileStream(
                    filePath,
                    FileMode.Create, FileAccess.Write, FileShare.None,
                    bufferSize: 4096, useAsync: true);

            Task writeTask = sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
            sourceStreams.Add(sourceStream);

            writeTaskList.Add(writeTask);
        }

        await Task.WhenAll(writeTaskList);
    }
    finally
    {
        foreach (FileStream sourceStream in sourceStreams)
        {
            sourceStream.Close();
        }
    }
}
```

При использовании методов WriteAsync и ReadAsync можно указать CancellationToken, который позволяет отменить операцию в середине потока. Подробные сведения см. в статье Отмена в управляемых потоках.

См. также раздел

- Асинхронное программирование с использованием ключевых слов Async и Await (C#)
- Типы возвращаемых значений асинхронных операций (C#)

Атрибуты (C#)

18.12.2021 • 4 minutes to read

Атрибуты предоставляют мощное средство для связывания метаданных или декларативной информации с кодом (сборки, типы, методы, свойства и т. д.). Связав атрибут связан с сущностью программы, вы можете проверять этот атрибут во время выполнения, используя технику *отражения*. Подробнее см. в разделе Отражение (C#).

Атрибуты имеют следующие свойства.

- Атрибуты добавляют в программу метаданные. *Метаданные* — это сведения о типах, определенных в программе. Все сборки .NET содержат некоторый набор метаданных, описывающих типы и члены типов, определенные в этой сборке. Вы можете добавить пользовательские атрибуты, чтобы указать любую дополнительную информацию. Подробнее см. в разделе Создание настраиваемых атрибутов (C#).
- Вы можете применить один или несколько атрибутов ко всей сборке, к модулю или к более мелким элементам программы, например к классам и свойствам.
- Атрибуты могут принимать аргументы, так же как методы и свойства.
- Программа может проверить собственные метаданные или метаданные в других программах, используя отражение. Дополнительные сведения см. в разделе Обращение к атрибутам с помощью отражения (C#).

Использование атрибутов

Атрибуты можно использовать почти в любых объявлениях, но для каждого атрибута можно ограничить типы объявлений, в которых он является допустимым. Чтобы указать атрибут на C#, поместите имя атрибута в квадратных скобках ([]) над объявлением сущности, к которой он применяется.

В этом примере атрибут SerializableAttribute используется для применения определенной характеристики к классу:

```
[Serializable]
public class SampleClass
{
    // Objects of this type can be serialized.
}
```

Метод с атрибутом DllImportAttribute объявляется следующим образом:

```
[System.Runtime.InteropServices.DllImport("user32.dll")]
extern static void SampleMethod();
```

В объявлении можно разместить несколько атрибутов, как показано в следующем примере:

```
using System.Runtime.InteropServices;
```

```
void MethodA([In][Out] ref double x) { }
void MethodB([Out][In] ref double x) { }
void MethodC([In, Out] ref double x) { }
```

Некоторые атрибуты можно указать для одной сущности более одного раза. Пример такого многократно используемого атрибута — ConditionalAttribute:

```
[Conditional("DEBUG"), Conditional("TEST1")]
void TraceMethod()
{
    // ...
}
```

> **NOTE**
> По соглашению все имена атрибутов заканчиваются словом Attribute, чтобы отличать их от других элементов библиотек .NET. Но при использовании атрибута в коде этот суффикс можно не указывать. Например, обращение `[DllImport]` эквивалентно `[DllImportAttribute]`, хотя в библиотеке классов .NET этот атрибут имеет имя `DllImportAttribute`.

Параметры атрибутов

Многие атрибуты имеют параметры, которые могут быть позиционными, неименованными или именованными. Позиционные параметры должны указываться в определенном порядке и не могут опускаться. Именованные параметры являются необязательными и могут указываться в любом порядке. Сначала указываются позиционные параметры. Например, эти три атрибута являются эквивалентными:

```
[DllImport("user32.dll")]
[DllImport("user32.dll", SetLastError=false, ExactSpelling=false)]
[DllImport("user32.dll", ExactSpelling=false, SetLastError=false)]
```

Первый параметр (имя библиотеки DLL) является позиционным и всегда располагается первым, остальные параметры являются именованными. В этом примере оба именованных параметра имеют значение по умолчанию (false), и мы можем их опустить. Позиционные параметры соответствуют параметрам конструктора атрибутов. Именованные (или необязательные) параметры соответствуют свойствам или полям атрибута. Сведения о значениях параметров по умолчанию указываются в документации по каждому отдельному атрибуту.

Дополнительные сведения о разрешенных типах параметров см. в разделе Атрибуты в спецификации языка C#.

Целевые объекты атрибутов

Целевой объект атрибута — это сущность, к которой применяется атрибут. Например, атрибут можно применить к классу, отдельному методу или ко всей сборке. По умолчанию атрибут применяется к тому элементу, который следует за ним. Но у вас есть возможность явным образом указать, например, что атрибут применяется к методу, параметру или возвращаемому значению.

Чтобы явным образом определить целевой объект атрибута, используйте следующий синтаксис:

```
[target : attribute-list]
```

Возможные значения `target` перечислены в следующей таблице.

ЦЕЛЕВОЕ ЗНАЧЕНИЕ	ПРИМЕНЕНИЕ
`assembly`	Вся сборка

ЦЕЛЕВОЕ ЗНАЧЕНИЕ	ПРИМЕНЕНИЕ
`module`	Модуль текущей сборки
`field`	Поле в классе или структуре
`event`	событие
`method`	Метод либо методы доступа к свойствам `get` и `set`
`param`	Параметры метода или параметры метода доступа `set`
`property`	Свойство.
`return`	Возвращаемое значение метода, индексатора свойства или метода доступа к свойствам `get`
`type`	Структура, класс, интерфейс, перечисление или делегат

Следует указать целевое значение `field`, чтобы применить атрибут к резервной переменной, созданной для автоматически реализуемого свойства.

Следующий пример демонстрирует, как применить атрибуты к сборкам и модулям. Дополнительные сведения см. в разделе Общие атрибуты (C#).

```
using System;
using System.Reflection;
[assembly: AssemblyTitleAttribute("Production assembly 4")]
[module: CLSCompliant(true)]
```

В следующем примере показано, как применять атрибуты к методам, параметрам и возвращаемым значениям методов в C#.

```
// default: applies to method
[ValidatedContract]
int Method1() { return 0; }

// applies to method
[method: ValidatedContract]
int Method2() { return 0; }

// applies to parameter
int Method3([ValidatedContract] string contract) { return 0; }

// applies to return value
[return: ValidatedContract]
int Method4() { return 0; }
```

NOTE

Вне зависимости от целевых объектов, для которых действует атрибут `ValidatedContract`, необходимо задать целевой объект для `return`, даже если атрибут `ValidatedContract` назначен только возвращаемым значениям. Другими словами, компилятор не будет использовать сведения `AttributeUsage` для разрешения конфликтов между неоднозначными целевыми объектами атрибута. Подробнее см. в разделе AttributeUsage (C#).

Популярные методы применения атрибутов

В следующем списке перечислены несколько распространенных применений для атрибутов.

- Указание для методов в веб-службах атрибута `WebMethod`, который обозначает, что метод должен вызываться по протоколу SOAP. Для получения дополнительной информации см. WebMethodAttribute.
- Описание способов упаковки параметров методов при взаимодействии с машинным кодом. Для получения дополнительной информации см. MarshalAsAttribute.
- Описание свойств COM для классов, методов и интерфейсов.
- Вызов неуправляемого кода с помощью класса DllImportAttribute.
- Указание для сборки таких параметров, как заголовок, версия, описание или товарный знак.
- Указание членов класса, которые будут сериализованы при сохранении класса.
- Описание правил сопоставления членов класса с XML-узлами при XML-сериализации.
- Описание требований безопасности для методов.
- Указание характеристик, используемых для обеспечения безопасности.
- Управление оптимизацией для JIT-компилятора, сохраняя при этом простоту отладки кода.
- Получение сведений об объекте, вызывающем метод.

Связанные разделы

Дополнительные сведения можно найти в разделе

- Создание настраиваемых атрибутов (C#)
- Обращение к атрибутам с помощью отражения (C#)
- Практическое руководство. Создание объединения C/C++ с помощью атрибутов (C#)
- Общие атрибуты (C#)
- Сведения о вызывающем объекте (C#)

См. также

- Руководство по программированию на C#
- Отражение (C#)
- Атрибуты
- Использование атрибутов в C#

Создание настраиваемых атрибутов (C#)

18.12.2021 • 2 minutes to read

Собственные настраиваемые атрибуты можно создать, определив класс атрибута, то есть класс, прямо или косвенно наследующий от Attribute, который упрощает задание определений атрибутов в метаданных. Предположим, что требуется пометить тип тегом с именем программиста, который его разработал. Вы можете определить класс настраиваемых атрибутов `Author`:

```
[System.AttributeUsage(System.AttributeTargets.Class |
                       System.AttributeTargets.Struct)
]
public class AuthorAttribute : System.Attribute
{
    private string name;
    public double version;

    public AuthorAttribute(string name)
    {
        this.name = name;
        version = 1.0;
    }
}
```

Имя класса `AuthorAttribute` является именем атрибута, `Author` плюс суффикс `Attribute`. Он является производным от `System.Attribute`, поэтому это класс настраиваемых атрибутов. Параметры конструктора являются позиционными параметрами настраиваемого атрибута. В этом примере `name` является позиционным параметром. Все открытые поля или свойства, доступные для чтения и записи, являются именованными параметрами. В этом случае `version` — единственный именованный параметр. Обратите внимание на использование атрибута `AttributeUsage`, делающего атрибут `Author` допустимым только для класса и объявлений `struct`.

Этот атрибут можно использовать следующим образом:

```
[Author("P. Ackerman", version = 1.1)]
class SampleClass
{
    // P. Ackerman's code goes here...
}
```

`AttributeUsage` имеет именованный параметр, `AllowMultiple`, с помощью которого можно задавать для настраиваемого атрибута однократное или многократное использование. В следующем примере кода создается многократно используемый атрибут.

```
[System.AttributeUsage(System.AttributeTargets.Class |
                       System.AttributeTargets.Struct,
                       AllowMultiple = true)  // multiuse attribute
]
public class AuthorAttribute : System.Attribute
```

В следующем примере кода несколько атрибутов одного типа применяются к классу.

```
[Author("P. Ackerman", version = 1.1)]
[Author("R. Koch", version = 1.2)]
class SampleClass
{
    // P. Ackerman's code goes here...
    // R. Koch's code goes here...
}
```

См. также

- System.Reflection
- Руководство по программированию на C#
- Написание настраиваемых атрибутов
- Отражение (C#)
- Атрибуты (C#)
- Обращение к атрибутам с помощью отражения (C#)
- AttributeUsage (C#)

Обращение к атрибутам с помощью отражения (C#)

18.12.2021 • 2 minutes to read

Возможность определения настраиваемых атрибутов и их помещения в собственный исходный код не будет настолько значимой без наличия способа извлечения этих сведений и работы с ними. Отражение позволяет извлекать сведения, определенные с настраиваемыми атрибутами. Основным методом выступает `GetCustomAttributes`, который возвращает массив объектов, являющихся эквивалентами времени выполнения атрибутов исходного кода. Для этого метода существует несколько перегруженных версий. Для получения дополнительной информации см. Attribute.

Спецификация атрибута, например:

```
[Author("P. Ackerman", version = 1.1)]
class SampleClass
```

концептуально эквивалентна следующему коду:

```
Author anonymousAuthorObject = new Author("P. Ackerman");
anonymousAuthorObject.version = 1.1;
```

Однако код не выполняется до тех пор, пока у `SampleClass` не будут запрошены атрибуты. Вызов `GetCustomAttributes` в `SampleClass` приведет к тому, что объект `Author` будет создан и инициализирован так, как показано выше. Если класс имеет другие атрибуты, другие объекты атрибутов создаются аналогично. `GetCustomAttributes` затем возвращает объект `Author` и все другие объекты атрибутов в массиве. Потом можно пройти по этому массиву, определить в зависимости от типа каждого элемента массива какие атрибуты были применены и извлечь сведения из объектов атрибутов.

Пример

Ниже приведен полный пример. Определяется настраиваемый атрибут, который применяется к нескольким сущностям и извлекается через отражение.

```
// Multiuse attribute.
[System.AttributeUsage(System.AttributeTargets.Class |
                       System.AttributeTargets.Struct,
                       AllowMultiple = true)  // Multiuse attribute.
]
public class Author : System.Attribute
{
    string name;
    public double version;

    public Author(string name)
    {
        this.name = name;

        // Default value.
        version = 1.0;
    }

    public string GetName()
    {
```

```
    {
        return name;
    }
}

// Class with the Author attribute.
[Author("P. Ackerman")]
public class FirstClass
{
    // ...
}

// Class without the Author attribute.
public class SecondClass
{
    // ...
}

// Class with multiple Author attributes.
[Author("P. Ackerman"), Author("R. Koch", version = 2.0)]
public class ThirdClass
{
    // ...
}

class TestAuthorAttribute
{
    static void Test()
    {
        PrintAuthorInfo(typeof(FirstClass));
        PrintAuthorInfo(typeof(SecondClass));
        PrintAuthorInfo(typeof(ThirdClass));
    }

    private static void PrintAuthorInfo(System.Type t)
    {
        System.Console.WriteLine("Author information for {0}", t);

        // Using reflection.
        System.Attribute[] attrs = System.Attribute.GetCustomAttributes(t);  // Reflection.

        // Displaying output.
        foreach (System.Attribute attr in attrs)
        {
            if (attr is Author)
            {
                Author a = (Author)attr;
                System.Console.WriteLine("   {0}, version {1:f}", a.GetName(), a.version);
            }
        }
    }
}
/* Output:
    Author information for FirstClass
       P. Ackerman, version 1.00
    Author information for SecondClass
    Author information for ThirdClass
       R. Koch, version 2.00
       P. Ackerman, version 1.00
*/
```

См. также

- System.Reflection
- Attribute
- Руководство по программированию на C#

- Извлечение информации, сохраненной в атрибуте
- Отражение (C#)
- Атрибуты (C#)
- Создание настраиваемых атрибутов (C#)

Практическое руководство. Создание объединения C/C++ с помощью атрибутов (C#)

18.12.2021 • 2 minutes to read

С помощью атрибутов можно настраивать расположение структур в памяти. Например, можно создать так называемое объединение в C/C++ с помощью атрибутов `StructLayout(LayoutKind.Explicit)` и `FieldOffset`.

Примеры

В этом сегменте кода все поля объединения `TestUnion` начинаются с одного адреса в памяти.

```
// Add a using directive for System.Runtime.InteropServices.

[System.Runtime.InteropServices.StructLayout(LayoutKind.Explicit)]
struct TestUnion
{
    [System.Runtime.InteropServices.FieldOffset(0)]
    public int i;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public double d;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public char c;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public byte b;
}
```

Ниже приведен еще один пример, в котором поля начинаются с разных явно заданных адресов.

```
// Add a using directive for System.Runtime.InteropServices.

[System.Runtime.InteropServices.StructLayout(LayoutKind.Explicit)]
struct TestExplicit
{
    [System.Runtime.InteropServices.FieldOffset(0)]
    public long lg;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public int i1;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public int i2;

    [System.Runtime.InteropServices.FieldOffset(8)]
    public double d;

    [System.Runtime.InteropServices.FieldOffset(12)]
    public char c;

    [System.Runtime.InteropServices.FieldOffset(14)]
    public byte b;
}
```

Два целочисленных поля `i1` и `i2` используют те же адреса памяти, что и `lg`. Такое управление расположением структуры полезно при использовании вызова неуправляемого кода.

См. также

- System.Reflection
- Attribute
- Руководство по программированию на C#
- Атрибуты
- Отражение (C#)
- Атрибуты (C#)
- Создание настраиваемых атрибутов (C#)
- Обращение к атрибутам с помощью отражения (C#)

Коллекции (C#)

18.12.2021 • 12 minutes to read

Во многих приложениях требуется создавать группы связанных объектов и управлять ими. Существует два способа группировки объектов: создать массив объектов и создать коллекцию.

Массивы удобнее всего использовать для создания фиксированного числа строго типизированных объектов и работы с ними. Информацию о массивах см. в разделе Массивы.

Коллекции предоставляют более гибкий способ работы с группами объектов. В отличие от массивов, коллекция, с которой вы работаете, может расти или уменьшаться динамически при необходимости. Некоторые коллекции допускают назначение ключа любому объекту, который добавляется в коллекцию, чтобы в дальнейшем можно было быстро извлечь связанный с ключом объект из коллекции.

Коллекция является классом, поэтому необходимо объявить экземпляр класса перед добавлением в коллекцию элементов.

Если коллекция содержит элементы только одного типа данных, можно использовать один из классов в пространстве имен System.Collections.Generic. Универсальная коллекция обеспечивает строгую типизацию, так что в нее нельзя добавить другие типы данных. При извлечении элемента из универсальной коллекции не нужно определять или преобразовывать его тип данных.

> **NOTE**
> Для примеров в этом разделе включите директивы using для пространств имен `System.Collections.Generic` и `System.Linq`.

Содержание раздела

- Использование простой коллекции
- Виды коллекций
  - Классы System.Collections.Generic
  - Классы System.Collections.Concurrent
  - Классы System.Collections
- Реализация коллекции пар "ключ-значение"
- Использование LINQ для доступа к коллекции
- Сортировка коллекции
- Определение настраиваемой коллекции
- Итераторы

Использование простой коллекции

В примерах этого раздела используется универсальный класс List<T>, который позволяет работать со строго типизированными списками объектов.

В приведенном ниже примере создается список строк, а затем выполняется перебор строк с помощью оператора foreach.

```
// Create a list of strings.
var salmons = new List<string>();
salmons.Add("chinook");
salmons.Add("coho");
salmons.Add("pink");
salmons.Add("sockeye");

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook coho pink sockeye
```

Если содержимое коллекции известно заранее, для ее инициализации можно использовать *инициализатор коллекции*. Дополнительные сведения см. в разделе Инициализаторы объектов и коллекций.

Следующий пример аналогичен предыдущему за исключением того, что для добавления элементов в коллекцию используется инициализатор коллекции.

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook coho pink sockeye
```

Для перебора коллекции можно использовать оператор for вместо оператора `foreach`. Для этого доступ к элементам коллекции осуществляется по позиции индекса. Индекс элементов начинается с 0 и заканчивается числом, равным количеству элементов минус 1.

В приведенном ниже примере выполняется перебор элементов коллекции с помощью оператора `for` вместо `foreach`.

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

for (var index = 0; index < salmons.Count; index++)
{
    Console.Write(salmons[index] + " ");
}
// Output: chinook coho pink sockeye
```

В приведенном ниже примере элемент удаляется из коллекции путем указания удаляемого объекта.

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

// Remove an element from the list by specifying
// the object.
salmons.Remove("coho");

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook pink sockeye
```

В приведенном ниже примере удаляются элементы из универсального списка. Вместо оператора `foreach` используется оператор `for`, выполняющий перебор в порядке убывания. Связано это с тем, что в результате работы метода RemoveAt элементы, следующие за удаленным элементом, получают меньшее значение индекса.

```
var numbers = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Remove odd numbers.
for (var index = numbers.Count - 1; index >= 0; index--)
{
    if (numbers[index] % 2 == 1)
    {
        // Remove the element by specifying
        // the zero-based index in the list.
        numbers.RemoveAt(index);
    }
}

// Iterate through the list.
// A lambda expression is placed in the ForEach method
// of the List(T) object.
numbers.ForEach(
    number => Console.Write(number + " "));
// Output: 0 2 4 6 8
```

Для типа элементов в List<T> можно также определить собственный класс. В приведенном ниже примере класс `Galaxy`, который используется объектом List<T>, определен в коде.

```
private static void IterateThroughList()
{
    var theGalaxies = new List<Galaxy>
        {
            new Galaxy() { Name="Tadpole", MegaLightYears=400},
            new Galaxy() { Name="Pinwheel", MegaLightYears=25},
            new Galaxy() { Name="Milky Way", MegaLightYears=0},
            new Galaxy() { Name="Andromeda", MegaLightYears=3}
        };

    foreach (Galaxy theGalaxy in theGalaxies)
    {
        Console.WriteLine(theGalaxy.Name + "  " + theGalaxy.MegaLightYears);
    }

    // Output:
    //  Tadpole  400
    //  Pinwheel  25
    //  Milky Way  0
    //  Andromeda  3
}

public class Galaxy
{
    public string Name { get; set; }
    public int MegaLightYears { get; set; }
}
```

Виды коллекций

Многие типовые коллекции предоставляются платформой .NET. Каждый тип коллекции предназначен для определенной цели.

В этом разделе описываются следующие часто используемые классы коллекций:

- Классы System.Collections.Generic
- Классы System.Collections.Concurrent
- Классы System.Collections

Классы System.Collections.Generic

Универсальную коллекцию можно создать, используя один из классов в пространстве имен System.Collections.Generic. Универсальная коллекция применяется в том случае, если все элементы в коллекции имеют одинаковый тип данных. Универсальная коллекция обеспечивает строгую типизацию, позволяя добавлять данные только необходимого типа.

В таблице ниже перечислены некоторые из часто используемых классов пространства имен System.Collections.Generic.

КЛАСС	ОПИСАНИЕ
Dictionary<TKey,TValue>	Предоставляет коллекцию пар «ключ-значение», которые упорядочены по ключу.
List<T>	Представляет список объектов, доступных по индексу. Предоставляет методы для поиска по списку, его сортировки и изменения.

КЛАСС	ОПИСАНИЕ
Queue<T>	Представляет коллекцию объектов, которая обслуживается в порядке поступления (FIFO).
SortedList<TKey,TValue>	Представляет коллекцию пар "ключ-значение", упорядоченных по ключу на основе реализации IComparer<T>.
Stack<T>	Представляет коллекцию объектов, которая обслуживается в обратном порядке (LIFO).

Дополнительные сведения см. в разделе Часто используемые типы коллекций, Выбор класса коллекции и System.Collections.Generic.

Классы System.Collections.Concurrent

В .NET Framework 4 или более поздних версиях коллекции пространства имен System.Collections.Concurrent предоставляют эффективные потокобезопасные операции для доступа к элементам коллекции из нескольких потоков.

Классы пространства имен System.Collections.Concurrent следует использовать вместо соответствующих типов пространств имен System.Collections.Generic и System.Collections, если несколько потоков параллельно обращаются к такой коллекции. Дополнительные сведения см. в статьях Потокобезопасные коллекции и System.Collections.Concurrent.

Некоторые из классов, входящих в пространство имен System.Collections.Concurrent, — это BlockingCollection<T>, ConcurrentDictionary<TKey,TValue>, ConcurrentQueue<T> и ConcurrentStack<T>.

Классы System.Collections

Классы в пространстве имен System.Collections хранят элементы не в виде конкретно типизированных объектов, а как объекты типа `Object`.

Везде, где это возможно, следует использовать универсальные коллекции пространства имен System.Collections.Generic или пространства имен System.Collections.Concurrent вместо устаревших типов пространства имен `System.Collections`.

В следующей таблице перечислены некоторые из часто используемых классов пространства имен `System.Collections`:

КЛАСС	ОПИСАНИЕ
ArrayList	Представляет массив объектов, размер которого динамически увеличивается по мере необходимости.
Hashtable	Представляет коллекцию пар «ключ-значение», которые упорядочены по хэш-коду ключа.
Queue	Представляет коллекцию объектов, которая обслуживается в порядке поступления (FIFO).
Stack	Представляет коллекцию объектов, которая обслуживается в обратном порядке (LIFO).

Пространство имен System.Collections.Specialized предоставляет специализированные и строго типизированные классы коллекций, такие как коллекции строк, связанные списки и гибридные словари.

Реализация коллекции пар «ключ-значение»

Универсальная коллекция Dictionary<TKey,TValue> позволяет получить доступ к элементам коллекции с помощью ключа каждого элемента. Каждый элемент, добавляемый в словарь, состоит из значения и связанного с ним ключа. Извлечение значения по его ключу происходит быстро, так как класс `Dictionary` реализован как хэш-таблица.

В приведенном ниже примере создается коллекция `Dictionary` и выполняется перебор словаря с помощью оператора `foreach`.

```
private static void IterateThruDictionary()
{
    Dictionary<string, Element> elements = BuildDictionary();

    foreach (KeyValuePair<string, Element> kvp in elements)
    {
        Element theElement = kvp.Value;

        Console.WriteLine("key: " + kvp.Key);
        Console.WriteLine("values: " + theElement.Symbol + " " +
            theElement.Name + " " + theElement.AtomicNumber);
    }
}

private static Dictionary<string, Element> BuildDictionary()
{
    var elements = new Dictionary<string, Element>();

    AddToDictionary(elements, "K", "Potassium", 19);
    AddToDictionary(elements, "Ca", "Calcium", 20);
    AddToDictionary(elements, "Sc", "Scandium", 21);
    AddToDictionary(elements, "Ti", "Titanium", 22);

    return elements;
}

private static void AddToDictionary(Dictionary<string, Element> elements,
    string symbol, string name, int atomicNumber)
{
    Element theElement = new Element();

    theElement.Symbol = symbol;
    theElement.Name = name;
    theElement.AtomicNumber = atomicNumber;

    elements.Add(key: theElement.Symbol, value: theElement);
}

public class Element
{
    public string Symbol { get; set; }
    public string Name { get; set; }
    public int AtomicNumber { get; set; }
}
```

Чтобы вместо этого использовать инициализатор коллекции для создания коллекции `Dictionary`, можно заменить методы `BuildDictionary` и `AddToDictionary` приведенным ниже методом.

```
private static Dictionary<string, Element> BuildDictionary2()
{
    return new Dictionary<string, Element>
    {
        {"K",
            new Element() { Symbol="K", Name="Potassium", AtomicNumber=19}},
        {"Ca",
            new Element() { Symbol="Ca", Name="Calcium", AtomicNumber=20}},
        {"Sc",
            new Element() { Symbol="Sc", Name="Scandium", AtomicNumber=21}},
        {"Ti",
            new Element() { Symbol="Ti", Name="Titanium", AtomicNumber=22}}
    };
}
```

В приведенном ниже примере используется метод ContainsKey и свойство Item[] `Dictionary` для быстрого поиска элемента по ключу. Свойство `Item` позволяет получить доступ к элементу в коллекции `elements` с помощью кода `elements[symbol]` в C#.

```
private static void FindInDictionary(string symbol)
{
    Dictionary<string, Element> elements = BuildDictionary();

    if (elements.ContainsKey(symbol) == false)
    {
        Console.WriteLine(symbol + " not found");
    }
    else
    {
        Element theElement = elements[symbol];
        Console.WriteLine("found: " + theElement.Name);
    }
}
```

В приведенном ниже примере вместо этого используется метод TryGetValue для быстрого поиска элемента по ключу.

```
private static void FindInDictionary2(string symbol)
{
    Dictionary<string, Element> elements = BuildDictionary();

    Element theElement = null;
    if (elements.TryGetValue(symbol, out theElement) == false)
        Console.WriteLine(symbol + " not found");
    else
        Console.WriteLine("found: " + theElement.Name);
}
```

Использование LINQ для доступа к коллекции

Для доступа к коллекции можно использовать язык LINQ. Запросы LINQ обеспечивают возможности фильтрации, упорядочения и группировки. Дополнительные сведения см. в разделе Приступая к работе с LINQ в C#.

В приведенном ниже примере выполняется запрос LINQ применительно к универсальной коллекции `List`. Запрос LINQ возвращает другую коллекцию, содержащую результаты.

```
private static void ShowLINQ()
{
    List<Element> elements = BuildList();

    // LINQ Query.
    var subset = from theElement in elements
                 where theElement.AtomicNumber < 22
                 orderby theElement.Name
                 select theElement;

    foreach (Element theElement in subset)
    {
        Console.WriteLine(theElement.Name + " " + theElement.AtomicNumber);
    }

    // Output:
    //  Calcium 20
    //  Potassium 19
    //  Scandium 21
}

private static List<Element> BuildList()
{
    return new List<Element>
    {
        { new Element() { Symbol="K", Name="Potassium", AtomicNumber=19}},
        { new Element() { Symbol="Ca", Name="Calcium", AtomicNumber=20}},
        { new Element() { Symbol="Sc", Name="Scandium", AtomicNumber=21}},
        { new Element() { Symbol="Ti", Name="Titanium", AtomicNumber=22}}
    };
}

public class Element
{
    public string Symbol { get; set; }
    public string Name { get; set; }
    public int AtomicNumber { get; set; }
}
```

Сортировка коллекции

Приведенный ниже пример демонстрирует процедуру сортировки коллекции. В примере сортируются экземпляры класса `Car`, которые хранятся в List<T>. Класс `Car` реализует интерфейс IComparable<T>, который требует реализации метода CompareTo.

Каждый вызов метода CompareTo выполняет одно сравнение, используемое для сортировки. Написанный пользователем код в методе `CompareTo` возвращает значение для каждого сравнения текущего объекта с другим объектом. Возвращаемое значение меньше нуля, если текущий объект меньше другого объекта, больше нуля, если текущий объект больше другого объекта, и равняется нулю, если объекты равны. Это позволяет определить в коде условия для отношения «больше», «меньше» и «равно».

В методе `ListCars` оператор `cars.Sort()` сортирует список. Этот вызов метода SortList<T> приводит к тому, что метод `CompareTo` вызывается автоматически для объектов `Car` в `List`.

```
private static void ListCars()
{
    var cars = new List<Car>
    {
        { new Car() { Name = "car1", Color = "blue", Speed = 20}},
        { new Car() { Name = "car2", Color = "red", Speed = 50}},
        { new Car() { Name = "car3", Color = "green", Speed = 10}},
        { new Car() { Name = "car4", Color = "blue", Speed = 50}},
```

```
        { new Car() { Name = "car5", Color = "blue", Speed = 30}},
        { new Car() { Name = "car6", Color = "red", Speed = 60}},
        { new Car() { Name = "car7", Color = "green", Speed = 50}}
    };

    // Sort the cars by color alphabetically, and then by speed
    // in descending order.
    cars.Sort();

    // View all of the cars.
    foreach (Car thisCar in cars)
    {
        Console.Write(thisCar.Color.PadRight(5) + " ");
        Console.Write(thisCar.Speed.ToString() + " ");
        Console.Write(thisCar.Name);
        Console.WriteLine();
    }

    // Output:
    //  blue  50 car4
    //  blue  30 car5
    //  blue  20 car1
    //  green 50 car7
    //  green 10 car3
    //  red   60 car6
    //  red   50 car2
}

public class Car : IComparable<Car>
{
    public string Name { get; set; }
    public int Speed { get; set; }
    public string Color { get; set; }

    public int CompareTo(Car other)
    {
        // A call to this method makes a single comparison that is
        // used for sorting.

        // Determine the relative order of the objects being compared.
        // Sort by color alphabetically, and then by speed in
        // descending order.

        // Compare the colors.
        int compare;
        compare = String.Compare(this.Color, other.Color, true);

        // If the colors are the same, compare the speeds.
        if (compare == 0)
        {
            compare = this.Speed.CompareTo(other.Speed);

            // Use descending order for speed.
            compare = -compare;
        }

        return compare;
    }
}
```

Определение настраиваемой коллекции

Вы можете определить коллекцию, реализовав интерфейс IEnumerable<T> или IEnumerable.

Хотя можно определить настраиваемую коллекцию, обычно лучше использовать коллекции, входящие в

.NET, которые описаны в разделе Виды коллекций ранее в этой статье.

В приведенном ниже примере определяется настраиваемый класс коллекции с именем `AllColors`. Этот класс реализует интерфейс IEnumerable, который требует реализации метода GetEnumerator.

Метод `GetEnumerator` возвращает экземпляр класса `ColorEnumerator`. Класс `ColorEnumerator` реализует интерфейс IEnumerator, который требует реализации свойства Current, метода MoveNext и метода Reset.

```
private static void ListColors()
{
    var colors = new AllColors();

    foreach (Color theColor in colors)
    {
        Console.Write(theColor.Name + " ");
    }
    Console.WriteLine();
    // Output: red blue green
}

// Collection class.
public class AllColors : System.Collections.IEnumerable
{
    Color[] _colors =
    {
        new Color() { Name = "red" },
        new Color() { Name = "blue" },
        new Color() { Name = "green" }
    };

    public System.Collections.IEnumerator GetEnumerator()
    {
        return new ColorEnumerator(_colors);

        // Instead of creating a custom enumerator, you could
        // use the GetEnumerator of the array.
        //return _colors.GetEnumerator();
    }

    // Custom enumerator.
    private class ColorEnumerator : System.Collections.IEnumerator
    {
        private Color[] _colors;
        private int _position = -1;

        public ColorEnumerator(Color[] colors)
        {
            _colors = colors;
        }

        object System.Collections.IEnumerator.Current
        {
            get
            {
                return _colors[_position];
            }
        }

        bool System.Collections.IEnumerator.MoveNext()
        {
            _position++;
            return (_position < _colors.Length);
        }

        void System.Collections.IEnumerator.Reset()
        {
            _position = -1;
```

```
        }
    }
}

// Element class.
public class Color
{
    public string Name { get; set; }
}
```

Iterators

Итератор используется для выполнения настраиваемого перебора коллекции. Итератор может быть методом или методом доступа `get`. Итератор использует оператор yield return для возврата всех элементов коллекции по одному за раз.

Итератор вызывается с помощью оператора foreach. Каждая итерация цикла `foreach` вызывает итератор. При достижении оператора `yield return` в итераторе возвращается выражение, и текущее расположение в коде сохраняется. При следующем вызове итератора выполнение возобновляется с этого места.

Дополнительные сведения см. в разделе Итераторы (C#).

В приведенном ниже примере используется метод-итератор. Метод итератора содержит оператор `yield return`, находящийся внутри цикла `for`. В методе `ListEvenNumbers` каждая итерация тела оператора `foreach` создает вызов метода-итератора, который переходит к следующему оператору `yield return`.

```
private static void ListEvenNumbers()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.Write(number.ToString() + " ");
    }
    Console.WriteLine();
    // Output: 6 8 10 12 14 16 18
}

private static IEnumerable<int> EvenSequence(
    int firstNumber, int lastNumber)
{
    // Yield even numbers in the range.
    for (var number = firstNumber; number <= lastNumber; number++)
    {
        if (number % 2 == 0)
        {
            yield return number;
        }
    }
}
```

См. также

- Инициализаторы объектов и коллекций
- Основные понятия программирования (C#)
- Оператор Option Strict
- LINQ to Objects (C#)
- Parallel LINQ (PLINQ)

- Коллекции и структуры данных
- Выбор класса коллекции
- Сравнение и сортировка в коллекциях
- Когда следует использовать универсальные коллекции
- Инструкции итерации

Ковариация и контравариантность (C#)

18.12.2021 • 2 minutes to read

В C# ковариация и контрвариантность позволяют использовать неявное преобразование ссылок для типов массивов и делегатов, а также для аргументов универсального типа. Ковариация сохраняет совместимость присваивания, а при контрвариантности присваивание начинает работать противоположным образом.

Следующий код показывает разницу между совместимостью присваивания, ковариацией и контрвариантностью.

```
// Assignment compatibility.
string str = "test";
// An object of a more derived type is assigned to an object of a less derived type.
object obj = str;

// Covariance.
IEnumerable<string> strings = new List<string>();
// An object that is instantiated with a more derived type argument
// is assigned to an object instantiated with a less derived type argument.
// Assignment compatibility is preserved.
IEnumerable<object> objects = strings;

// Contravariance.
// Assume that the following method is in the class:
// static void SetObject(object o) { }
Action<object> actObject = SetObject;
// An object that is instantiated with a less derived type argument
// is assigned to an object instantiated with a more derived type argument.
// Assignment compatibility is reversed.
Action<string> actString = actObject;
```

Ковариация для массивов позволяет неявно преобразовать массив более производного типа в массив менее производного типа. Но эта операция не является типобезопасной, как показано в следующем примере кода.

```
object[] array = new String[10];
// The following statement produces a run-time exception.
// array[0] = 10;
```

Поддержка ковариации и контрвариантности для групп методов позволяет сопоставить сигнатуры методов с типами делегатов. За счет этого вы можете назначать делегатам не только методы с совпадающими сигнатурами, но и методы, которые возвращают более производные типы (ковариация) или принимают параметры с менее производными типами (контрвариантность), чем задает тип делегата. Дополнительные сведения см. в разделах Вариативность в делегатах (C#) и Использование вариативности в делегатах (C#).

В следующем примере кода демонстрируется поддержка ковариации и контрвариантности для групп методов.

```
static object GetObject() { return null; }
static void SetObject(object obj) { }

static string GetString() { return ""; }
static void SetString(string str) { }

static void Test()
{
    // Covariance. A delegate specifies a return type as object,
    // but you can assign a method that returns a string.
    Func<object> del = GetString;

    // Contravariance. A delegate specifies a parameter type as string,
    // but you can assign a method that takes an object.
    Action<string> del2 = SetObject;
}
```

В .NET Framework 4 и более поздних версиях язык C# поддерживает ковариацию и контрвариантность для универсальных интерфейсов и делегатов, а также позволяет выполнять неявное преобразование параметров универсального типа. Дополнительные сведения см. в разделах Вариативность в универсальных интерфейсах (C#) и Вариативность в делегатах (C#).

В следующем примере кода показано неявное преобразование ссылок для универсальных интерфейсов.

```
IEnumerable<String> strings = new List<String>();
IEnumerable<Object> objects = strings;
```

Универсальный интерфейс или делегат называется *вариантным*, если его универсальные параметры объявлены ковариантными или контрвариантными. C# позволяет вам создавать собственные вариантные интерфейсы и делегаты. Дополнительные сведения см. в разделах Создание вариантных универсальных интерфейсов (C#) и Вариативность в делегатах (C#).

См. также

ЗАГОЛОВОК	ОПИСАНИЕ
Вариативность в универсальных интерфейсах (C#)	В этом разделе описываются ковариантность и контрвариантность в универсальных интерфейсах, а также представлен список вариативных универсальных интерфейсов в .NET.
Создание вариантных универсальных интерфейсов (C#)	Узнайте, как создавать ваши собственные вариантные интерфейсы.
Использование вариативности в интерфейсах для универсальных коллекций (C#)	Узнайте, как использовать поддержку ковариации и контрвариантности в интерфейсах IEnumerable<T> и IComparable<T> для многократного использования кода.
Вариативность в делегатах (C#)	В этом разделе описываются ковариантность и контрвариантность в универсальных и неуниверсальных делегатах, а также представлен список вариантных универсальных делегатов в .NET.
Использование вариативности в делегатах (C#)	Узнайте, как использовать поддержку ковариации и контрвариантности в неуниверсальных делегатах для сопоставления сигнатур методов с типами делегатов.

ЗАГОЛОВОК	ОПИСАНИЕ
Использование вариативности в универсальных методах-делегатах Func и Action (C#)	Узнайте, как использовать поддержку ковариации и контрвариантности в делегатах `Func` и `Action` для многократного использования кода.

Вариативность в универсальных интерфейсах (C#)

18.12.2021 • 2 minutes to read

В платформе .NET Framework 4 появилась поддержка вариативности для нескольких существующих универсальных интерфейсов. Поддержка вариативности позволяет выполнять неявное преобразование классов, реализующих эти интерфейсы.

Начиная с .NET Framework 4 вариативными являются следующие интерфейсы:

- IEnumerable<T> (T является ковариантным)
- IEnumerator<T> (T является ковариантным)
- IQueryable<T> (T является ковариантным)
- IGrouping<TKey,TElement> (`TKey` и `TElement` являются ковариантными)
- IComparer<T> (T является контравариантным)
- IEqualityComparer<T> (T является контравариантным)
- IComparable<T> (T является контравариантным)

Начиная с .NET Framework 4.5, вариативными являются следующие интерфейсы:

- IReadOnlyList<T> (T является ковариантным)
- IReadOnlyCollection<T> (T является ковариантным)

Ковариация позволяет методу иметь тип возвращаемого значения, степень наследования которого больше, чем указано в параметре универсального типа интерфейса. Чтобы продемонстрировать функцию ковариации, рассмотрим следующие универсальные интерфейсы: `IEnumerable<Object>` и `IEnumerable<String>`. Интерфейс `IEnumerable<String>` не наследует интерфейс `IEnumerable<Object>`. При этом тип `String` наследует тип `Object`, и в некоторых случаях может потребоваться назначить объекты этих интерфейсов друг другу. Это показано в следующем примере кода.

```
IEnumerable<String> strings = new List<String>();
IEnumerable<Object> objects = strings;
```

В более ранних версиях .NET Framework этот код приводил к ошибке компиляции в C#, если в Visual Basic был включен оператор `Option Strict`. Но теперь можно использовать `strings` вместо `objects`, как показано в предыдущем примере, поскольку интерфейс IEnumerable<T> является ковариантным.

Контравариантность позволяет методу иметь типы аргументов, степень наследования которых меньше, чем указано в параметре универсального типа интерфейса. Чтобы продемонстрировать функцию контравариантности, предположим, что создан класса `BaseComparer` для сравнения экземпляров класса `BaseClass`. Класс `BaseComparer` реализует интерфейс `IEqualityComparer<BaseClass>`. Поскольку теперь интерфейс IEqualityComparer<T> является контравариантным, для сравнения экземпляров классов, наследующих класс `BaseClass`, можно использовать класс `BaseComparer`. Это показано в следующем примере кода.

```
// Simple hierarchy of classes.
class BaseClass { }
class DerivedClass : BaseClass { }

// Comparer class.
class BaseComparer : IEqualityComparer<BaseClass>
{
    public int GetHashCode(BaseClass baseInstance)
    {
        return baseInstance.GetHashCode();
    }
    public bool Equals(BaseClass x, BaseClass y)
    {
        return x == y;
    }
}
class Program
{
    static void Test()
    {
        IEqualityComparer<BaseClass> baseComparer = new BaseComparer();

        // Implicit conversion of IEqualityComparer<BaseClass> to
        // IEqualityComparer<DerivedClass>.
        IEqualityComparer<DerivedClass> childComparer = baseComparer;
    }
}
```

Дополнительные примеры см.в разделе Использование вариативности в интерфейсах для универсальных коллекций (C#).

Вариативность в универсальных интерфейсах поддерживается только для ссылочных типов. Типы значений не поддерживают вариативность. Например, `IEnumerable<int>` нельзя неявно преобразовать в `IEnumerable<object>`, так как типом значения является integer.

```
IEnumerable<int> integers = new List<int>();
// The following statement generates a compiler error,
// because int is a value type.
// IEnumerable<Object> objects = integers;
```

Кроме того, важно помнить, что классы, которые реализуют вариативные интерфейсы, сами являются инвариантными. Например, несмотря на то, что List<T> реализует ковариантный интерфейс IEnumerable<T>, неявно преобразовать `List<String>` в `List<Object>` нельзя. Это показано в следующем примере кода.

```
// The following line generates a compiler error
// because classes are invariant.
// List<Object> list = new List<String>();

// You can use the interface object instead.
IEnumerable<Object> listObjects = new List<String>();
```

См. также

- Использование вариативности в интерфейсах для универсальных коллекций (C#)
- Создание вариантных универсальных интерфейсов (C#)
- Универсальные интерфейсы
- Вариативность в делегатах (C#)

Создание вариантных универсальных интерфейсов (C#)

18.12.2021 • 4 minutes to read

Параметры универсального типа можно объявить в интерфейсах как ковариантные или контравариантные. *Ковариация* позволяет методам интерфейса иметь тип возвращаемого значения, степень наследования которого больше, чем указано в параметрах универсального типа. *Контравариантность* позволяет методам интерфейса иметь типы аргументов, степень наследования которых меньше, чем указано в параметре универсального типа. Универсальный интерфейс, который имеет ковариантные или контравариантные параметры универсального типа, называется *вариантным*.

> **NOTE**
> В платформе .NET Framework 4 появилась поддержка вариативности для нескольких существующих универсальных интерфейсов. Список вариативных интерфейсов в .NET см. в статье Вариативность в универсальных интерфейсах (C#).

Объявление вариантных универсальных интерфейсов

Вариантные универсальные интерфейсы можно объявить с помощью ключевых слов `in` и `out` для параметров универсального типа.

> **IMPORTANT**
> Параметры `ref`, `in` и `out` в C# не могут быть вариантными. Типы значений также не поддерживают вариативность.

Для объявления ковариантного параметра универсального типа можно использовать ключевое слово `out`. Ковариантный тип должен удовлетворять следующим условиям:

- Тип используется только в качестве типа значения, возвращаемого методами интерфейса, и не используется в качестве типа аргументов метода. Это показано в следующем примере, в котором тип `R` объявлен ковариантным.

  ```
  interface ICovariant<out R>
  {
      R GetSomething();
      // The following statement generates a compiler error.
      // void SetSomething(R sampleArg);

  }
  ```

  Существует одно исключение из данного правила. Если в качестве параметра метода используется контравариантный универсальный делегат, этот тип можно использовать в качестве параметра универсального типа для этого делегата. Это продемонстрировано ниже на примере типа `R`. Дополнительные сведения см. в разделах Вариативность в делегатах (C#) и Использование вариативности в универсальных методах-делегатах Func и Action (C#).

```
interface ICovariant<out R>
{
    void DoSomething(Action<R> callback);
}
```

- Тип не используется в качестве универсального ограничения для методов интерфейса. Это демонстрируется в следующем примере кода.

```
interface ICovariant<out R>
{
    // The following statement generates a compiler error
    // because you can use only contravariant or invariant types
    // in generic constraints.
    // void DoSomething<T>() where T : R;
}
```

Для объявления контравариантного параметра универсального типа можно использовать ключевое слово `in`. Контравариантный тип можно использовать только в качестве типа аргументов метода, но не в качестве типа значения, возвращаемого методами интерфейса. Контравариантный тип можно также использовать для универсальных ограничений. В следующем примере кода показано объявление контравариантного интерфейса и использование универсального ограничения для одного из его методов.

```
interface IContravariant<in A>
{
    void SetSomething(A sampleArg);
    void DoSomething<T>() where T : A;
    // The following statement generates a compiler error.
    // A GetSomething();
}
```

Кроме того, можно реализовать поддержку ковариации и контравариации в одном интерфейсе, но для разных параметров типа, как показано в следующем примере кода.

```
interface IVariant<out R, in A>
{
    R GetSomething();
    void SetSomething(A sampleArg);
    R GetSetSomethings(A sampleArg);
}
```

Реализация вариантных универсальных интерфейсов

Для реализации вариантных универсальных интерфейсов в классах используется тот же синтаксис, что и для инвариантных интерфейсов. В следующем примере кода показана реализация ковариантного интерфейса в универсальном классе.

```
interface ICovariant<out R>
{
    R GetSomething();
}
class SampleImplementation<R> : ICovariant<R>
{
    public R GetSomething()
    {
        // Some code.
        return default(R);
    }
}
```

Классы, которые реализуют вариантные интерфейсы, являются инвариантными. Например, рассмотрим следующий код.

```
// The interface is covariant.
ICovariant<Button> ibutton = new SampleImplementation<Button>();
ICovariant<Object> iobj = ibutton;

// The class is invariant.
SampleImplementation<Button> button = new SampleImplementation<Button>();
// The following statement generates a compiler error
// because classes are invariant.
// SampleImplementation<Object> obj = button;
```

Расширение вариантных универсальных интерфейсов

При расширении вариантных универсальных интерфейсов необходимо использовать ключевые слова `in` и `out` для явного указания того, поддерживает ли вариативность производный интерфейс. Компилятор не подразумевает вариативность интерфейса, который расширяется. Например, рассмотрим следующие интерфейсы.

```
interface ICovariant<out T> { }
interface IInvariant<T> : ICovariant<T> { }
interface IExtCovariant<out T> : ICovariant<T> { }
```

В интерфейсе `IInvariant<T>` параметр универсального типа `T` является инвариантным, тогда как в `IExtCovariant<out T>` параметр типа является ковариантным, хотя оба интерфейса расширяют один и тот же интерфейс. То же правило применяется к контравариантным параметрам универсального типа.

Можно создать интерфейс, который расширяет и интерфейс, в котором параметр универсального типа `T` является ковариантным, и интерфейс, где он является контравариантным, если в расширяемом интерфейсе параметр универсального типа `T` является инвариантным. Это показано в следующем примере кода.

```
interface ICovariant<out T> { }
interface IContravariant<in T> { }
interface IInvariant<T> : ICovariant<T>, IContravariant<T> { }
```

Тем не менее, если параметр универсального типа `T` объявлен ковариантным в одном интерфейсе, его нельзя объявить контравариантным в расширенном интерфейсе и наоборот. Это показано в следующем примере кода.

```
interface ICovariant<out T> { }
// The following statement generates a compiler error.
// interface ICoContraVariant<in T> : ICovariant<T> { }
```

Недопущение неоднозначности

При реализации вариантных универсальных интерфейсов вариативность может приводить к неоднозначности. Такой неоднозначности следует избегать.

Например, если вы явно реализуете один вариантный универсальный интерфейс с разными параметрами универсального типа в одном классе, это может создавать неоднозначность. Компилятор не сообщает об ошибке в данном случае, но и не указывает, какая реализация интерфейса будет выбрана во время выполнения. Такая неоднозначность может привести к возникновению неявных ошибок в коде. Рассмотрим следующий пример кода:

```
// Simple class hierarchy.
class Animal { }
class Cat : Animal { }
class Dog : Animal { }

// This class introduces ambiguity
// because IEnumerable<out T> is covariant.
class Pets : IEnumerable<Cat>, IEnumerable<Dog>
{
    IEnumerator<Cat> IEnumerable<Cat>.GetEnumerator()
    {
        Console.WriteLine("Cat");
        // Some code.
        return null;
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        // Some code.
        return null;
    }

    IEnumerator<Dog> IEnumerable<Dog>.GetEnumerator()
    {
        Console.WriteLine("Dog");
        // Some code.
        return null;
    }
}
class Program
{
    public static void Test()
    {
        IEnumerable<Animal> pets = new Pets();
        pets.GetEnumerator();
    }
}
```

В этом примере не указано, каким образом метод `pets.GetEnumerator` делает выбор между `Cat` и `Dog`. Это может вызвать проблемы в вашем коде.

См. также

- Вариативность в универсальных интерфейсах (C#)
- Использование вариативности в универсальных методах-делегатах Func и Action (C#)

Использование вариативности в интерфейсах для универсальных коллекций (C#)

18.12.2021 • 2 minutes to read

Ковариантный интерфейс позволяет его методам возвращать более производные типы, чем указанные в интерфейсе. Контравариантный интерфейс позволяет его методам принимать параметры менее производных типов, чем указанные в интерфейсе.

В .NET Framework 4 несколько имеющихся интерфейсов стали ковариантными и контравариантными. В их числе IEnumerable<T> и IComparable<T>. Это позволяет повторно использовать методы, оперирующие универсальными коллекциями базовых типов, для коллекций производных типов.

Список вариативных интерфейсов в .NET см. в статье Вариативность в универсальных интерфейсах (C#).

Преобразование универсальных коллекций

Следующий пример иллюстрирует преимущества поддержки ковариантности в интерфейсе IEnumerable<T>. Метод `PrintFullName` принимает коллекцию типа `IEnumerable<Person>` в качестве параметра. При этом его можно повторно использовать для коллекции типа `IEnumerable<Employee>`, так как `Employee` наследует `Person`.

```
// Simple hierarchy of classes.
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Employee : Person { }

class Program
{
    // The method has a parameter of the IEnumerable<Person> type.
    public static void PrintFullName(IEnumerable<Person> persons)
    {
        foreach (Person person in persons)
        {
            Console.WriteLine("Name: {0} {1}",
            person.FirstName, person.LastName);
        }
    }

    public static void Test()
    {
        IEnumerable<Employee> employees = new List<Employee>();

        // You can pass IEnumerable<Employee>,
        // although the method expects IEnumerable<Person>.

        PrintFullName(employees);

    }
}
```

Сравнение универсальных коллекций

Следующий пример иллюстрирует преимущества поддержки контрвариантности в интерфейсе IEqualityComparer<T>. Класс `PersonComparer` реализует интерфейс `IEqualityComparer<Person>`. Тем не менее этот класс можно повторно использовать для сравнения последовательности объектов типа `Employee`, так как `Employee` наследует `Person`.

```
// Simple hierarchy of classes.
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Employee : Person { }

// The custom comparer for the Person type
// with standard implementations of Equals()
// and GetHashCode() methods.
class PersonComparer : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
        if (Object.ReferenceEquals(x, y)) return true;
        if (Object.ReferenceEquals(x, null) ||
            Object.ReferenceEquals(y, null))
            return false;
        return x.FirstName == y.FirstName && x.LastName == y.LastName;
    }
    public int GetHashCode(Person person)
    {
        if (Object.ReferenceEquals(person, null)) return 0;
        int hashFirstName = person.FirstName == null
            ? 0 : person.FirstName.GetHashCode();
        int hashLastName = person.LastName.GetHashCode();
        return hashFirstName ^ hashLastName;
    }
}

class Program
{

    public static void Test()
    {
        List<Employee> employees = new List<Employee> {
               new Employee() {FirstName = "Michael", LastName = "Alexander"},
               new Employee() {FirstName = "Jeff", LastName = "Price"}
            };

        // You can pass PersonComparer,
        // which implements IEqualityComparer<Person>,
        // although the method expects IEqualityComparer<Employee>.

        IEnumerable<Employee> noduplicates =
            employees.Distinct<Employee>(new PersonComparer());

        foreach (var employee in noduplicates)
            Console.WriteLine(employee.FirstName + " " + employee.LastName);
    }
}
```

См. также

- Вариативность в универсальных интерфейсах (C#)

Вариативность в делегатах (C#)

18.12.2021 • 5 minutes to read

В платформе .NET Framework 3.5 появилась поддержка вариативности при сопоставлении сигнатур методов с типами делегатов во всех делегатах в C#. Это означает, что делегатам можно назначать не только методы, которые обладают соответствующими сигнатурами, но и методы, которые возвращают более производные типы (ковариация), или принимают параметры, которые имеют менее производные типы (контравариативность), чем указано в типе делегата. Это касается не только универсальных методов-делегатов, но и методов-делегатов, не являющихся универсальными.

Например, рассмотрим следующий код, который содержит два класса и два делегата: универсальный и неуниверсальный.

```
public class First { }
public class Second : First { }
public delegate First SampleDelegate(Second a);
public delegate R SampleGenericDelegate<A, R>(A a);
```

При создании делегатов типов `SampleDelegate` или `SampleGenericDelegate<A, R>` им можно назначить любой из следующих методов.

```
// Matching signature.
public static First ASecondRFirst(Second second)
{ return new First(); }

// The return type is more derived.
public static Second ASecondRSecond(Second second)
{ return new Second(); }

// The argument type is less derived.
public static First AFirstRFirst(First first)
{ return new First(); }

// The return type is more derived
// and the argument type is less derived.
public static Second AFirstRSecond(First first)
{ return new Second(); }
```

В следующем примере кода показано неявное преобразование между сигнатурой метода и типом делегата.

```
// Assigning a method with a matching signature
// to a non-generic delegate. No conversion is necessary.
SampleDelegate dNonGeneric = ASecondRFirst;
// Assigning a method with a more derived return type
// and less derived argument type to a non-generic delegate.
// The implicit conversion is used.
SampleDelegate dNonGenericConversion = AFirstRSecond;

// Assigning a method with a matching signature to a generic delegate.
// No conversion is necessary.
SampleGenericDelegate<Second, First> dGeneric = ASecondRFirst;
// Assigning a method with a more derived return type
// and less derived argument type to a generic delegate.
// The implicit conversion is used.
SampleGenericDelegate<Second, First> dGenericConversion = AFirstRSecond;
```

Дополнительные примеры см. в разделах Использование вариативности в делегатах (C#) и Использование вариативности в универсальных методах-делегатах Func и Action (C#).

Вариативность в параметрах универсального типа

В платформе .NET Framework 4 и более поздних версиях можно включить неявное преобразование между делегатами, которое позволит универсальным методам-делегатам, имеющим разные типы, указанные параметрами универсального типа, быть назначенными друг другу, если типы наследуются друг от друга так, как того требует вариативность.

Чтобы включить неявное преобразование, необходимо явно объявить универсальные параметры в делегате как ковариантные или контравариантные с помощью ключевого слова `in` или `out`.

В следующем примере кода показано, как создать делегат, который имеет ковариантный параметр универсального типа.

```
// Type T is declared covariant by using the out keyword.
public delegate T SampleGenericDelegate <out T>();

public static void Test()
{
    SampleGenericDelegate <String> dString = () => " ";

    // You can assign delegates to each other,
    // because the type T is declared covariant.
    SampleGenericDelegate <Object> dObject = dString;
}
```

Если поддержка вариативности используется только для сопоставления сигнатур методов с типами делегатов, а ключевые слова `in` и `out` не используются, можно создать экземпляры делегатов с одинаковыми лямбда-выражениями или методами, но нельзя назначить один делегат другому.

В следующем примере кода `SampleGenericDelegate<String>` нельзя явно преобразовать в `SampleGenericDelegate<Object>`, хотя `String` наследует `Object`. Эту проблему можно устранить, пометив универсальный параметр `T` ключевым словом `out`.

```
public delegate T SampleGenericDelegate<T>();

public static void Test()
{
    SampleGenericDelegate<String> dString = () => " ";

    // You can assign the dObject delegate
    // to the same lambda expression as dString delegate
    // because of the variance support for
    // matching method signatures with delegate types.
    SampleGenericDelegate<Object> dObject = () => " ";

    // The following statement generates a compiler error
    // because the generic type T is not marked as covariant.
    // SampleGenericDelegate <Object> dObject = dString;

}
```

Универсальные методы-делегаты с вариативными параметрами типа в .NET

В платформе .NET Framework 4 появилась поддержка вариативности для параметров универсального типа в нескольких существующих методах-делегатах.

- Делегаты `Action` из пространства имен System, например Action<T> и Action<T1,T2>
- Делегаты `Func` из пространства имен System, например Func<TResult> и Func<T,TResult>
- Делегат Predicate<T>
- Делегат Comparison<T>
- Делегат Converter<TInput,TOutput>

Дополнительные примеры см. в разделе Использование вариативности в универсальных методах-делегатах Func и Action (C#).

Объявление вариативных параметров типа в универсальных методах-делегатах

Если универсальный метод-делегат содержит ковариантные или контравариантные параметры универсального типа, он называется *вариативным универсальным методом-делегатом*.

Для объявления ковариантного параметра универсального типа в универсальном методе-делегате можно использовать ключевое слово `out`. Ковариантный тип можно использовать только в качестве типа значения, возвращаемого методом, и нельзя использовать в качестве типа аргументов метода. В следующем примере кода показано, как объявить ковариантный универсальный метод-делегат.

```
public delegate R DCovariant<out R>();
```

Для объявления контравариантного параметра универсального типа в универсальном методе-делегате можно использовать ключевое слово `in`. Контравариантный тип можно использовать только в качестве типа аргументов метода, и нельзя использовать в качестве типа значения, возвращаемого методом. В следующем примере кода показано, как объявить контравариантный универсальный метод-делегат.

```
public delegate void DContravariant<in A>(A a);
```

IMPORTANT

Параметры `ref`, `in` и `out` в C# невозможно пометить как вариативные.

В одном делегате можно реализовать поддержку вариативности и ковариации, но для разных параметров типа. Эти действия показаны в следующем примере.

```
public delegate R DVariant<in A, out R>(A a);
```

Создание экземпляра и вызов вариативных универсальных методов-делегатов

Создание экземпляра и вызов вариативных делегатов возможен только при создании экземпляра и вызове инвариантных делегатов. В следующем примере создается экземпляр делегата с помощью лямбда-выражения.

```
DVariant<String, String> dvariant = (String str) => str + " ";
dvariant("test");
```

Объединение вариативных универсальных методов-делегатов

Не объединяйте вариативные делегаты. Метод Combine не поддерживает преобразование вариантных делегатов и ожидает делегаты того же самого типа. Это может вызвать исключение времени выполнения при объединении делегатов с помощью метода Combine или оператора `+`, как показано в следующем примере кода.

```
Action<object> actObj = x => Console.WriteLine("object: {0}", x);
Action<string> actStr = x => Console.WriteLine("string: {0}", x);
// All of the following statements throw exceptions at run time.
// Action<string> actCombine = actStr + actObj;
// actStr += actObj;
// Delegate.Combine(actStr, actObj);
```

Вариативность в параметрах универсального типа для значения и ссылочных типов

Вариативность для параметров универсального типа поддерживается только для ссылочных типов. Например, `DVariant<int>` нельзя неявно преобразовать в `DVariant<Object>` или `DVariant<long>`, поскольку integer является типом значения.

В следующем примере показано, что вариативность в параметрах универсального типа не поддерживается для типов значения.

```
// The type T is covariant.
public delegate T DVariant<out T>();

// The type T is invariant.
public delegate T DInvariant<T>();

public static void Test()
{
    int i = 0;
    DInvariant<int> dInt = () => i;
    DVariant<int> dVariantInt = () => i;

    // All of the following statements generate a compiler error
    // because type variance in generic parameters is not supported
    // for value types, even if generic type parameters are declared variant.
    // DInvariant<Object> dObject = dInt;
    // DInvariant<long> dLong = dInt;
    // DVariant<Object> dVariantObject = dVariantInt;
    // DVariant<long> dVariantLong = dVariantInt;
}
```

См. также

- Универсальные шаблоны
- Использование вариативности в универсальных методах-делегатах Func и Action (C#)
- Практическое руководство. Объединение делегатов (многоадресные делегаты)

Использование вариативности в делегатах (C#)

18.12.2021 • 2 minutes to read

При назначении метода делегату *ковариация* и *контравариантость* обеспечивают гибкость сопоставления типа делегата с сигнатурой метода. Ковариация позволяет методу иметь тип возвращаемого значения, степень наследования которого больше, чем указано в делегате. Контравариантность позволяет использовать метод с типами параметров, степень наследования которых меньше, чем у типа делегата.

Пример 1. Ковариантность

Описание

В этом примере демонстрируется использование делегатов с методами, типы возвращаемых значений которых являются производными от типа возвращаемого значения в сигнатуре делегата. Тип данных, возвращаемый `DogsHandler`, является типом `Dogs`, производным от определенного в делегате типа `Mammals`.

Код

```
class Mammals {}
class Dogs : Mammals {}

class Program
{
    // Define the delegate.
    public delegate Mammals HandlerMethod();

    public static Mammals MammalsHandler()
    {
        return null;
    }

    public static Dogs DogsHandler()
    {
        return null;
    }

    static void Test()
    {
        HandlerMethod handlerMammals = MammalsHandler;

        // Covariance enables this assignment.
        HandlerMethod handlerDogs = DogsHandler;
    }
}
```

Пример 2: Контрвариантность

Описание

В этом примере демонстрируется использование делегатов с методами, параметры типа которых являются базовыми типами типа параметра сигнатуры делегата. Контравариантность позволяет использовать один обработчик событий вместо нескольких. В следующем примере используется два делегата:

- Делегат KeyEventHandler, определяющий сигнатуру события Button.KeyDown. Его сигнатура:

```
public delegate void KeyEventHandler(object sender, KeyEventArgs e)
```

- Делегат MouseEventHandler, определяющий сигнатуру события Button.MouseClick. Его сигнатура:

```
public delegate void MouseEventHandler(object sender, MouseEventArgs e)
```

В примере определяется обработчик событий с параметром EventArgs, который используется для обработки событий `Button.KeyDown` и `Button.MouseClick`. Это можно сделать, поскольку EventArgs является базовым типом KeyEventArgs и MouseEventArgs.

Код

```
// Event handler that accepts a parameter of the EventArgs type.
private void MultiHandler(object sender, System.EventArgs e)
{
    label1.Text = System.DateTime.Now.ToString();
}

public Form1()
{
    InitializeComponent();

    // You can use a method that has an EventArgs parameter,
    // although the event expects the KeyEventArgs parameter.
    this.button1.KeyDown += this.MultiHandler;

    // You can use the same method
    // for an event that expects the MouseEventArgs parameter.
    this.button1.MouseClick += this.MultiHandler;

}
```

См. также

- Вариативность в делегатах (C#)
- Использование вариативности в универсальных методах-делегатах Func и Action (C#)

Использование вариативности в универсальных методах-делегатах Func и Action (C#)

18.12.2021 • 2 minutes to read

Эти примеры показывают, как обеспечить возможность многократного использования методов и сделать код более гибким, используя ковариацию и контравариацию в универсальных методах-делегатах `Func` и `Action`.

Дополнительные сведения о ковариации и контравариации см. в разделе Вариативность в делегатах (C#).

Использование методов-делегатов с параметрами ковариантного типа

Следующий пример иллюстрирует преимущества поддержки ковариации в универсальных методах-делегатах `Func`. Метод `FindByTitle` принимает параметр типа `String` и возвращает объект типа `Employee`. При этом данный метод можно назначить методу-делегату `Func<String, Person>`, поскольку `Employee` наследует `Person`.

```
// Simple hierarchy of classes.
public class Person { }
public class Employee : Person { }
class Program
{
    static Employee FindByTitle(String title)
    {
        // This is a stub for a method that returns
        // an employee that has the specified title.
        return new Employee();
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Func<String, Employee> findEmployee = FindByTitle;

        // The delegate expects a method to return Person,
        // but you can assign it a method that returns Employee.
        Func<String, Person> findPerson = FindByTitle;

        // You can also assign a delegate
        // that returns a more derived type
        // to a delegate that returns a less derived type.
        findPerson = findEmployee;

    }
}
```

Использование методов-делегатов с параметрами контравариантного типа

Следующий пример иллюстрирует преимущества поддержки контравариации в универсальных методах-делегатах `Action`. Метод `AddToContacts` принимает параметр типа `Person`. При этом данный метод можно назначить методу-делегату `Action<Employee>`, поскольку `Employee` наследует `Person`.

```
public class Person { }
public class Employee : Person { }
class Program
{
    static void AddToContacts(Person person)
    {
        // This method adds a Person object
        // to a contact list.
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Action<Person> addPersonToContacts = AddToContacts;

        // The Action delegate expects
        // a method that has an Employee parameter,
        // but you can assign it a method that has a Person parameter
        // because Employee derives from Person.
        Action<Employee> addEmployeeToContacts = AddToContacts;

        // You can also assign a delegate
        // that accepts a less derived parameter to a delegate
        // that accepts a more derived parameter.
        addEmployeeToContacts = addPersonToContacts;
    }
}
```

См. также

- Ковариация и контрвариантность (C#)
- Универсальные шаблоны

Деревья выражений (C#)

18.12.2021 • 4 minutes to read

Деревья выражений представляют код в виде древовидной структуры, где каждый узел является выражением, например, вызовом метода или двоичной операцией, такой как `x < y`.

Вы можете компилировать и выполнять код, представленный деревьями выражений. Это позволяет динамически изменять выполняемый код, выполнять запросы LINQ в различных базах данных и создавать динамические запросы. См. дополнительные сведения об использовании деревьев выражений в LINQ для создания динамических запросов (C#).

Кроме того, деревья выражений используются в среде выполнения динамического языка (DLR) для обеспечения взаимодействия между динамическими языками и платформой .NET, а также предоставления разработчикам компиляторов возможности выдавать деревья выражений вместо промежуточного языка MSIL. Дополнительные сведения о DLR см. в разделе Общие сведения о среде DLR.

Вы можете использовать компилятор C# или Visual Basic для создания дерева выражений на основе анонимного лямбда-выражения или создания деревьев выражений вручную с помощью пространства имен System.Linq.Expressions.

Создание деревьев выражений на основе лямбда-выражений

Когда лямбда-выражение назначается переменной с типом Expression<TDelegate>, компилятор выдает код для создания дерева выражений, представляющего лямбда-выражение.

Компилятор C# может создавать деревья выражений только на основе лямбда-выражений (или однострочных лямбда-функций). Они не могут анализировать лямбды операторов (или многострочные лямбды). Дополнительные сведения о лямбда-выражениях на C# см. в разделе Лямбда-выражения.

В следующем примере кода демонстрируется способ применения компилятора C# для создания дерева выражений, представляющего лямбда-выражение `num => num < 5`.

```
Expression<Func<int, bool>> lambda = num => num < 5;
```

Создание деревьев выражений с помощью API-интерфейса

Для создания деревьев выражений с помощью API-интерфейса используйте класс Expression. Этот класс содержит статические методы фабрики, позволяющие создать узлы дерева выражения конкретного типа, например, ParameterExpression, который представляет переменную или параметр, или MethodCallExpression, который представляет вызов метода. ParameterExpression, MethodCallExpression и другие зависящие от выражения типы также определяются в пространстве имен System.Linq.Expressions. Эти типы являются производными от абстрактного типа Expression.

В следующем примере кода демонстрируется способ создания дерева выражений, представляющего лямбда-выражение `num => num < 5`, с помощью API.

```
// Add the following using directive to your code file:
// using System.Linq.Expressions;

// Manually build the expression tree for
// the lambda expression num => num < 5.
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 =
    Expression.Lambda<Func<int, bool>>(
        numLessThanFive,
        new ParameterExpression[] { numParam });
```

На платформе .NET Framework 4 или более поздней версии API деревьев выражений также поддерживает присваивание и выражения потока управления, такие как циклы, условные блоки и блоки `try-catch`. С помощью API-интерфейса можно создавать деревья выражений, более сложные, чем деревья, создаваемые компилятором C# из лямбда-выражений. В следующем примере показан способ создания дерева выражений, которое вычисляет факториал числа.

```
// Creating a parameter expression.
ParameterExpression value = Expression.Parameter(typeof(int), "value");

// Creating an expression to hold a local variable.
ParameterExpression result = Expression.Parameter(typeof(int), "result");

// Creating a label to jump to from a loop.
LabelTarget label = Expression.Label(typeof(int));

// Creating a method body.
BlockExpression block = Expression.Block(
    // Adding a local variable.
    new[] { result },
    // Assigning a constant to a local variable: result = 1
    Expression.Assign(result, Expression.Constant(1)),
    // Adding a loop.
        Expression.Loop(
    // Adding a conditional block into the loop.
           Expression.IfThenElse(
    // Condition: value > 1
               Expression.GreaterThan(value, Expression.Constant(1)),
    // If true: result *= value --
               Expression.MultiplyAssign(result,
                   Expression.PostDecrementAssign(value)),
    // If false, exit the loop and go to the label.
               Expression.Break(label, result)
           ),
    // Label to jump to.
       label
    )
);

// Compile and execute an expression tree.
int factorial = Expression.Lambda<Func<int, int>>(block, value).Compile()(5);

Console.WriteLine(factorial);
// Prints 120.
```

Дополнительные сведения см. в записи блога Generating Dynamic Methods with Expression Trees in Visual Studio 2010 (Создание динамических методов с использованием деревьев выражений в Visual Studio 2010), которая также применима к более поздним версиям Visual Studio.

Синтаксический анализ деревьев выражений

В следующем примере кода показано, как дерево выражений, представляющее лямбда-выражение `num => num < 5`, может быть разложено на части.

```
// Add the following using directive to your code file:
// using System.Linq.Expressions;

// Create an expression tree.
Expression<Func<int, bool>> exprTree = num => num < 5;

// Decompose the expression tree.
ParameterExpression param = (ParameterExpression)exprTree.Parameters[0];
BinaryExpression operation = (BinaryExpression)exprTree.Body;
ParameterExpression left = (ParameterExpression)operation.Left;
ConstantExpression right = (ConstantExpression)operation.Right;

Console.WriteLine("Decomposed expression: {0} => {1} {2} {3}",
                  param.Name, left.Name, operation.NodeType, right.Value);

// This code produces the following output:

// Decomposed expression: num => num LessThan 5
```

Неизменность деревьев выражений

Деревья выражений должны быть неизменными. Это означает, что если требуется изменить дерево выражений, следует создать новое дерево выражений путем копирования существующего и заменить узлы в нем. Для прохода по существующему дереву выражений можно использовать другое дерево выражений (посетитель). См. дополнительные сведения об изменении деревьев выражений (C#).

Компиляция деревьев выражений

Тип Expression<TDelegate> предоставляет метод Compile, который компилирует код, представляемый деревом выражений, в исполняемый делегат.

В следующем примере кода показан способ компиляции дерева выражений и выполнения результирующего кода.

```
// Creating an expression tree.
Expression<Func<int, bool>> expr = num => num < 5;

// Compiling the expression tree into a delegate.
Func<int, bool> result = expr.Compile();

// Invoking the delegate and writing the result to the console.
Console.WriteLine(result(4));

// Prints True.

// You can also use simplified syntax
// to compile and run an expression tree.
// The following line can replace two previous statements.
Console.WriteLine(expr.Compile()(4));

// Also prints True.
```

См. дополнительные сведения о выполнении деревьев выражений (C#).

См. также

- System.Linq.Expressions
- Выполнение деревьев выражений (C#)
- Изменение деревьев выражений (C#)
- Лямбда-выражения
- Общие сведения о среде DLR
- Основные понятия программирования (C#)

Практическое руководство. Выполнение деревьев выражений (C#)

18.12.2021 • 2 minutes to read

В этом разделе показано, как выполнить дерево выражения. В результате выполнения дерева выражения может возвращаться значение или просто выполняться действие, такое как вызов метода.

Можно выполнять только деревья выражений, представляющие лямбда-выражения. Деревья выражений, представляющие лямбда-выражения, имеют тип LambdaExpression или Expression<TDelegate>. Для выполнения таких деревьев выражений вызовите метод Compile, чтобы создать исполняемый делегат, а затем вызовите делегат.

> **NOTE**
> Если тип делегата неизвестен, то есть лямбда-выражение имеет тип LambdaExpression, а не тип Expression<TDelegate>, необходимо вызвать метод DynamicInvoke для делегата, а не напрямую.

Если дерево выражения не представляет лямбда-выражение, можно создать новое лямбда-выражение, в качестве тела которого будет использоваться исходное дерево выражения. Для этого следует вызвать метод Lambda<TDelegate>(Expression, IEnumerable<ParameterExpression>). Затем можно выполнить лямбда-выражение, как описано ранее в этом разделе.

Пример

В приведенном ниже примере кода показано, как путем создания и выполнения лямбда-выражения выполнить дерево выражения, которое представляет возведение числа в степень. Выводится результат, представляющий число, возведенное в степень.

```
// The expression tree to execute.
BinaryExpression be = Expression.Power(Expression.Constant(2D), Expression.Constant(3D));

// Create a lambda expression.
Expression<Func<double>> le = Expression.Lambda<Func<double>>(be);

// Compile the lambda expression.
Func<double> compiledExpression = le.Compile();

// Execute the lambda expression.
double result = compiledExpression();

// Display the result.
Console.WriteLine(result);

// This code produces the following output:
// 8
```

Компиляция кода

- Включите пространство имен System.Linq.Expressions.

См. также

- Деревья выражений (C#)
- Изменение деревьев выражений (C#)

Практическое руководство. Изменение деревьев выражений (C#)

18.12.2021 • 2 minutes to read

В этом разделе показано, как изменить дерево выражения. Деревья выражений являются неизменяемыми, что означает невозможность их изменения напрямую. Чтобы изменить дерево выражения, необходимо создать копию существующего дерева выражения, а затем внести необходимые изменения. Для прохода по существующему дереву выражения и копирования каждого пройденного узла можно использовать класс ExpressionVisitor.

Изменение дерева выражения

1. Создайте новый проект **консольного приложения**.
2. Добавьте в файл директиву `using` для пространства имен `System.Linq.Expressions`.
3. Добавьте в проект класс `AndAlsoModifier`.

```
public class AndAlsoModifier : ExpressionVisitor
{
    public Expression Modify(Expression expression)
    {
        return Visit(expression);
    }

    protected override Expression VisitBinary(BinaryExpression b)
    {
        if (b.NodeType == ExpressionType.AndAlso)
        {
            Expression left = this.Visit(b.Left);
            Expression right = this.Visit(b.Right);

            // Make this binary expression an OrElse operation instead of an AndAlso operation.
            return Expression.MakeBinary(ExpressionType.OrElse, left, right, b.IsLiftedToNull,
b.Method);
        }

        return base.VisitBinary(b);
    }
}
```

   Этот класс наследует класс ExpressionVisitor и специально предназначен для изменения выражений, которые представляют условные операции `AND`. Он изменяет эти операции с условного `AND` на условное `OR`. Для этого класс переопределяет метод VisitBinary базового типа, потому что условные выражения `AND` представлены как двоичные выражения. Если выражение, переданное в метод `VisitBinary`, представляет условную операцию `AND`, код создает новое выражение, которое содержит условный оператор `OR` вместо условного оператора `AND`. Если выражение, передаваемое в `VisitBinary`, не представляет условную операцию `AND`, метод передает выполнение реализации базового класса. Методы базового класса создают узлы, которые похожи на переданные деревья выражений, однако поддеревья этих деревьев заменены на деревья выражений, которые были рекурсивно созданы при обходе.

4. Добавьте в файл директиву `using` для пространства имен `System.Linq.Expressions`.
5. Добавьте код в метод `Main` в файле Program.cs для создания дерева выражения и передачи его в

метод, который изменит его.

```
Expression<Func<string, bool>> expr = name => name.Length > 10 && name.StartsWith("G");
Console.WriteLine(expr);

AndAlsoModifier treeModifier = new AndAlsoModifier();
Expression modifiedExpr = treeModifier.Modify((Expression) expr);

Console.WriteLine(modifiedExpr);

/*  This code produces the following output:

    name => ((name.Length > 10) && name.StartsWith("G"))
    name => ((name.Length > 10) || name.StartsWith("G"))
*/
```

В приведенном ниже коде создается выражение, которое содержит условную операцию `AND`. Затем в нем создается экземпляр класса `AndAlsoModifier`, и в метод `Modify` этого класса передается выражение. Как исходные, так и измененные деревья выражений выводятся для отображения изменения.

6. Скомпилируйте и запустите приложение.

См. также

- Выполнение деревьев выражений (C#)
- Деревья выражений (C#)

Выполнение запросов на основе состояния среды выполнения (C#)

18.12.2021 • 7 minutes to read

Рассмотрим код, определяющий IQueryable или IQueryable<T> для источника данных:

```
var companyNames = new[] {
    "Consolidated Messenger", "Alpine Ski House", "Southridge Video",
    "City Power & Light", "Coho Winery", "Wide World Importers",
    "Graphic Design Institute", "Adventure Works", "Humongous Insurance",
    "Woodgrove Bank", "Margie's Travel", "Northwind Traders",
    "Blue Yonder Airlines", "Trey Research", "The Phone Company",
    "Wingtip Toys", "Lucerne Publishing", "Fourth Coffee"
};

// We're using an in-memory array as the data source, but the IQueryable could have come
// from anywhere -- an ORM backed by a database, a web request, or any other LINQ provider.
IQueryable<string> companyNamesSource = companyNames.AsQueryable();
var fixedQry = companyNames.OrderBy(x => x);
```

Каждый раз при выполнении этого кода выполняется один и тот же запрос. Зачастую это неэффективно, поскольку может потребоваться, чтобы код выполнял различные запросы в зависимости от условий во время выполнения. В этой статье описывается, как можно выполнить другой запрос на основе состояния среды выполнения.

IQueryable/IQueryable<T> и деревья выражений

В своей основе IQueryable имеет два компонента:

- Expression—представление компонентов текущего запроса с учетом языка и источника данных в виде дерева выражения.
- Provider—экземпляр поставщика LINQ, которому известно, как материализовать текущий запрос в значение или набор значений.

В контексте динамических запросов поставщик обычно остается неизменным; дерево выражения запроса будет отличаться от запроса к запросу.

Деревья выражений являются неизменяемыми; если вы хотите использовать другое дерево выражения — и, таким образом, другой запрос—, вам потребуется перевести существующее дерево выражения в новое, а значит в новое IQueryable.

В следующих разделах описываются конкретные методы запроса в зависимости от состояния среды выполнения.

- Использование состояния среды выполнения из дерева выражений
- Вызов дополнительных методов LINQ
- Изменение дерева выражения, переданного в методы LINQ
- Создайте выражение дерево выражения Expression<TDelegate> с помощью фабричных методов в Expression
- Добавление узлов вызова метода в дерево выражения IQueryable.
- Создание строк и использование динамической библиотеки LINQ

Использование состояния среды выполнения из дерева выражений

Предположив, что поставщик LINQ поддерживает его, самый простой способ динамического запроса будет заключаться в ссылке на состояние среды выполнения непосредственно в запросе с помощью закрытой переменной, такой как `length` в следующем примере кода:

```
var length = 1;
var qry = companyNamesSource
    .Select(x => x.Substring(0, length))
    .Distinct();

Console.WriteLine(string.Join(",", qry));
// prints: C, A, S, W, G, H, M, N, B, T, L, F

length = 2;
Console.WriteLine(string.Join(",", qry));
// prints: Co, Al, So, Ci, Wi, Gr, Ad, Hu, Wo, Ma, No, Bl, Tr, Th, Lu, Fo
```

Внутреннее дерево выражения, —а соответственно и запрос—, не были изменены; запрос возвращает разные значения только из-за изменения значения `length`.

Вызов дополнительных методов LINQ

Как правило, встроенные методы LINQ в Queryable выполняют два действия:

- Заключение текущего дерево выражения в оболочку в MethodCallExpression, который представляет вызов метода.
- Передача инкапсулированного дерева выражения в поставщик, чтобы вернуть значение через метод IQueryProvider.Execute поставщика, либо чтобы вернуть объект переведенного запроса с помощью метода IQueryProvider.CreateQuery.

Чтобы получить новый запрос, можно заменить исходный запрос на результат метода, возвращающего IQueryable<T>. Это можно сделать по условию на основе состояния среды выполнения, как показано в следующем примере:

```
// bool sortByLength = /* ... */;

var qry = companyNamesSource;
if (sortByLength)
{
    qry = qry.OrderBy(x => x.Length);
}
```

Изменение дерева выражения, переданного в методы LINQ

В методы LINQ можно передать разные выражения в зависимости от состояния среды выполнения:

```
// string? startsWith = /* ... */;
// string? endsWith = /* ... */;

Expression<Func<string, bool>> expr = (startsWith, endsWith) switch
{
    ("" or null, "" or null) => x => true,
    (_, "" or null) => x => x.StartsWith(startsWith),
    ("" or null, _) => x => x.EndsWith(endsWith),
    (_, _) => x => x.StartsWith(startsWith) || x.EndsWith(endsWith)
};

var qry = companyNamesSource.Where(expr);
```

Также может потребоваться составить различные подвыражения, используя сторонние библиотеки, такие как PredicateBuilder от LinqKit:

```
// This is functionally equivalent to the previous example.

// using LinqKit;
// string? startsWith = /* ... */;
// string? endsWith = /* ... */;

Expression<Func<string, bool>>? expr = PredicateBuilder.New<string>(false);
var original = expr;
if (!string.IsNullOrEmpty(startsWith))
{
    expr = expr.Or(x => x.StartsWith(startsWith));
}
if (!string.IsNullOrEmpty(endsWith))
{
    expr = expr.Or(x => x.EndsWith(endsWith));
}
if (expr == original)
{
    expr = x => true;
}

var qry = companyNamesSource.Where(expr);
```

Создание деревьев выражений и запросов с помощью фабричных методов

Во всех примерах нам известен тип элемента во время компиляции,— `string` —а соответственно и тип запроса— `IQueryable<string>`. Может потребоваться добавить компоненты в запрос любого типа элемента или добавить различные компоненты в зависимости от типа элемента. Можно создавать деревья выражений с нуля, используя фабричные методы в System.Linq.Expressions.Expression и таким образом адаптировать выражение в среде выполнения к определенному типу элемента.

Создание выражения<TDelegate>

При создании выражения для его передачи в один из методов LINQ фактически создается экземпляр выражения<TDelegate>, где `TDelegate` — это некоторый тип делегата, например `Func<string, bool>`, `Action` или настраиваемый тип делегата.

Выражение<TDelegate> наследуется от LambdaExpression, которое представляет собой полное лямбда-выражение следующего вида:

```
Expression<Func<string, bool>> expr = x => x.StartsWith("a");
```

LambdaExpression имеет два компонента:

- список параметров,— `(string x)` —представленных свойством Parameters;
- тело,— `x.StartsWith("a")` —представленное свойством Body.

Ниже указаны основные шаги по созданию выражения<TDelegate>.

- Определите объекты ParameterExpression для каждого из параметров (если таковые имеются) в лямбда-выражении с помощью фабричного метода Parameter.

  ```
  ParameterExpression x = Parameter(typeof(string), "x");
  ```

- Создайте текст LambdaExpression, используя заданные вами выражения ParameterExpression и фабричные методы в Expression. Например, выражение, представляющее `x.StartsWith("a")`, может быть построено следующим образом:

  ```
  Expression body = Call(
      x,
      typeof(string).GetMethod("StartsWith", new[] { typeof(string) })!,
      Constant("a")
  );
  ```

- Заключите параметры и текст в выражение<TDelegate> с типов времени компиляции, используя соответствующую перегрузку фабричного метода Lambda:

  ```
  Expression<Func<string, bool>> expr = Lambda<Func<string, bool>>(body, x);
  ```

В следующих разделах описывается сценарий, в котором может потребоваться создать выражение<TDelegate> для передачи в метод LINQ, а также представлен полный пример того, как это сделать с помощью фабричных методов.

Сценарий

Допустим, у вас есть несколько типов сущностей:

```
record Person(string LastName, string FirstName, DateTime DateOfBirth);
record Car(string Model, int Year);
```

Для любого из этих типов сущностей необходимо отфильтровать и возвратить только те сущности, которые имеют заданный текст в одном из полей `string`. Для `Person` необходимо найти свойства `FirstName` и `LastName`:

```
string term = /* ... */;
var personsQry = new List<Person>()
    .AsQueryable()
    .Where(x => x.FirstName.Contains(term) || x.LastName.Contains(term));
```

Но для `Car` требуется найти только свойство `Model`:

```
string term = /* ... */;
var carsQry = new List<Car>()
    .AsQueryable()
    .Where(x => x.Model.Contains(term));
```

Несмотря на то что можно написать одну настраиваемую функцию для `IQueryable<Person>` и другую для `IQueryable<Car>`, следующая функция добавляет эту фильтрацию в любой существующий запрос независимо от конкретного типа элемента.

Пример

```
// using static System.Linq.Expressions.Expression;

IQueryable<T> TextFilter<T>(IQueryable<T> source, string term)
{
    if (string.IsNullOrEmpty(term)) { return source; }

    // T is a compile-time placeholder for the element type of the query.
    Type elementType = typeof(T);

    // Get all the string properties on this specific type.
    PropertyInfo[] stringProperties =
        elementType.GetProperties()
            .Where(x => x.PropertyType == typeof(string))
            .ToArray();
    if (!stringProperties.Any()) { return source; }

    // Get the right overload of String.Contains
    MethodInfo containsMethod = typeof(string).GetMethod("Contains", new[] { typeof(string) })!;

    // Create a parameter for the expression tree:
    // the 'x' in 'x => x.PropertyName.Contains("term")'
    // The type of this parameter is the query's element type
    ParameterExpression prm = Parameter(elementType);

    // Map each property to an expression tree node
    IEnumerable<Expression> expressions = stringProperties
        .Select(prp =>
            // For each property, we have to construct an expression tree node like
x.PropertyName.Contains("term")
            Call(                  // .Contains(...)
                Property(          // .PropertyName
                    prm,           // x
                    prp
                ),
                containsMethod,
                Constant(term)     // "term"
            )
        );

    // Combine all the resultant expression nodes using ||
    Expression body = expressions
        .Aggregate(
            (prev, current) => Or(prev, current)
        );

    // Wrap the expression body in a compile-time-typed lambda expression
    Expression<Func<T, bool>> lambda = Lambda<Func<T, bool>>(body, prm);

    // Because the lambda is compile-time-typed (albeit with a generic parameter), we can use it with the
Where method
    return source.Where(lambda);
}
```

Поскольку функция `TextFilter` принимает и возвращает IQueryable<T> (а не только IQueryable), после текстового фильтра можно добавить дополнительные элементы запроса с типом времени компиляции.

```
var qry = TextFilter(
        new List<Person>().AsQueryable(),
        "abcd"
    )
    .Where(x => x.DateOfBirth < new DateTime(2001, 1, 1));

var qry1 = TextFilter(
        new List<Car>().AsQueryable(),
        "abcd"
    )
    .Where(x => x.Year == 2010);
```

Добавление узлов вызова метода в дерево выражения IQueryable

При наличии IQueryable вместо IQueryable<T> нельзя напрямую вызывать универсальные методы LINQ. В качестве альтернативы можно создать внутреннее дерево выражения, как показано выше, и использовать отражение для вызова соответствующего метода LINQ при передаче в дерево выражения.

Можно также дублировать функциональность метода LINQ, заключив все дерево в MethodCallExpression, который представляет вызов метода LINQ:

```
IQueryable TextFilter_Untyped(IQueryable source, string term)
{
    if (string.IsNullOrEmpty(term)) { return source; }
    Type elementType = source.ElementType;

    // The logic for building the ParameterExpression and the LambdaExpression's body is the same as in the
previous example,
    // but has been refactored into the constructBody function.
    (Expression? body, ParameterExpression? prm) = constructBody(elementType, term);
    if (body is null) {return source;}

    Expression filteredTree = Call(
        typeof(Queryable),
        "Where",
        new[] { elementType},
        source.Expression,
        Lambda(body, prm!)
    );

    return source.Provider.CreateQuery(filteredTree);
}
```

В этом случае у вас нет общего заполнителя `T` времени компиляции, поэтому вы будете использовать перегрузку Lambda, для которой не требуются сведения о типе времени компиляции, в результате чего вместо выражения<TDelegate> создается LambdaExpression.

Динамическая библиотека LINQ

Построение деревьев выражений с помощью фабричных методов достаточно сложное занятие; проще создавать строки. Динамическая библиотека LINQ предоставляет набор методов расширения для IQueryable, соответствующих стандартным методам LINQ в Queryable, и который принимает строки в специальном синтаксисе вместо деревьев выражений. Библиотека создает соответствующее дерево выражения из строки и может возвращать результирующий преобразованный IQueryable.

Например, предыдущий пример можно переписать следующим образом:

```
// using System.Linq.Dynamic.Core

IQueryable TextFilter_Strings(IQueryable source, string term) {
    if (string.IsNullOrEmpty(term)) { return source; }

    var elementType = source.ElementType;

    // Get all the string property names on this specific type.
    var stringProperties =
        elementType.GetProperties()
            .Where(x => x.PropertyType == typeof(string))
            .ToArray();
    if (!stringProperties.Any()) { return source; }

    // Build the string expression
    string filterExpr = string.Join(
        " || ",
        stringProperties.Select(prp => $"{prp.Name}.Contains(@0)")
    );

    return source.Where(filterExpr, term);
}
```

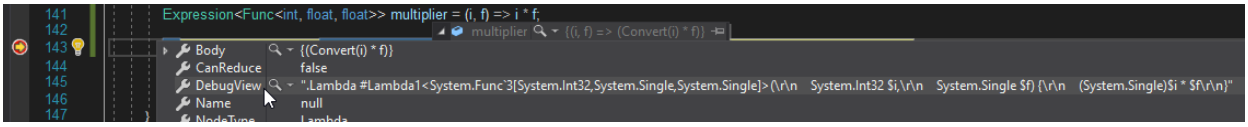
См. также

- Деревья выражений (C#)
- Выполнение деревьев выражений (C#)
- Динамическое определение фильтров предикатов во время выполнения

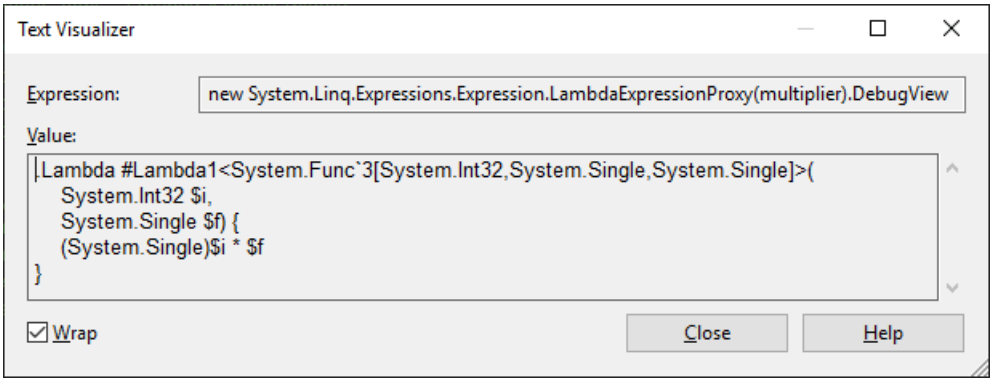
Отладка деревьев выражений в Visual Studio (C#)

18.12.2021 • 2 minutes to read

При отладке приложений можно анализировать структуру и содержимое деревьев выражений. Чтобы получить краткий обзор структуры дерева выражения, вы можете использовать свойство `DebugView`, которое представляет деревья выражений, используя специальный синтаксис. (Обратите внимание, что `DebugView` доступен только в режиме отладки.)

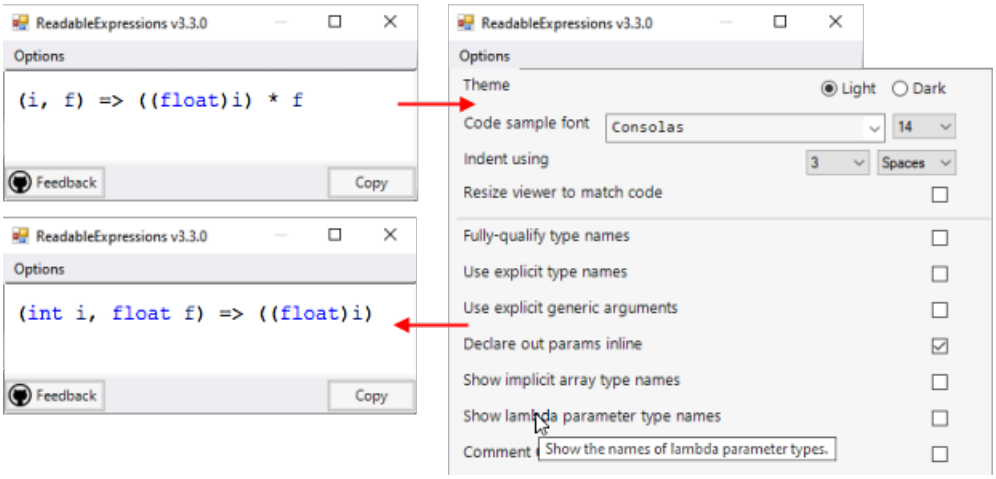


Так как `DebugView` представляет собой строку, можно использовать встроенный визуализатор текста для его просмотра по нескольким строкам, выбрав **визуализатор текста** из меню со значком лупы рядом с меткой `DebugView`.

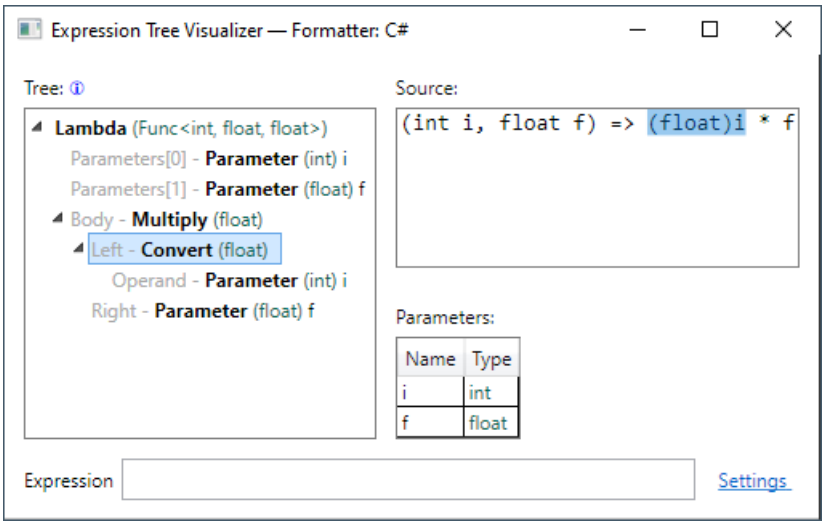


Или вы можете установить и использовать пользовательский визуализатор для деревьев выражений, например:

- Выражения для чтения (лицензия MIT, доступная в Visual Studio Marketplace) отображают дерево выражения в виде кода C# с темой и различными вариантами отрисовки:



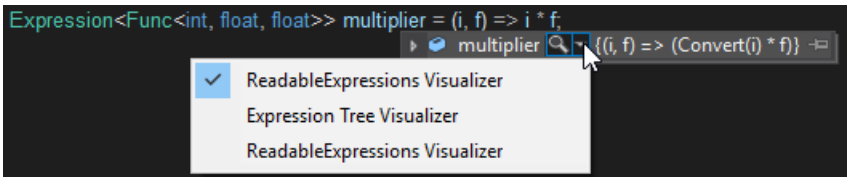
- Визуализатор дерева выражения (лицензия MIT) — древовидное представление дерева выражения и его узлов:



Открытие визуализатора дерева выражения

1. Щелкните значок лупы рядом с деревом выражения в окне **Подсказки по данным**, **Контрольные значения**, **Видимые** или **Локальные**.

   Отображается список доступных визуализаторов:



2. Щелкните визуализатор, который необходимо использовать.

См. также

- Деревья выражений (C#)
- Отладка в Visual Studio
- Создание настраиваемых визуализаторов
- Синтаксис `DebugView`

Синтаксис DebugView

18.12.2021 • 2 minutes to read

Свойство **DebugView** (доступно только при отладке) предоставляет строковое представление деревьев выражений. Большая часть синтаксиса достаточно проста для понимания; особые случаи описаны в разделах ниже.

Каждый пример сопровождается комментарием с **DebugView**.

ParameterExpression

В начале имен переменных ParameterExpression отображается символ `$`.

Если параметр не имеет имени, оно назначается автоматически, например `$var1` или `$var2`.

Примеры

```
ParameterExpression numParam =  Expression.Parameter(typeof(int), "num");
/*
    $num
*/

ParameterExpression numParam =  Expression.Parameter(typeof(int));
/*
    $var1
*/
```

ConstantExpression

Для объектов ConstantExpression, представляющих целочисленные значения, строки и `null`, отображается значение константы.

Для числовых типов, имеющих стандартные суффиксы, такие как литералы C#, суффикс добавляется к значению. В следующей таблице показаны суффиксы для различных числовых типов.

ТИП	КЛЮЧЕВОЕ СЛОВО	СУФФИКС
System.UInt32	uint	U
System.Int64	long	L
System.UInt64	ulong	UL
System.Double	double	D
System.Single	float	F
System.Decimal	decimal	M

Примеры

```
int num = 10;
ConstantExpression expr = Expression.Constant(num);
/*
    10
*/

double num = 10;
ConstantExpression expr = Expression.Constant(num);
/*
    10D
*/
```

BlockExpression

Если тип объекта BlockExpression отличается от типа последнего выражения в блоке, то тип отображается в угловых скобках (`<` и `>`). В противном случае тип объекта BlockExpression не отображается.

Примеры

```
BlockExpression block = Expression.Block(Expression.Constant("test"));
/*
    .Block() {
        "test"
    }
*/

BlockExpression block =  Expression.Block(typeof(Object), Expression.Constant("test"));
/*
    .Block<System.Object>() {
        "test"
    }
*/
```

LambdaExpression

Объекты LambdaExpression отображаются вместе со своими типами делегатов.

Если лямбда-выражение не имеет имени, оно назначается автоматически, например `#Lambda1` или `#Lambda2` .

Примеры

```
LambdaExpression lambda =  Expression.Lambda<Func<int>>(Expression.Constant(1));
/*
    .Lambda #Lambda1<System.Func'1[System.Int32]>() {
        1
    }
*/

LambdaExpression lambda =  Expression.Lambda<Func<int>>(Expression.Constant(1), "SampleLambda", null);
/*
    .Lambda #SampleLambda<System.Func'1[System.Int32]>() {
        1
    }
*/
```

LabelExpression

Если указать значение по умолчанию для объекта LabelExpression, оно будет отображаться перед объектом LabelTarget.

Маркер `.Label` указывает начало метки. Маркер `.LabelTarget` задает конечную цель перехода для целевого объекта.

Если метка не имеет имени, оно назначается автоматически, например `#Label1` или `#Label2`.

Примеры

```
LabelTarget target = Expression.Label(typeof(int), "SampleLabel");
BlockExpression block = Expression.Block(
    Expression.Goto(target, Expression.Constant(0)),
    Expression.Label(target, Expression.Constant(-1))
);
/*
    .Block() {
        .Goto SampleLabel { 0 };
        .Label
            -1
        .LabelTarget SampleLabel:
    }
*/

LabelTarget target = Expression.Label();
BlockExpression block = Expression.Block(
    Expression.Goto(target),
    Expression.Label(target)
);
/*
    .Block() {
        .Goto #Label1 { };
        .Label
        .LabelTarget #Label1:
    }
*/
```

Проверяемые операторы

Проверяемые операторы отображаются с символом `#` перед оператором. Например, проверяемый оператор сложения отображается как `#+`.

Примеры

```
Expression expr = Expression.AddChecked( Expression.Constant(1), Expression.Constant(2));
/*
    1 #+ 2
*/

Expression expr = Expression.ConvertChecked( Expression.Constant(10.0), typeof(int));
/*
    #(System.Int32)10D
*/
```

Итераторы (C#)

18.12.2021 • 6 minutes to read

Итератор можно использовать для прохода по коллекции, такой как список или массив.

Метод итератора или метод доступа `get` выполняет настраиваемую итерацию по коллекции. Метод итератора использует оператор yield return для поочередного возврата каждого элемента. При достижении инструкции `yield return` текущее расположение в коде запоминается. При следующем вызове функции итератора выполнение возобновляется с этого места.

Итератор используется из клиентского кода с помощью оператора foreach или с помощью запроса LINQ.

В приведенном ниже примере первая итерация цикла `foreach` приводит к вызову метода итератора `SomeNumbers`, пока не будет достигнут первый оператор `yield return`. Эта итерация возвращает значение 3. Текущее расположение в методе итератора сохраняется. В следующей итерации цикла выполнение метода итератора возобновляется с того же места и снова приостанавливается при достижении оператора `yield return`. Эта итерация возвращает значение 5. Текущее расположение в методе итератора снова сохраняется. Цикл завершается при достижении конца метода итератора.

```
static void Main()
{
    foreach (int number in SomeNumbers())
    {
        Console.Write(number.ToString() + " ");
    }
    // Output: 3 5 8
    Console.ReadKey();
}

public static System.Collections.IEnumerable SomeNumbers()
{
    yield return 3;
    yield return 5;
    yield return 8;
}
```

Типом возвращаемого метода итератора или метода доступа `get` может быть IEnumerable, IEnumerable<T>, IEnumerator или IEnumerator<T>.

Для завершения итерации можно использовать оператор `yield break`.

NOTE

Все примеры в этом разделе, кроме примера простого итератора, включают директивы using для пространств имен `System.Collections` и `System.Collections.Generic`.

Простой итератор

В следующем примере имеется один оператор `yield return`, который находится внутри цикла for. В методе `Main` каждая итерация оператора `foreach` создает вызов функции итератора, которая выполняет следующий оператор `yield return`.

```
static void Main()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.Write(number.ToString() + " ");
    }
    // Output: 6 8 10 12 14 16 18
    Console.ReadKey();
}

public static System.Collections.Generic.IEnumerable<int>
    EvenSequence(int firstNumber, int lastNumber)
{
    // Yield even numbers in the range.
    for (int number = firstNumber; number <= lastNumber; number++)
    {
        if (number % 2 == 0)
        {
            yield return number;
        }
    }
}
```

Создание класса коллекции

В следующем примере класс `DaysOfTheWeek` реализует интерфейс IEnumerable, которому требуется метод GetEnumerator. Компилятор неявно вызывает метод `GetEnumerator`, который возвращает IEnumerator.

Метод `GetEnumerator` возвращает каждую строку поочередно с помощью оператора `yield return`.

```
static void Main()
{
    DaysOfTheWeek days = new DaysOfTheWeek();

    foreach (string day in days)
    {
        Console.Write(day + " ");
    }
    // Output: Sun Mon Tue Wed Thu Fri Sat
    Console.ReadKey();
}

public class DaysOfTheWeek : IEnumerable
{
    private string[] days = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };

    public IEnumerator GetEnumerator()
    {
        for (int index = 0; index < days.Length; index++)
        {
            // Yield each day of the week.
            yield return days[index];
        }
    }
}
```

В приведенном ниже примере создается класс `Zoo`, содержащий коллекцию животных.

Оператор `foreach`, который обращается к экземпляру класса (`theZoo`), неявно вызывает метод `GetEnumerator`. Операторы `foreach`, которые обращаются к свойствам `Birds` и `Mammals`, используют метод итератора с именем `AnimalsForType`.

```
static void Main()
{
    Zoo theZoo = new Zoo();

    theZoo.AddMammal("Whale");
    theZoo.AddMammal("Rhinoceros");
    theZoo.AddBird("Penguin");
    theZoo.AddBird("Warbler");

    foreach (string name in theZoo)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Whale Rhinoceros Penguin Warbler

    foreach (string name in theZoo.Birds)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Penguin Warbler

    foreach (string name in theZoo.Mammals)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Whale Rhinoceros

    Console.ReadKey();
}

public class Zoo : IEnumerable
{
    // Private members.
    private List<Animal> animals = new List<Animal>();

    // Public methods.
    public void AddMammal(string name)
    {
        animals.Add(new Animal { Name = name, Type = Animal.TypeEnum.Mammal });
    }

    public void AddBird(string name)
    {
        animals.Add(new Animal { Name = name, Type = Animal.TypeEnum.Bird });
    }

    public IEnumerator GetEnumerator()
    {
        foreach (Animal theAnimal in animals)
        {
            yield return theAnimal.Name;
        }
    }

    // Public members.
    public IEnumerable Mammals
    {
        get { return AnimalsForType(Animal.TypeEnum.Mammal); }
    }

    public IEnumerable Birds
    {
        get { return AnimalsForType(Animal.TypeEnum.Bird); }
    }

    // Private methods.
```

```
    // Private methods.
    private IEnumerable AnimalsForType(Animal.TypeEnum type)
    {
        foreach (Animal theAnimal in animals)
        {
            if (theAnimal.Type == type)
            {
                yield return theAnimal.Name;
            }
        }
    }

    // Private class.
    private class Animal
    {
        public enum TypeEnum { Bird, Mammal }

        public string Name { get; set; }
        public TypeEnum Type { get; set; }
    }
}
```

Использование итераторов с универсальным списком

В следующем примере универсальный класс Stack<T> также реализует универсальный интерфейс IEnumerable<T>. Метод Push присваивает значения массиву типа `T`. Метод GetEnumerator возвращает массив значений с помощью оператора `yield return`.

Помимо универсального метода GetEnumerator должен быть реализован и неуниверсальный метод GetEnumerator. Это связано с тем, что IEnumerable<T> наследуется от IEnumerable. Неуниверсальная реализация подчиняется универсальной реализации.

В этом примере используются именованные итераторы для поддержки различных способов итерации по одной и той же коллекции данных. Эти именованные итераторы являются свойствами `TopToBottom` и `BottomToTop` и методом `TopN`.

Свойство `BottomToTop` использует итератор в методе доступа `get`.

```
static void Main()
{
    Stack<int> theStack = new Stack<int>();

    //  Add items to the stack.
    for (int number = 0; number <= 9; number++)
    {
        theStack.Push(number);
    }

    // Retrieve items from the stack.
    // foreach is allowed because theStack implements IEnumerable<int>.
    foreach (int number in theStack)
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3 2 1 0

    // foreach is allowed, because theStack.TopToBottom returns IEnumerable(Of Integer).
    foreach (int number in theStack.TopToBottom)
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3 2 1 0
```

```
    foreach (int number in theStack.BottomToTop)
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 0 1 2 3 4 5 6 7 8 9

    foreach (int number in theStack.TopN(7))
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3

    Console.ReadKey();
}

public class Stack<T> : IEnumerable<T>
{
    private T[] values = new T[100];
    private int top = 0;

    public void Push(T t)
    {
        values[top] = t;
        top++;
    }
    public T Pop()
    {
        top--;
        return values[top];
    }

    // This method implements the GetEnumerator method. It allows
    // an instance of the class to be used in a foreach statement.
    public IEnumerator<T> GetEnumerator()
    {
        for (int index = top - 1; index >= 0; index--)
        {
            yield return values[index];
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }

    public IEnumerable<T> TopToBottom
    {
        get { return this; }
    }

    public IEnumerable<T> BottomToTop
    {
        get
        {
            for (int index = 0; index <= top - 1; index++)
            {
                yield return values[index];
            }
        }
    }

    public IEnumerable<T> TopN(int itemsFromTop)
    {
        // Return less than itemsFromTop if necessary.
        int startIndex = itemsFromTop >= top ? 0 : top - itemsFromTop;
```

```
        for (int index = top - 1; index >= startIndex; index--)
        {
            yield return values[index];
        }
    }

}
```

Сведения о синтаксисе

Итератор может являться методом или методом доступа `get`. Итератор не может использоваться в событии, конструкторе экземпляра, статическом конструкторе или статическом методе завершения.

Должно существовать неявное преобразование типа выражения в операторе `yield return` в аргумент типа для `IEnumerable<T>`, возвращаемого итератором.

В C# метод итератора не может иметь параметры `in`, `ref` или `out`.

В C# `yield` не является зарезервированным словом и имеет специальное значение, только если используется перед ключевым словом `return` или `break`.

Техническая реализация

Хотя итератор создается как метод, компилятор переводит его во вложенный класс, который фактически является конечным автоматом. Этот класс отслеживает положение итератора, пока в клиентском коде выполняется цикл `foreach`.

Чтобы просмотреть операции компилятора, воспользуйтесь средством Ildasm.exe, чтобы просмотреть код промежуточного языка Майкрософт, создаваемый для метода итератора.

При создании итератора для класса или структуры реализация всего интерфейса IEnumerator не требуется. Когда компилятор обнаруживает итератор, он автоматически создает методы `Current`, `MoveNext` и `Dispose` интерфейса IEnumerator или IEnumerator<T>.

В каждой последовательной итерации цикла `foreach` (или непосредственном вызове метода `IEnumerator.MoveNext`) код тела следующего итератора возобновляет выполнение после предыдущего оператора `yield return`. Затем он выполняется до следующего оператора `yield return` до тех пор, пока не будет достигнут конец тела итератора или пока не будет обнаружен оператор `yield break`.

Итераторы не поддерживают метод IEnumerator.Reset. Для повторной итерации сначала необходимо получить новый итератор. Вызов Reset в итераторе, который возвращается методом итератора, вызывает исключение NotSupportedException.

Дополнительные сведения см. в спецификации языка C#.

Использование итераторов

Итераторы позволяют поддерживать простоту цикла `foreach`, когда необходимо использовать сложный код для заполнения последовательности списков. Это может оказаться полезным в следующих случаях:

- Изменение последовательности списков после первой итерации цикла `foreach`.
- Если необходимо избежать полной загрузки большого списка перед первой итерацией цикла `foreach`. Пример: при постраничной загрузке пакета строк таблицы. Другой пример — метод EnumerateFiles, реализующий итераторы в .NET.
- Инкапсулирование построения списка в итераторе. В методе итератора можно построить список, а

затем выдавать каждый результат в цикле.

См. также

- System.Collections.Generic
- IEnumerable<T>
- foreach, in
- yield
- Использование оператора foreach с массивами
- Универсальные шаблоны

LINQ — (C#)

18.12.2021 • 3 minutes to read

Аббревиатура LINQ обозначает целый набор технологий, создающих и использующих возможности интеграции запросов непосредственно в язык C#. Традиционно запросы к данным выражаются в виде простых строк без проверки типов при компиляции или поддержки IntelliSense. Кроме того, разработчику приходится изучать различные языки запросов для каждого типа источников данных: баз данных SQL, XML-документов, различных веб-служб и т. д. Технологии LINQ превращают запросы в удобную языковую конструкцию, которая применяется аналогично классам, методам и событиям. Вы создаете запросы к строго типизированным коллекциям объектов с помощью ключевых слов языка и знакомых операторов. Семейство технологий LINQ обеспечивает согласованное функционирование запросов для объектов (LINQ to Objects), реляционных баз данных (LINQ to SQL) и XML (LINQ to XML).

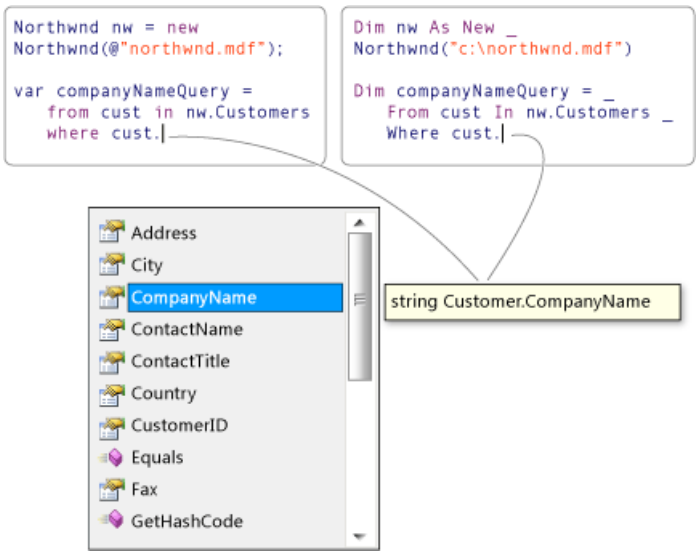
Для разработчика, который создает запросы, наиболее очевидной частью LINQ является интегрированное выражение запроса. Выражения запроса используют декларативный *синтаксис запроса*. С помощью синтаксиса запроса можно выполнять фильтрацию, упорядочение и группирование данных из источника данных, обходясь минимальным объемом программного кода. Одни и те же базовые выражения запроса позволяют запрашивать и преобразовывать данные из баз данных SQL, наборов данных ADO.NET, XML-документов, XML-потоков и коллекций .NET.

Вы можете писать запросы LINQ в C# для обращения к базам данных SQL Server, XML-документам, наборам данных ADO.NET и любым коллекциям объектов, поддерживающим интерфейс IEnumerable или универсальный интерфейс IEnumerable<T>. Кроме того, сторонние разработчики обеспечивают поддержку LINQ для множества веб-служб и других реализаций баз данных.

В следующем примере показан полный пример использования запроса. Полная операция сначала создает источник данных, затем определяет выражение запроса и выполняет этот запрос в инструкции `foreach`.

```
class LINQQueryExpressions
{
    static void Main()
    {

        // Specify the data source.
        int[] scores = new int[] { 97, 92, 81, 60 };

        // Define the query expression.
        IEnumerable<int> scoreQuery =
            from score in scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in scoreQuery)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 97 92 81
```

На приведенном ниже рисунке показан частично выполненный запрос LINQ к базе данных SQL Server в C# и Visual Basic с полной проверкой типов и поддержкой IntelliSense:



Описание выражения запроса

- Выражение запроса можно использовать для получения и преобразования данных из любого источника данных, поддерживающего LINQ. Например, можно одним запросом получить данные из базы данных SQL и создать на их основе выходной XML-поток.
- Выражение запроса очень легко понять, поскольку в нем используется много знакомых конструкций языка C#.
- Все переменные в выражениях запросов строго типизированы, хотя во многих случаях вам не нужно указывать тип явным образом, поскольку компилятор определит его автоматически. Дополнительные сведения см. в статье Отношения между типами в операциях запросов LINQ (C#).
- Запрос не будет выполняться, пока вы не начнете обращаться к переменной запроса, например, с помощью инструкции `foreach`. Дополнительные сведения см. в статье Введение в запросы LINQ (C#).
- Во время компиляции выражения запроса преобразуются в вызовы метода стандартного оператора запроса. Для преобразования используются правила, заданные в спецификации C#. Любой запрос, который может быть выражен с помощью синтаксиса запросов, также может быть выражен с помощью синтаксиса методов. Однако в большинстве случаев синтаксис запроса будет понятнее и лаконичнее. Дополнительные сведения см. в статьях C# language specification (Спецификация языка C#) и Общие сведения о стандартных операторах запроса (C#).
- Мы рекомендуем при написании запросов LINQ использовать синтаксис запросов везде, где это возможно, а синтаксис метода — только если это совершенно необходимо. Между этими формами синтаксиса нет никакой разницы в семантике или производительности. Выражения запросов обычно более удобочитаемыми, чем аналогичные выражения с использованием синтаксиса метода.
- Некоторых операции запросов, например Count или Max, не имеют эквивалентных предложений выражения для запросов и должны выражаться как вызовы методов. Синтаксис запросов и синтаксис методов можно сочетать друг с другом различными способами. Дополнительные сведения см. в статье Синтаксис запросов и синтаксис методов в LINQ (C#).
- Выражения запросов могут компилироваться в деревья выражений или в делегаты, в зависимости от типа, к которому применяется конкретный запрос. Запросы IEnumerable<T> компилируются в делегаты. Запросы IQueryable и IQueryable<T> компилируются в деревья выражений. Дополнительные сведения см. в статье Деревья выражений.

Следующие шаги

Чтобы получить дополнительные сведения о LINQ, сначала ознакомьтесь с некоторыми основным понятиями в статье Основы выражения запроса, а затем переходите к документации по интересующей вас технологии LINQ.

- XML-документы: LINQ to XML
- Платформа ADO.NET Entity Framework: LINQ to Entities
- Коллекции, файлы, строки и другие сущности .NET: LINQ to Objects (C#)

Чтобы глубже разобраться в базовой концепции LINQ изучите статью о LINQ в C#.

Чтобы быстрее приступить к работе с LINQ в C#, переходите к руководству Работа с LINQ.

Введение в запросы LINQ (C#)

18.12.2021 • 5 minutes to read

Запрос представляет собой выражение, извлекающее данные из источника данных. Запросы обычно выражаются на специализированном языке запросов. Со временем для различных типов источников данных, например SQL для реляционных баз данных и XQuery для XML, были разработаны разные языки. Поэтому разработчикам приходится учить новый язык запросов для каждого типа источника данных или формата данных, для которых они должны обеспечить поддержку. LINQ упрощает ситуацию, реализуя согласованную модель работы с данными для различных типов источников и форматов данных. В запросе LINQ вы всегда работаете с объектами. Вы используете одинаковые базовые шаблоны кода для запроса и преобразования данных в XML-документы, базы данных SQL, наборы данных ADO.NET, коллекции .NET и любые другие форматы, для которых доступен поставщик LINQ.

Три составляющие операции запроса

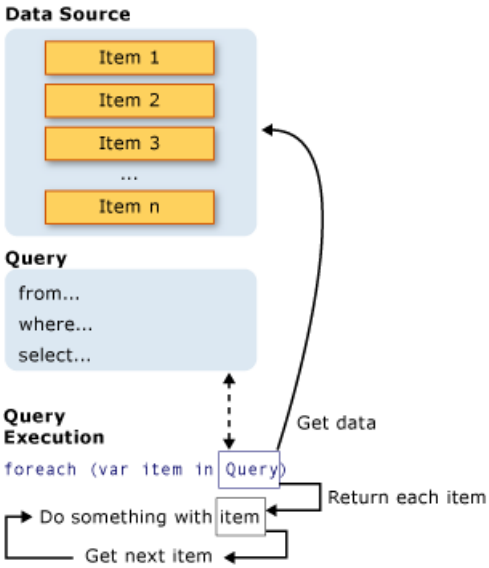
Все операции запросов LINQ состоят из трех отдельных действий:

1. получение источника данных;
2. создание запроса;
3. выполнение запроса.

В следующем примере показано, как эти три части операции запроса выражаются в исходном коде. В этом примере для удобства используется массив целых чисел в качестве источника данных. Тем не менее те же принципы применимы к другим источникам данных. Остальные процедуры этого раздела содержат ссылки на этот пример.

```
class IntroToLINQ
{
    static void Main()
    {
        // The Three Parts of a LINQ Query:
        // 1. Data source.
        int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };

        // 2. Query creation.
        // numQuery is an IEnumerable<int>
        var numQuery =
            from num in numbers
            where (num % 2) == 0
            select num;

        // 3. Query execution.
        foreach (int num in numQuery)
        {
            Console.Write("{0,1} ", num);
        }
    }
}
```

На следующем рисунке показана полная операция запроса. Выполнение запроса в LINQ отличается от самого запроса. Иными словами, данные не извлекаются путем создания переменной запроса.



Источник данных

Так как источник данных в предыдущем примере является массивом, он неявно поддерживает универсальный интерфейс IEnumerable<T>. Это значит, что его можно запросить с помощью LINQ. Запрос выполняется в операторе `foreach`, и для `foreach` требуется IEnumerable или IEnumerable<T>. Типы, которые поддерживают IEnumerable<T> или производный интерфейс, например универсальный интерфейс IQueryable<T>, называются *запрашиваемыми типами*.

Запрашиваемый тип не требует внесения изменений или специальной обработки, чтобы служить источником данных LINQ. Если источник данных не находится в памяти в качестве запрашиваемого типа, поставщик LINQ должен представить его как таковой. Например, LINQ to XML загружает XML-документ в запрашиваемый тип XElement:

```
// Create a data source from an XML document.
// using System.Xml.Linq;
XElement contacts = XElement.Load(@"c:\myContactList.xml");
```

В LINQ to SQL сначала необходимо создать объектно-реляционное сопоставление во время разработки вручную либо с помощью инструментов LINQ to SQL в Visual Studio. Вы создаете запросы к объектам, а LINQ to SQL во время выполнения обрабатывает взаимодействие с базой данных. В следующем примере `Customers` представляет собой определенную таблицу в базе данных, а тип результата запроса (IQueryable<T>) является производным от IEnumerable<T>.

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");

// Query for customers in London.
IQueryable<Customer> custQuery =
    from cust in db.Customers
    where cust.City == "London"
    select cust;
```

Дополнительные сведения о способах создания определенных типов источников данных см. в документации для различных поставщиков LINQ. Но общее правило очень простое: источником данных LINQ является любой объект, который поддерживает универсальный интерфейс IEnumerable<T> или интерфейс, наследуемый от него.

NOTE

Такие типы как ArrayList, которые поддерживают неуниверсальный интерфейс IEnumerable, также можно использовать в качестве источника данных LINQ. Дополнительные сведения см. в разделе Практическое руководство. Выполнение запроса к ArrayList с помощью LINQ (C#).

Запрос

Запрос указывает, какую информацию нужно извлечь из источника или источников данных. Дополнительно в запросе можно указать, как следует сортировать, группировать и формировать возвращаемую информацию. Запрос хранится в переменной запроса и инициализируется выражением запроса. Чтобы упростить написание запросов, в C# был представлен новый синтаксис запроса.

В предыдущем примере запрос возвращает все четные числа из массива целых чисел. Выражение запроса содержит три предложения: `from`, `where` и `select`. (Если вы знакомы с SQL, то должны были заметить, что порядок предложений противоположен порядку в SQL.) Предложение `from` указывает источник данных, предложение `where` применяет фильтр, а предложение `select` задает тип возвращаемых элементов. Эти и другие предложения запросов подробно описываются в разделе LINQ. А сейчас важно то, что в LINQ сама переменная запроса не выполняет никаких действий и не возвращает никаких данных. Она просто хранит сведения, необходимые для предоставления результатов при выполнении запроса на более позднем этапе. Дополнительные сведения о принципах конструирования запросов см. в разделе Общие сведения о стандартных операторах запросов (C#).

NOTE

Запросы могут также выражаться с помощью синтаксиса методов. Дополнительные сведения см. в разделе Синтаксис запросов и синтаксис методов в LINQ.

Выполнение запроса

Отложенное выполнение

Как уже говорилось ранее, сама переменная запроса хранит команды запроса. Фактическое выполнение запроса откладывается до тех пор, пока переменная запроса не будет обработана в операторе `foreach`. Эта концепция называется *отложенным выполнением* и демонстрируется в следующем примере:

```
//  Query execution.
foreach (int num in numQuery)
{
    Console.Write("{0,1} ", num);
}
```

Оператор `foreach` также является частью кода, в которой извлекаются результаты запроса. Например, в предыдущем запросе переменная итерации `num` содержит каждое значение (по одному за раз) в возвращенной последовательности.

Поскольку сама переменная запроса никогда не содержит результаты запроса, вы можете выполнять ее так часто, как это необходимо. Например, может иметься база данных, постоянно обновляемая отдельным приложением. В приложении можно создать один запрос, получающий самые последние данные, и регулярно выполнять его с некоторым интервалом, каждый раз получая разные результаты.

Принудительное немедленное выполнение

Запросы, выполняющие статистические функции для диапазона исходных элементов, сначала должны

выполнить итерацию по этим элементам. Примерами таких запросов являются `Count`, `Max`, `Average` и `First`. Они выполняются без явного оператора `foreach`, поскольку сам запрос должен использовать `foreach` для возвращения результата. Обратите внимание, что такие типы запросов возвращают одно значение, а не коллекцию `IEnumerable`. Следующий запрос возвращает количество четных чисел в исходном массиве:

```
var evenNumQuery =
    from num in numbers
    where (num % 2) == 0
    select num;

int evenNumCount = evenNumQuery.Count();
```

Чтобы принудительно вызвать немедленное выполнение любого запроса и кэшировать его результаты, вы можете вызвать методы ToList или ToArray.

```
List<int> numQuery2 =
    (from num in numbers
     where (num % 2) == 0
     select num).ToList();

// or like this:
// numQuery3 is still an int[]

var numQuery3 =
    (from num in numbers
     where (num % 2) == 0
     select num).ToArray();
```

Принудительно вызвать выполнение также можно, поместив цикл `foreach` сразу после выражения запроса. При этом путем вызова `ToList` или `ToArray` можно также кэшировать все данные в одном объекте коллекции.

См. также

- Приступая к работе с LINQ в C#
- Пошаговое руководство: Написание запросов на C#
- LINQ
- foreach, in
- Ключевые слова запроса (LINQ)

LINQ и универсальные типы (C#)

18.12.2021 • 2 minutes to read

Запросы LINQ основаны на универсальных типах, которые впервые появились в .NET Framework 2.0. Для того чтобы приступить к написанию запросов, не требуется глубокое знание универсальных типов. Тем не менее необходимо понимание двух основных принципов.

1. При создании экземпляра универсального класса коллекции, такого как List<T>, замените "T" типом объектов, которые будут храниться в списке. Например, список строк выражается как `List<string>`, а список объектов `Customer` — как `List<Customer>`. Универсальный список является строго типизированным и предоставляет ряд преимуществ по сравнению с коллекциями, которые хранят свои элементы как Object. При попытке добавить `Customer` в `List<string>` будет выдана ошибка во время компиляции. Универсальные коллекции просты в использовании, поскольку нет необходимости выполнять приведение типов во время выполнения.

2. IEnumerable<T> — это интерфейс, поддерживающий перечисление универсальных классов коллекций с помощью оператора `foreach`. Универсальные классы коллекций поддерживают IEnumerable<T> так же, как неуниверсальные классы коллекций (например ArrayList) поддерживают IEnumerable.

Дополнительные сведения об универсальных классах см. в разделе Универсальные шаблоны.

Переменные IEnumerable<T> в запросах LINQ

Переменные запроса LINQ типизированы как IEnumerable<T> или производный тип, например IQueryable<T>. Если появляется переменная запроса, которая типизируется как `IEnumerable<Customer>`, это просто означает, что запрос при выполнении выведет последовательность из нуля или более объектов `Customer`.

```
IEnumerable<Customer> customerQuery =
    from cust in customers
    where cust.City == "London"
    select cust;

foreach (Customer customer in customerQuery)
{
    Console.WriteLine(customer.LastName + ", " + customer.FirstName);
}
```

Дополнительные сведения см. в разделе Связи типов в операциях запроса LINQ.

Передача обработки объявлений универсальных типов компилятору

При желании можно отказаться от обычного синтаксиса универсальных типов, используя ключевое слово var. Ключевое слово `var` дает компилятору указание вывести тип переменной запроса путем поиска в источнике данных, указанном в предложении `from`. В следующем примере создается тот же скомпилированный код, что и в предыдущем примере:

```
var customerQuery2 =
    from cust in customers
    where cust.City == "London"
    select cust;

foreach(var customer in customerQuery2)
{
    Console.WriteLine(customer.LastName + ", " + customer.FirstName);
}
```

Ключевое слово `var` удобно, если тип переменной является очевидным, или если не требуется явно указывать вложенные универсальные типы, например создаваемые групповыми запросами. Как правило, рекомендуется помнить о том, что использование `var` делает код более сложным для чтения. Дополнительные сведения см. в статье Неявно типизированные локальные переменные (руководство по программированию на C#).

См. также

- Универсальные шаблоны

Основные операции запросов LINQ (C#)

18.12.2021 • 4 minutes to read

В этом разделе содержится краткое описание выражений запроса LINQ и некоторые типичные операции, выполняемые в запросе. Более подробные сведения приводятся в следующих разделах:

Выражения запросов LINQ

Общие сведения о стандартных операторах запроса (C#)

Пошаговое руководство: Написание запросов на C#

> **NOTE**
> Если вы уже знакомы с языком запросов, таким как SQL или XQuery, можно пропустить большую часть этого раздела. Чтобы изучить порядок предложений в выражениях запроса LINQ, прочитайте о "предложении `from`" в следующем разделе.

Получение источника данных

В первую очередь в запросе LINQ нужно указать источник данных. В C#, как и в большинстве языков программирования, переменная должна быть объявлена до ее использования. В запросе LINQ первым идет предложение `from` для указания источника данных (`customers`) и *переменная диапазона* (`cust`).

```
//queryAllCustomers is an IEnumerable<Customer>
var queryAllCustomers = from cust in customers
                        select cust;
```

Переменная диапазона схожа с переменной итерации в цикле `foreach` за исключением того, что в выражении запроса не происходит фактической итерации. При выполнении запроса переменная диапазона будет использоваться как ссылка на каждый последующий элемент в `customers` . Так как компилятор может определить тип `cust` , нет необходимости указывать его в явном виде. Дополнительные переменные диапазона могут быть введены предложением `let` . Дополнительные сведения см. в разделе Предложение let.

> **NOTE**
> Для неуниверсальных источников данных, таких как ArrayList, переменная диапазона должна быть явно типизирована. Дополнительные сведения см. в разделах Практическое руководство. Выполнение запроса к ArrayList с помощью LINQ (C#) и Предложение from.

Фильтрация

По-видимому, одной из наиболее распространенных операций с запросом является применение фильтра в форме логического выражения. Фильтр приводит к возвращению запросом только тех элементов, для которых выражение является истинным. Результат вырабатывается с использованием предложения `where` . Фильтр фактически указывает элементы для исключения из исходной последовательности. В приведенном ниже примере возвращаются только те элементы `customers` , которые находятся в Лондоне.

```
var queryLondonCustomers = from cust in customers
                           where cust.City == "London"
                           select cust;
```

Для применения нужного числа выражений фильтра в предложении `where` можно использовать знакомые логические операторы C# `AND` и `OR`. Например, для получения только заказчиков из Лондона `AND` с именем Devon следует написать следующий код:

```
where cust.City == "London" && cust.Name == "Devon"
```

Для получения заказчиков из Лондона или Парижа следует написать следующий код:

```
where cust.City == "London" || cust.City == "Paris"
```

Дополнительные сведения см. в разделе Предложение where.

Упорядочение

Часто целесообразно отсортировать возвращенные данные. Предложение `orderby` сортирует элементы возвращаемой последовательности в зависимости от компаратора по умолчанию для сортируемого типа. Например, приведенный ниже запрос можно расширить для сортировки результатов на основе свойства `Name`. Так как `Name` является строкой, сравнение по умолчанию выполняется в алфавитном порядке от A до Я.

```
var queryLondonCustomers3 =
    from cust in customers
    where cust.City == "London"
    orderby cust.Name ascending
    select cust;
```

Для упорядочения результатов в обратном порядке от Я до A используется предложение `orderby…descending`.

Дополнительные сведения см. в разделе Предложение orderby.

Группирование

Предложение `group` позволяет группировать результаты на основе указанного ключа. Например, можно указать, что результаты должны быть сгруппированы по `City` так, чтобы все заказчики из Лондона или Парижа оказались в отдельных группах. В этом случае ключом является `cust.City`.

```
// queryCustomersByCity is an IEnumerable<IGrouping<string, Customer>>
  var queryCustomersByCity =
      from cust in customers
      group cust by cust.City;

  // customerGroup is an IGrouping<string, Customer>
  foreach (var customerGroup in queryCustomersByCity)
  {
      Console.WriteLine(customerGroup.Key);
      foreach (Customer customer in customerGroup)
      {
          Console.WriteLine("    {0}", customer.Name);
      }
  }
```

Когда запрос завершается предложением `group`, результаты представляются в виде списка из списков. Каждый элемент в списке является объектом, имеющим член `Key` и список элементов, сгруппированных по этому ключу. При итерации запроса, создающего последовательность групп, необходимо использовать вложенный цикл `foreach`. Внешний цикл выполняет итерацию каждой группы, а внутренний цикл — итерацию членов каждой группы.

Если необходимо ссылаться на результаты операции группировки, можно использовать ключевое слово `into` для создания идентификатора, который можно будет запрашивать. Приведенный ниже запрос возвращает только те группы, которые содержат более двух заказчиков.

```
// custQuery is an IEnumerable<IGrouping<string, Customer>>
var custQuery =
    from cust in customers
    group cust by cust.City into custGroup
    where custGroup.Count() > 2
    orderby custGroup.Key
    select custGroup;
```

Дополнительные сведения см. в разделе Предложение group.

Соединение

Операции соединения создают связи между последовательностями, неявно смоделированными в источниках данных. Например, можно выполнить соединение для поиска всех клиентов и дистрибьюторов, которые находятся в одном месте. В LINQ предложение `join` всегда работает с коллекциями объектов, а не непосредственно с таблицами базы данных.

```
var innerJoinQuery =
    from cust in customers
    join dist in distributors on cust.City equals dist.City
    select new { CustomerName = cust.Name, DistributorName = dist.Name };
```

В LINQ нет необходимости использовать `join` так часто, как в SQL, так как внешние ключи в LINQ представлены в объектной модели свойствами, содержащими коллекцию элементов. Например, объект `Customer` содержит коллекцию объектов `Order`. Вместо выполнения соединения доступ к заказам можно получить с помощью точечной нотации.

```
from order in Customer.Orders...
```

Дополнительные сведения см. в разделе Предложение join.

Выбор (проецирование)

Предложение `select` создает результаты запроса и задает форму или тип каждого возвращаемого элемента. Например, можно указать, будут ли результаты состоять из полных объектов `Customer`, только из одного члена, подмножества членов или некоторых совершенно других типов, на основе вычислений или создания новых объектов. Когда предложение `select` создает что-либо отличное от копии исходного элемента, операция называется *проекцией*. Использование проекций для преобразования данных является эффективной возможностью выражений запросов LINQ. Дополнительные сведения см. в разделах Преобразования данных с помощью LINQ (C#) и Предложение select.

См. также

- Выражения запросов LINQ
- Пошаговое руководство: Написание запросов на C#
- Ключевые слова запроса (LINQ)
- Анонимные типы

Преобразования данных с помощью LINQ (C#)

18.12.2021 • 5 minutes to read

LINQ используется не только для получения данных. Это эффективный инструмент для их преобразования. С помощью запросов LINQ можно использовать исходную последовательность в качестве входных данных и изменять ее разными способами для создания новой выходной последовательности. Можно изменить саму последовательность, не изменяя элементы, с помощью сортировки и группировки. Но самой интересной функцией запросов LINQ можно назвать возможность создания новых типов. Это выполняется в предложении select. Например, можно выполнить следующие задачи.

- Объединение нескольких входных последовательностей в одну выходную последовательность, имеющую новый тип.
- Создание выходных последовательностей, элементы которых состоят из одного или нескольких свойств каждого элемента в исходной последовательности.
- Создание выходных последовательностей, элементы которых состоят из результатов операций, выполняемых с источником данных.
- Создание выходных последовательностей в другом формате. Например, можно преобразовать данные из строк SQL или текстовых файлов в XML.

Это лишь несколько примеров. Очевидно, что эти преобразования могут сочетаться различными способами в одном запросе. Более того, выходную последовательность одного запроса можно использовать как входную последовательность для нового запроса.

Объединение нескольких входных последовательностей в одну выходную

Запрос LINQ можно использовать для создания выходной последовательности, содержащей элементы из нескольких входных последовательностей. Следующий пример демонстрирует объединение двух структур данных в памяти, однако те же принципы могут применяться для объединения данных из источников XML, SQL или DataSet. Рассмотрим следующие два типа классов:

```
class Student
{
    public string First { get; set; }
    public string Last {get; set;}
    public int ID { get; set; }
    public string Street { get; set; }
    public string City { get; set; }
    public List<int> Scores;
}

class Teacher
{
    public string First { get; set; }
    public string Last { get; set; }
    public int ID { get; set; }
    public string City { get; set; }
}
```

В следующем примере показан запрос:

```
class DataTransformations
{
    static void Main()
    {
        // Create the first data source.
        List<Student> students = new List<Student>()
        {
            new Student { First="Svetlana",
                Last="Omelchenko",
                ID=111,
                Street="123 Main Street",
                City="Seattle",
                Scores= new List<int> { 97, 92, 81, 60 } },
            new Student { First="Claire",
                Last="O'Donnell",
                ID=112,
                Street="124 Main Street",
                City="Redmond",
                Scores= new List<int> { 75, 84, 91, 39 } },
            new Student { First="Sven",
                Last="Mortensen",
                ID=113,
                Street="125 Main Street",
                City="Lake City",
                Scores= new List<int> { 88, 94, 65, 91 } },
        };

        // Create the second data source.
        List<Teacher> teachers = new List<Teacher>()
        {
            new Teacher { First="Ann", Last="Beebe", ID=945, City="Seattle" },
            new Teacher { First="Alex", Last="Robinson", ID=956, City="Redmond" },
            new Teacher { First="Michiyo", Last="Sato", ID=972, City="Tacoma" }
        };

        // Create the query.
        var peopleInSeattle = (from student in students
                    where student.City == "Seattle"
                    select student.Last)
                    .Concat(from teacher in teachers
                            where teacher.City == "Seattle"
                            select teacher.Last);

        Console.WriteLine("The following students and teachers live in Seattle:");
        // Execute the query.
        foreach (var person in peopleInSeattle)
        {
            Console.WriteLine(person);
        }

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    The following students and teachers live in Seattle:
    Omelchenko
    Beebe
 */
```

Дополнительные сведения см. в разделе Предложение join и Предложение select.

Выбор подмножества каждого исходного элемента

Существует два основных способа выбора подмножества каждого элемента в исходной

последовательности:

1. Выбор только одного члена исходного элемента с помощью операции dot. В следующем примере предполагается, что объект `Customer` содержит несколько открытых свойств, включая строку с именем `City`. При выполнении этот запрос создаст выходную последовательность строк.

```
var query = from cust in Customers
            select cust.City;
```

2. Для создания элементов, содержащих более одного свойства исходного элемента, можно использовать инициализатор объектов с именованным объектом или анонимным типом. В следующем примере показано использование анонимного типа для инкапсуляции двух свойств из каждого элемента `Customer`:

```
var query = from cust in Customer
            select new {Name = cust.Name, City = cust.City};
```

Дополнительные сведения см. в разделе Инициализаторы объектов и коллекций и Анонимные типы.

Преобразование объектов в памяти в XML

Запросы LINQ позволяют легко преобразовывать данные между структурами данных в памяти, базами данных SQL, наборами данных ADO.NET и XML-потоками или документами. В следующем примере объекты в структуре данных в памяти преобразуются в XML-элементы.

```
class XMLTransform
{
    static void Main()
    {
        // Create the data source by using a collection initializer.
        // The Student class was defined previously in this topic.
        List<Student> students = new List<Student>()
        {
            new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores = new List<int>{97, 92, 81,
60}},
            new Student {First="Claire", Last="O'Donnell", ID=112, Scores = new List<int>{75, 84, 91, 39}},
            new Student {First="Sven", Last="Mortensen", ID=113, Scores = new List<int>{88, 94, 65, 91}},
        };

        // Create the query.
        var studentsToXML = new XElement("Root",
            from student in students
            let scores = string.Join(",", student.Scores)
            select new XElement("student",
                       new XElement("First", student.First),
                       new XElement("Last", student.Last),
                       new XElement("Scores", scores)
                    ) // end "student"
                ); // end "Root"

        // Execute the query.
        Console.WriteLine(studentsToXML);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

Этот код создает следующий выходные данные XML:

```
<Root>
  <student>
    <First>Svetlana</First>
    <Last>Omelchenko</Last>
    <Scores>97,92,81,60</Scores>
  </student>
  <student>
    <First>Claire</First>
    <Last>O'Donnell</Last>
    <Scores>75,84,91,39</Scores>
  </student>
  <student>
    <First>Sven</First>
    <Last>Mortensen</Last>
    <Scores>88,94,65,91</Scores>
  </student>
</Root>
```

Дополнительные сведения см. в разделе Создание деревьев XML C# (LINQ to XML).

Выполнение операций с исходными элементами

Выходная последовательность не может содержать любые элементы или свойства элементов из исходной последовательности. Результатом может быть последовательность значений, вычисляемых с использованием исходных элементов в качестве входных аргументов.

Следующий запрос принимает последовательность чисел, представляющих радиусы окружностей, вычисляет площадь каждой окружности и возвращает выходную последовательность, содержащую строки, отформатированные с учетом вычисленной площади.

Каждая строка для выходной последовательности будет отформатирована с помощью интерполяции строк. Перед открывающей кавычкой в интерполированной строке будет находиться `$`, а операции будут выполняться внутри фигурных скобок, помещенных в интерполированную строку. После выполнения операций результаты будут объединены.

> **NOTE**
>
> Вызов методов в выражениях запросов не поддерживается, если запрос будет преобразован в некоторую другую область. Например, невозможно вызвать обычный метод C# в LINQ to SQL, так как в SQL Server для него отсутствует контекст. Тем не менее можно сопоставить хранимые процедуры методов и вызвать их. Дополнительные сведения см. в разделе Хранимые процедуры.

```
class FormatQuery
{
    static void Main()
    {
        // Data source.
        double[] radii = { 1, 2, 3 };

        // LINQ query using method syntax.
        IEnumerable<string> output =
            radii.Select(r => $"Area for a circle with a radius of '{r}' = {r * r * Math.PI:F2}");

        /*
        // LINQ query using query syntax.
        IEnumerable<string> output =
            from rad in radii
            select $"Area for a circle with a radius of '{rad}' = {rad * rad * Math.PI:F2}";
        */

        foreach (string s in output)
        {
            Console.WriteLine(s);
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Area for a circle with a radius of '1' = 3.14
    Area for a circle with a radius of '2' = 12.57
    Area for a circle with a radius of '3' = 28.27
*/
```

См. также

- LINQ (C#)
- LINQ to SQL
- LINQ to DataSet
- LINQ to XML (C#)
- Выражения запросов LINQ
- предложение select

Отношения между типами в операциях запросов LINQ (C#)

18.12.2021 • 2 minutes to read

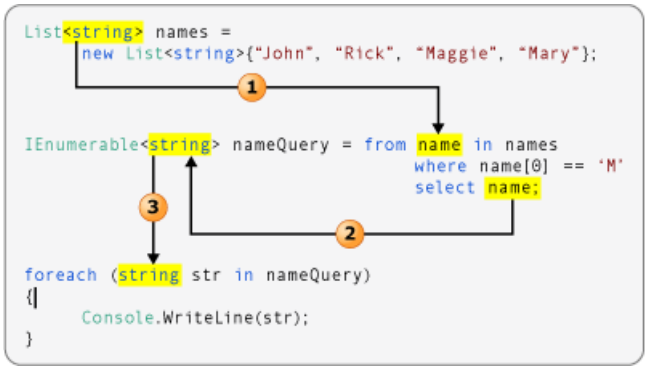
Для эффективного написания запросов следует понимать, как типы переменных связаны друг с другом в полной операции запроса. В таком случае вам будет проще работать с примерами LINQ и примерами кода в документации. Более того, можно будет представить, что происходит в фоновом режиме при неявном типизировании переменных с помощью `var`.

Операции запросов LINQ строго типизированы в источнике данных, в самом запросе и при выполнении запроса. Тип переменных в запросе должен быть совместим с типом элементов в источнике данных и с типом переменной итерации в операторе `foreach`. Строгая типизация гарантирует перехват ошибок во время компиляции, когда их можно будет исправить прежде, чем с ними столкнутся пользователи.

Для демонстрации связи типов большая часть примеров использует явную типизацию для всех переменных. Последний пример показывает применение тех же принципов даже при использовании неявной типизации с помощью var.

Запросы, не выполняющие преобразование исходных данных

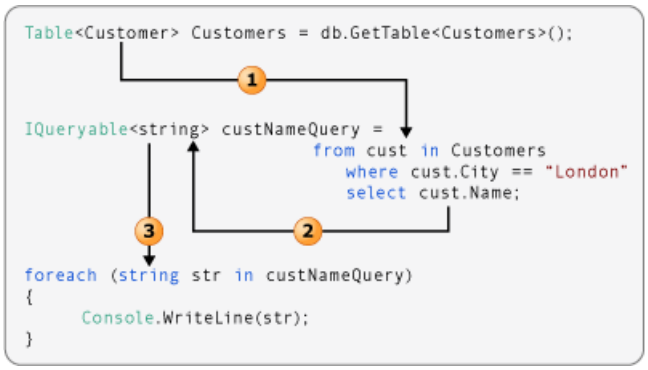
На следующем рисунке показана операция запроса LINQ to Objects, не выполняющая преобразование данных. Источник содержит последовательность строк, результат запроса также является последовательностью строк.



1. Аргумент типа источника данных определяет тип переменной диапазона.
2. Тип выбранного объекта определяет тип переменной запроса. Здесь `name` является строкой. Следовательно, переменная запроса представляет собой `IEnumerable<string>`.
3. Итерация переменной запроса выполняется в операторе `foreach`. Поскольку переменная запроса является последовательностью строк, переменная итерации также является строкой.

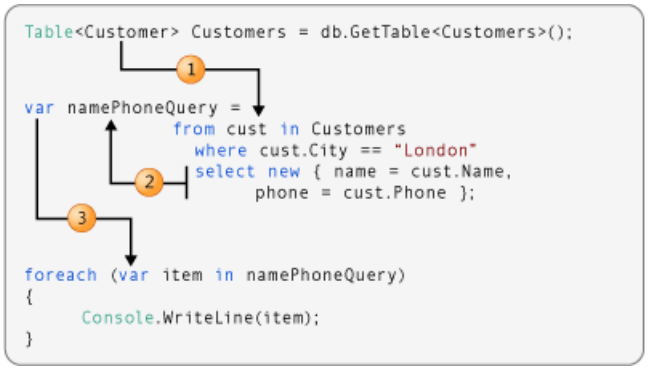
Запросы, выполняющие преобразование исходных данных

На следующем рисунке показана операция запроса LINQ to SQL, выполняющая простое преобразование данных. В качестве входных данных запрос получает последовательность объектов `Customer` и выбирает в результате только свойство `Name`. Поскольку `Name` является строкой, запрос формирует последовательность строк в качестве выходных данных.



1. Аргумент типа источника данных определяет тип переменной диапазона.
2. Оператор `select` возвращает свойство `Name` вместо целого объекта `Customer`. Поскольку `Name` является строкой, аргумент типа `custNameQuery` является `string`, а не `Customer`.
3. Поскольку `custNameQuery` представляет собой последовательность строк, переменная итерации цикла `foreach` также должна быть `string`.

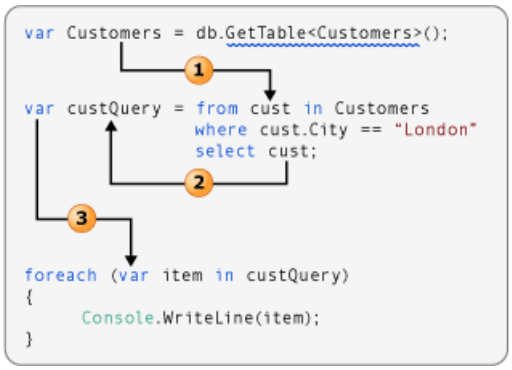
На следующем рисунке показано немного более сложное преобразование. Оператор `select` возвращает анонимный тип, захватывающий только два члена исходного объекта `Customer`.



1. Аргумент типа источника данных всегда является типом переменной диапазона в запросе.
2. Так как оператор `select` создает анонимный тип, переменная запроса должна неявно типизирована с помощью `var`.
3. Поскольку тип переменной запроса неявный, переменная итерации в цикле `foreach` также должна быть неявной.

Разрешение компилятору определять сведения о типе

Несмотря на то, что необходимо обладать знаниями об отношениях типов в операции запроса, существует возможность передачи выполнения всех действий компилятору. Ключевое слово var можно использовать для любой локальной переменной в операции запроса. Следующий рисунок похож на пример 2, рассмотренный выше. Однако компилятор предоставляет строгий тип для каждой переменной в операции запроса.



Дополнительные сведения о `var` см. в разделе Неявно типизированные локальные переменные.

Синтаксис запросов и синтаксис методов в LINQ (C#)

18.12.2021 • 3 minutes to read

Большинство запросов в вводной документации к LINQ написано с использованием декларативного синтаксиса запросов LINQ. Однако синтаксис запроса должен быть преобразован в вызовы методов для среды CLR .NET, когда код компилируется. Эти вызовы метода вызывают стандартные операторы запросов, которые имеют такие имена, как `Where`, `Select`, `GroupBy`, `Join`, `Max` и `Average`. Вместо синтаксиса запросов для их вызова можно использовать синтаксис методов.

Синтаксис запросов и синтаксис методов семантически идентичны, но многие пользователи найдут синтаксис запросов более простым и более удобным для чтения. Некоторые запросы должны быть выражены как вызовы методов. Например, необходимо использовать вызов метода для выражения запроса, который возвращает число элементов, соответствующих указанным критериям. Вызов метода также необходимо использовать для запроса, который получает элемент с максимальным значением в исходной последовательности. В справочной документации по стандартным операторам запросов в пространствах имен System.Linq обычно применяется синтаксис методов. В связи с этим даже на начальном этапе работы с запросами LINQ полезно иметь представление о том, как использовать синтаксис методов в самих запросах и выражениях запросов.

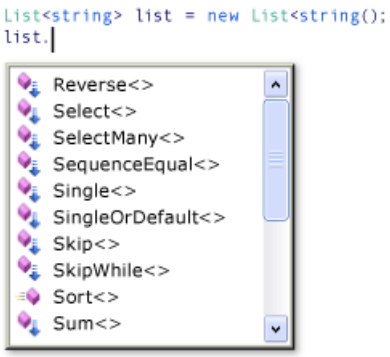
Методы расширения стандартных операторов запросов

В следующем примере показано простое *выражение запроса* и семантически эквивалентный ему запрос, написанный как *запрос, основанный на методе*.

```
class QueryVMethodSyntax
{
    static void Main()
    {
        int[] numbers = { 5, 10, 8, 3, 6, 12};

        //Query syntax:
        IEnumerable<int> numQuery1 =
            from num in numbers
            where num % 2 == 0
            orderby num
            select num;

        //Method syntax:
        IEnumerable<int> numQuery2 = numbers.Where(num => num % 2 == 0).OrderBy(n => n);

        foreach (int i in numQuery1)
        {
            Console.Write(i + " ");
        }
        Console.WriteLine(System.Environment.NewLine);
        foreach (int i in numQuery2)
        {
            Console.Write(i + " ");
        }

        // Keep the console open in debug mode.
        Console.WriteLine(System.Environment.NewLine);
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/*
    Output:
    6 8 10 12
    6 8 10 12
 */
```

Оба примера дают одинаковый результат. Видно, что тип переменной запроса в обеих формах одинаковый: IEnumerable<T>.

Чтобы разобраться в запросе, основанном на методе, изучим его более подробно. Обратите внимание на то, что в правой части выражения предложение `where` теперь выражается как метод экземпляра в объекта `numbers`, который, как вы, наверное, помните, имеет тип `IEnumerable<int>`. Если вы знакомы с универсальным интерфейсом IEnumerable<T>, то знаете, что в нем отсутствует метод `Where`. При этом, вызвав список завершения IntelliSense в Visual Studio IDE, вы увидите не только метод `Where`, но и многие другие методы, такие как `Select`, `SelectMany`, `Join` и `Orderby`. Все это — стандартные операторы запросов.



Может показаться, что класс IEnumerable<T> был переопределен и включает дополнительные методы, однако это не так. Стандартные операторы запросов реализуются как новый тип метода, который

называется *методы расширения*. Эти методы "расширяют" существующий тип и могут вызываться так, как если бы они являлись методами экземпляра для этого типа. Стандартные операторы запроса расширяют IEnumerable<T>, и поэтому вы можете написать `numbers.Where(...)`.

Чтобы приступить к использованию LINQ, о методах расширения достаточно знать только то, как ввести их в область действия в приложении, используя директивы `using`. С точки зрения приложения метод расширения и обычные методы экземпляров одинаковы.

Дополнительные сведения о методах расширения см. в разделе Методы расширения. Дополнительные сведения о стандартных операторах запросов см. в разделе Общие сведения о стандартных операторах запроса (C#). Некоторые поставщики LINQ, такие как LINQ to SQL и LINQ to XML, помимо IEnumerable<T> реализуют собственные стандартные операторы запросов и дополнительные методы расширения для других типов.

Лямбда-выражения

В предыдущем примере обратите внимание на то, что условное выражение (`num % 2 == 0`) передается в метод `Where` как встроенный аргумент: `Where(num => num % 2 == 0).`. Это встроенное выражение называется лямбда-выражением. Это удобный способ написания кода, который иначе пришлось бы записывать более громоздко: как анонимный метод, универсальный метод-делегат или дерево выражения. В C# `=>` представляет собой лямбда-оператор, который читается как "переходит в". `num` слева от оператора — входная переменная, которая соответствует переменной `num` в выражении запроса. Компилятор может вывести тип `num`, поскольку известно, что `numbers` является универсальным типом IEnumerable<T>. Тело лямбда-выражения — точно такое же, как выражение в синтаксисе запроса или в любом другом выражении или операторе C#, и может включать вызовы метода и другие сложные логические выражения. "Возвращаемое значение" — результат выражения.

Чтобы приступить к использованию LINQ, активно использовать лямбда-выражения необязательно. При этом одни запросы могут быть выражены с помощью синтаксиса запроса, в то время как другие требуют лямбда-выражений. После знакомства с лямбда-выражениями станет понятно, что они являются мощными и гибкими элементами в арсенале элементов LINQ. Дополнительные сведения см. в разделе Лямбда-выражения.

Совместимость запросов

Обратите внимание на то, что в представленном выше примере кода метод `OrderBy` вызывается с помощью оператора точки при вызове `Where`. `Where` создает отфильтрованную последовательность, а затем `Orderby` ее сортирует. Поскольку запросы возвращают `IEnumerable`, объедините их в синтаксис метода, собрав вызовы методов в цепочку. Компилятор выполняет это действие в фоновом режиме, когда вы пишете запросы, используя синтаксис запросов. А поскольку в переменной запроса результаты запроса не сохраняются, его можно в любое время изменить или использовать как базу для нового запроса даже после выполнения.

Возможности C#, поддерживающие LINQ

18.12.2021 • 3 minutes to read

В следующем разделе приведены новые конструкции языка, представленные в C# 3.0. Хотя эти новые возможности в некоторой степени используются с запросами LINQ, они не ограничиваются LINQ и могут использоваться в любом контексте, где они будут целесообразны.

Выражения запросов

Выражения запросов используют декларативный синтаксис, аналогичный SQL или XQuery, для выполнения запросов к коллекциям IEnumerable. Во время компиляции синтаксис запроса преобразуется в вызовы методов к реализации методов расширения стандартных операторов запросов поставщиком LINQ. Приложения управляют стандартными операторами запросов в области, указывая соответствующее пространство имен с помощью директивы `using`. Следующее выражение запроса принимает массив строк, группирует их в соответствии с первым символом в строке и упорядочивает группы.

```
var query = from str in stringArray
            group str by str[0] into stringGroup
            orderby stringGroup.Key
            select stringGroup;
```

Дополнительные сведения см. в разделе Выражения запросов LINQ.

Неявно типизированные переменные (var)

Вместо явного задания типа при объявлении и инициализации переменной можно использовать модификатор var, чтобы сообщить компилятору о необходимости определить и присвоить тип, как показано ниже:

```
var number = 5;
var name = "Virginia";
var query = from str in stringArray
            where str[0] == 'm'
            select str;
```

Переменные, объявленные как `var`, так же строго типизированы, как и переменные, тип которых задается явно. Использование `var` делает возможным создание анонимных типов, однако его можно использовать только для локальных переменных. Массивы также могут быть объявлены путем неявной типизации.

Дополнительные сведения см. в статье Неявно типизированные локальные переменные (руководство по программированию на C#).

Инициализаторы объектов и коллекций

Инициализаторы объектов и коллекций позволяют инициализировать объекты без явного вызова конструктора для объекта. Инициализаторы обычно используются в выражениях запросов при проецировании исходных данных в новый тип данных. При наличии, например, класса с именем `Customer` с общедоступными свойствами `Name` и `Phone` инициализатор объектов можно использовать как в

следующем примере кода:

```
var cust = new Customer { Name = "Mike", Phone = "555-1212" };
```

Продолжим рассматривать класс `Customer`. Предположим, что есть источник данных с именем `IncomingOrders` и что для каждого заказа с большим значением `OrderSize` нужно создавать класс `Customer` на основе этого заказа. Можно выполнить запрос LINQ для этого источника данных и использовать инициализацию объекта, чтобы заполнить коллекцию:

```
var newLargeOrderCustomers = from o in IncomingOrders
                             where o.OrderSize > 5
                             select new Customer { Name = o.Name, Phone = o.Phone };
```

У источника данных может быть больше скрытых свойств, чем у класса `Customer`, такого как `OrderSize`. Но инициализация объекта позволяет преобразовать данные, возвращаемые по запросу, в требуемый тип. Мы выберем данные, соответствующие нашему классу. В результате мы заполнили `IEnumerable` новыми классами `Customer`, как и требовалось. Приведенный выше код также можно представить в синтаксисе метода LINQ:

```
var newLargeOrderCustomers = IncomingOrders.Where(x => x.OrderSize > 5).Select(y => new Customer { Name =
y.Name, Phone = y.Phone });
```

Дополнительные сведения можно найти в разделе

- Инициализаторы объектов и коллекций
- Синтаксис выражений запроса для стандартных операторов запроса

Анонимные типы

Анонимный тип создается компилятором, и имя типа доступно только компилятору. Анонимные типы обеспечивают удобный способ временной группировки набора свойств в результатах запроса без необходимости определения отдельного именованного типа. Анонимные типы инициализируются с помощью нового выражения и инициализатора объектов, как показано ниже:

```
select new {name = cust.Name, phone = cust.Phone};
```

Дополнительные сведения см. в статье Анонимные типы.

Методы расширения

Метод расширения представляет собой статический метод, который может быть связан с типом, чтобы его можно было вызывать, как если бы он являлся методом экземпляра типа. Эта возможность позволяет, по сути, "добавлять" новые методы в существующие типы, фактически не изменяя их. Стандартные операторы запросов — это набор методов расширения, предоставляющий функции запросов LINQ для любого типа, реализующего IEnumerable<T>.

Дополнительные сведения см. в разделе Методы расширения.

Лямбда-выражения

Лямбда-выражение — это встроенная функция, которая использует оператор `=>` для отделения входных параметров от тела функции. Она может быть преобразована во время компиляции в делегат

или дерево выражения. В программировании LINQ вы столкнетесь с лямбда-выражениями при выполнении прямых вызовов к стандартным операторам запросов.

Дополнительные сведения см. в разделе:

- Лямбда-выражения
- Деревья выражений (C#)

См. также

- LINQ (C#)

Пошаговое руководство. Написание запросов на C# (LINQ)

18.12.2021 • 9 minutes to read

В этом пошаговом руководстве описываются возможности C#, предназначенные для написания выражений запросов LINQ.

Создание проекта C#

> **NOTE**
> Приведенные ниже инструкции относятся к Visual Studio. Если вы пользуетесь другой средой разработки, создайте консольный проект со ссылкой на библиотеку System.Core.dll и директиву `using` для пространства имен System.Linq.

Создание проекта в Visual Studio

1. Запустите Visual Studio.
2. В строке меню выберите **Файл**, **Создать**, **Проект**.

   Откроется диалоговое окно **Новый проект** .
3. Последовательно разверните узлы **Установленные**, **Шаблоны** и Visual C# , а затем выберите **Консольное приложение**.
4. В текстовое поле **Имя** введите другое имя или примите имя по умолчанию и нажмите кнопку **ОК**.

   В **обозревателе решений** появится новый проект.
5. Обратите внимание, что проект содержит ссылку на библиотеку System.Core.dll и директиву `using` для пространства имен System.Linq.

Создание источника данных в памяти

Источником данных для запросов является простой список объектов `Student`. Каждая запись `Student` включает имя, фамилию и массив целых чисел, соответствующих их баллам за проведенные в классе тесты. Скопируйте этот код в проект. Обратите внимание на следующие характеристики.

- Класс `Student` состоит из автоматически внедренных свойств.
- Каждый учащийся в списке инициализируется соответствующим инициализатором объекта.
- Сам список инициализируется инициализатором коллекции.

Вся эта структура данных инициализируется и создается без явных вызовов какого-либо конструктора или прямого доступа к членам. Дополнительные сведения об этих новых возможностях см. в разделах Автоматически внедренные свойства и Инициализаторы объектов и коллекций.

Добавление источника данных

- Добавьте класс `Student` и инициализированный список учащихся в класс `Program` в проекте.

```
public class Student
{
    public string First { get; set; }
    public string Last { get; set; }
    public int ID { get; set; }
    public List<int> Scores;
}

// Create a data source by using a collection initializer.
static List<Student> students = new List<Student>
{
    new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int> {97, 92, 81,
60}},
    new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int> {75, 84, 91, 39}},
    new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int> {88, 94, 65, 91}},
    new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int> {97, 89, 85, 82}},
    new Student {First="Debra", Last="Garcia", ID=115, Scores= new List<int> {35, 72, 91, 70}},
    new Student {First="Fadi", Last="Fakhouri", ID=116, Scores= new List<int> {99, 86, 90, 94}},
    new Student {First="Hanying", Last="Feng", ID=117, Scores= new List<int> {93, 92, 80, 87}},
    new Student {First="Hugo", Last="Garcia", ID=118, Scores= new List<int> {92, 90, 83, 78}},
    new Student {First="Lance", Last="Tucker", ID=119, Scores= new List<int> {68, 79, 88, 92}},
    new Student {First="Terry", Last="Adams", ID=120, Scores= new List<int> {99, 82, 81, 79}},
    new Student {First="Eugene", Last="Zabokritski", ID=121, Scores= new List<int> {96, 85, 91, 60}},
    new Student {First="Michael", Last="Tucker", ID=122, Scores= new List<int> {94, 92, 91, 91}}
};
```

Добавление нового учащегося в список учащихся

1. Добавьте новый объект `Student` в список `Students` и введите любое имя и результаты тестирования. Чтобы лучше изучить синтаксис инициализатора объекта, постарайтесь заполнить все данные нового учащегося.

Создание запроса

Создание простого запроса

- В методе `Main` приложения создайте простой запрос, при выполнении которого будет возвращаться список учащихся, набравших в результате тестирования больше 90 баллов. Поскольку выбран весь объект `Student`, запрос имеет тип `IEnumerable<Student>`. Несмотря на то, что код можно также набрать напрямую, используя ключевое слово var, для демонстрации результатов применяется неявная типизация. (Дополнительные сведения о `var` см. в разделе Неявно типизированные локальные переменные.)

  Кроме того переменная диапазона в запросе, `student`, служит ссылкой на каждый объект `Student` в источнике и предоставляет доступ к каждому объекту.

```
// Create the query.
// The first line could also be written as "var studentQuery ="
IEnumerable<Student> studentQuery =
    from student in students
    where student.Scores[0] > 90
    select student;
```

Выполнение запроса

Порядок выполнения запроса

1. Теперь напишите цикл `foreach`, вызывающий выполнение запроса. Обратите внимание на следующие моменты:

   - Доступ к каждому элементу в возвращаемой последовательности осуществляется с

помощью переменной итерации в цикле `foreach`.

- Эта переменная имеет тип `Student`, а переменная запроса — совместимый тип `IEnumerable<Student>`.

2. Добавив код, соберите и запустите приложение, чтобы отобразить результаты в окне **Консоль**.

```
// Execute the query.
// var could be used here also.
foreach (Student student in studentQuery)
{
    Console.WriteLine("{0}, {1}", student.Last, student.First);
}

// Output:
// Omelchenko, Svetlana
// Garcia, Cesar
// Fakhouri, Fadi
// Feng, Hanying
// Garcia, Hugo
// Adams, Terry
// Zabokritski, Eugene
// Tucker, Michael
```

Добавление другого условия фильтра

1. Чтобы сделать запрос более точным, можно объединить несколько логических условий в предложение `where`. Следующий код добавляет условие, согласно которому запрос возвращает учащихся, которые по первому тесту набрали больше 90 баллов, а по последнему — меньше 80 баллов. Предложение `where` должно быть аналогично приведенному ниже коду.

   ```
   where student.Scores[0] > 90 && student.Scores[3] < 80
   ```

   Дополнительные сведения см. в разделе Предложение where.

Изменение запроса

Упорядочение результатов

1. С результатами проще работать, если они упорядочены. Возвращаемую последовательность можно упорядочить по любому доступному полю в исходных элементах. Например, следующее предложение `orderby` упорядочивает результаты в алфавитном порядке от A до Z, используя фамилии учащихся. Добавьте к запросу следующее предложение `orderby`, указав его после оператора `where` и перед оператором `select`:

   ```
   orderby student.Last ascending
   ```

2. Теперь измените предложение `orderby` таким образом, чтобы результаты были упорядочены по баллам за первый тест в обратном порядке, от наибольшего к наименьшему.

   ```
   orderby student.Scores[0] descending
   ```

3. Измените строку формата `WriteLine` таким образом, чтобы отобразить баллы:

   ```
   Console.WriteLine("{0}, {1} {2}", student.Last, student.First, student.Scores[0]);
   ```

Дополнительные сведения см. в разделе Предложение orderby.

Группировка результатов

1. Группировка представляет собой полезную возможность в выражениях запросов. Запрос с предложением group создает последовательность групп, в которой каждая группа содержит `Key`, и последовательность, состоящую из всех членов этой группы. Представленный ниже запрос группирует учащихся, используя в качестве ключа первую букву фамилии.

```
// studentQuery2 is an IEnumerable<IGrouping<char, Student>>
var studentQuery2 =
    from student in students
    group student by student.Last[0];
```

2. Обратите внимание на то, что тип запроса изменился. Теперь он создает последовательность групп с типом `char` в качестве ключа и последовательность объектов `Student`. Поскольку тип запроса изменился, следующий код изменяет также цикл выполнения `foreach`:

```
// studentGroup is a IGrouping<char, Student>
foreach (var studentGroup in studentQuery2)
{
    Console.WriteLine(studentGroup.Key);
    foreach (Student student in studentGroup)
    {
        Console.WriteLine("   {0}, {1}",
                  student.Last, student.First);
    }
}

// Output:
// O
//   Omelchenko, Svetlana
//   O'Donnell, Claire
// M
//   Mortensen, Sven
// G
//   Garcia, Cesar
//   Garcia, Debra
//   Garcia, Hugo
// F
//   Fakhouri, Fadi
//   Feng, Hanying
// T
//   Tucker, Lance
//   Tucker, Michael
// A
//   Adams, Terry
// Z
//   Zabokritski, Eugene
```

3. Запустите приложение и просмотрите результаты в окне **Консоль**.

Дополнительные сведения см. в разделе Предложение group.

Преобразование переменных в явно типизированные

1. Набирать код `IEnumerables` в `IGroupings` напрямую — далеко не самое увлекательное занятие. Написать тот же запрос и цикл `foreach` можно быстрее и проще, воспользовавшись переменной `var`. Ключевое слово `var` не приводит к изменению типов объектов, оно просто сообщает компилятору о том, что он должен вывести типы логически. Измените тип `studentQuery` и переменную итерации `group` на `var` и выполните запрос заново. Обратите внимание на то, что во внутреннем цикле `foreach` переменная итерации по-прежнему типизируется как `Student`, а запрос

работает так же, как раньше. Измените переменную итерации `student` на `var` и выполните запрос заново. Результаты будут точно такими же.

```
var studentQuery3 =
    from student in students
    group student by student.Last[0];

foreach (var groupOfStudents in studentQuery3)
{
    Console.WriteLine(groupOfStudents.Key);
    foreach (var student in groupOfStudents)
    {
        Console.WriteLine("   {0}, {1}",
            student.Last, student.First);
    }
}

// Output:
// O
//   Omelchenko, Svetlana
//   O'Donnell, Claire
// M
//   Mortensen, Sven
// G
//   Garcia, Cesar
//   Garcia, Debra
//   Garcia, Hugo
// F
//   Fakhouri, Fadi
//   Feng, Hanying
// T
//   Tucker, Lance
//   Tucker, Michael
// A
//   Adams, Terry
// Z
//   Zabokritski, Eugene
```

Дополнительные сведения о переменной var см. в разделе Неявно типизированные локальные переменные.

Упорядочение групп по значению ключа

1. Выполняя предыдущий запрос, вы могли заметить, что группы отображаются не в алфавитном порядке. Для того чтобы это изменить, необходимо указать предложение `orderby` после предложения `group`. Но, чтобы использовать предложение `orderby`, нужен идентификатор, служащий в качестве ссылки на группы, создаваемые предложением `group`. Укажите идентификатор с помощью ключевого слова `into`, как показано ниже:

```
var studentQuery4 =
    from student in students
    group student by student.Last[0] into studentGroup
    orderby studentGroup.Key
    select studentGroup;

foreach (var groupOfStudents in studentQuery4)
{
    Console.WriteLine(groupOfStudents.Key);
    foreach (var student in groupOfStudents)
    {
        Console.WriteLine("   {0}, {1}",
            student.Last, student.First);
    }
}

// Output:
//A
//   Adams, Terry
//F
//   Fakhouri, Fadi
//   Feng, Hanying
//G
//   Garcia, Cesar
//   Garcia, Debra
//   Garcia, Hugo
//M
//   Mortensen, Sven
//O
//   Omelchenko, Svetlana
//   O'Donnell, Claire
//T
//   Tucker, Lance
//   Tucker, Michael
//Z
//   Zabokritski, Eugene
```

После выполнения этого запроса группы будут отсортированы в алфавитном порядке.

Введение идентификаторов с помощью предложения let

1. Ключевое слово `let` позволяет ввести идентификатор для любого результата в выражении запроса. Этот идентификатор может применяться для удобства, как в следующем примере, или повышать производительность, сохраняя результаты выражения, чтобы не вычислять их повторно.

```
// studentQuery5 is an IEnumerable<string>
// This query returns those students whose
// first test score was higher than their
// average score.
var studentQuery5 =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    where totalScore / 4 < student.Scores[0]
    select student.Last + " " + student.First;

foreach (string s in studentQuery5)
{
    Console.WriteLine(s);
}

// Output:
// Omelchenko Svetlana
// O'Donnell Claire
// Mortensen Sven
// Garcia Cesar
// Fakhouri Fadi
// Feng Hanying
// Garcia Hugo
// Adams Terry
// Zabokritski Eugene
// Tucker Michael
```

Дополнительные сведения см. в разделе Предложение let.

Использование синтаксиса метода в выражении запроса

1. Как описано в разделе Синтаксис запросов и синтаксис методов в LINQ, некоторые операции запросов можно выразить, только используя синтаксис метода. Представленный ниже код вычисляет общий балл для каждого объекта `Student` в исходной последовательности, а затем применяет метод `Average()` к результатам запроса, чтобы рассчитать средний балл класса.

```
var studentQuery6 =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    select totalScore;

double averageScore = studentQuery6.Average();
Console.WriteLine("Class average score = {0}", averageScore);

// Output:
// Class average score = 334.166666666667
```

Преобразование или проецирование в предложение select

1. Запрос очень часто выдает последовательность, элементы которой отличаются от элементов в исходной последовательности. Удалите или закомментируйте предыдущий запрос и цикл выполнения и замените его на представленный ниже код. Обратите внимание на то, что запрос возвращает последовательность строк (не `Students`), и этот факт отражается в цикле `foreach`.

```
IEnumerable<string> studentQuery7 =
    from student in students
    where student.Last == "Garcia"
    select student.First;

Console.WriteLine("The Garcias in the class are:");
foreach (string s in studentQuery7)
{
    Console.WriteLine(s);
}

// Output:
// The Garcias in the class are:
// Cesar
// Debra
// Hugo
```

2. Код, представленный выше в этом пошаговом руководстве, показывает, что среднее количество баллов по классу составляет около 334. Для создания последовательности `Students`, сумма баллов в которой будет выше, чем средний показатель по классу, с указанием `Student ID` можно вставить в оператор `select` анонимный тип:

```
var studentQuery8 =
    from student in students
    let x = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    where x > averageScore
    select new { id = student.ID, score = x };

foreach (var item in studentQuery8)
{
    Console.WriteLine("Student ID: {0}, Score: {1}", item.id, item.score);
}

// Output:
// Student ID: 113, Score: 338
// Student ID: 114, Score: 353
// Student ID: 116, Score: 369
// Student ID: 117, Score: 352
// Student ID: 118, Score: 343
// Student ID: 120, Score: 341
// Student ID: 122, Score: 368
```

Следующие шаги

Ознакомившись с основными аспектами работы с запросами в C#, вы будете готовы прочитать документацию и примеры по интересующему вас типу поставщика LINQ:

LINQ to SQL

LINQ to DataSet

LINQ to XML (C#)

LINQ to Objects (C#)

См. также

- LINQ (C#)
- Выражения запросов LINQ

Общие сведения о стандартных операторах запроса (C#)

18.12.2021 • 3 minutes to read

Стандартные операторы запросов — это методы, формирующие шаблон LINQ. Большинство этих методов работают с последовательностями. В данном контексте последовательность — это объект, тип которого реализует интерфейс IEnumerable<T> или IQueryable<T>. Функциональные возможности стандартных операторов запросов включают фильтрацию, проекцию, статистическую обработку, сортировку и многие другие.

Существует два набора стандартных операторов запросов LINQ, один из них работает с объектами типа IEnumerable<T>, а другой — с объектами типа IQueryable<T>. Методы, входящие в каждый из наборов, являются статическими членами классов Enumerable и Queryable соответственно. Они определяются как *методы расширения* типа, к которому применяются. Методы расширения можно вызывать, используя либо синтаксис статического метода, либо синтаксис метода экземпляра.

Кроме того, несколько стандартных методов операторов запроса работают с типами, отличными от основанных на IEnumerable<T> или IQueryable<T>. Тип Enumerable определяет два таких метода, которые работают с объектами типа IEnumerable. Эти методы (Cast<TResult>(IEnumerable) и OfType<TResult>(IEnumerable)) позволяют выполнять запрос к непараметризованным или не являющимся универсальными коллекциям с использованием шаблона LINQ. Для этого создается строго типизированная коллекция объектов. Класс Queryable определяет два схожих метода (Cast<TResult>(IQueryable) и OfType<TResult>(IQueryable)), которые работают с объектами типа IQueryable.

Стандартные операторы запросов имеют различное время выполнения, которое зависит от того, возвращают они одноэлементное значение или последовательность значений. Методы, возвращающие одноэлементное значение (например, Average и Sum), выполняются одновременно. Методы, которые возвращают последовательность, откладывают выполнение запроса и возвращают перечисляемый объект.

При использовании методов, которые применяются к коллекциям, т. е. методов, которые расширяют IEnumerable<T>, возвращаемый перечисляемый объект захватывает переданные в этот метод аргументы. При перечислении этого объекта задействуется логика оператора запросов и возвращаются результаты запросов.

Напротив, методы, расширяющие IQueryable<T>, не реализуют никаких поведений запросов. Они создают дерево выражений, представляющее выполняемый запрос. Запрос обрабатывается объектом IQueryable<T> источника.

Вызовы методов запросов можно объединять в один запрос, в результате чего запросы могут становиться довольно сложными.

В следующем примере кода показано, как использовать стандартные операторы запросов для получения сведений о последовательности.

```
string sentence = "the quick brown fox jumps over the lazy dog";
// Split the string into individual words to create a collection.
string[] words = sentence.Split(' ');

// Using query expression syntax.
var query = from word in words
            group word.ToUpper() by word.Length into gr
            orderby gr.Key
            select new { Length = gr.Key, Words = gr };

// Using method-based query syntax.
var query2 = words.
    GroupBy(w => w.Length, w => w.ToUpper()).
    Select(g => new { Length = g.Key, Words = g }).
    OrderBy(o => o.Length);

foreach (var obj in query)
{
    Console.WriteLine("Words of length {0}:", obj.Length);
    foreach (string word in obj.Words)
        Console.WriteLine(word);
}

// This code example produces the following output:
//
// Words of length 3:
// THE
// FOX
// THE
// DOG
// Words of length 4:
// OVER
// LAZY
// Words of length 5:
// QUICK
// BROWN
// JUMPS
```

Синтаксис выражений запросов

Некоторые из наиболее часто используемых стандартных операторов запросов имеют представление в виде ключевых слов в синтаксисе языка C# и Visual Basic, что позволяет вызывать их как часть *выражения запроса*. Дополнительные сведения о стандартных операторах запросов с выделенными ключевыми словами и соответствующим синтаксисом см. в разделе Синтаксис выражений запросов для стандартных операторов запросов (C#).

Расширение стандартных операторов запросов

Набор стандартных операторов запросов можно дополнить, создав специальные методы, соответствующие вашему целевому домену или технологии. Кроме того, можно заменить стандартные операторы запросов на собственные реализации, предоставляющие дополнительные услуги, такие как удаленное вычисление, перевод запросов и оптимизация. Пример см. в разделе AsEnumerable.

Связанные разделы

Следующие ссылки адресуют к статьям, содержащим дополнительные сведения о различных стандартных операторах запросов в зависимости от их функциональности.

Сортировка данных (C#)

Операции над множествами (C#)

Фильтрация данных (C#)

Операции, использующие квантификаторы (C#)

Операции проекции (C#)

Секционирование данных (C#)

Операции соединения (C#)

Группирование данных (C#)

Операции создания (C#)

Операции сравнения (C#)

Операции с элементами (C#)

Преобразование типов данных (C#)

Операции объединения (C#)

Операции агрегирования (C#)

См. также

- Enumerable
- Queryable
- Введение в запросы LINQ (C#)
- Синтаксис выражений запроса для стандартных операторов запроса (C#)
- Классификация стандартных операторов запросов по способу выполнения (C#)
- Методы расширения

Синтаксис выражений запроса для стандартных операторов запроса (C#)

18.12.2021 • 2 minutes to read

Некоторые из наиболее часто используемых стандартных операторов запросов имеют представление в виде ключевых слов в синтаксисе языка C#, что позволяет вызывать их как часть *выражения запроса*. Выражение запроса является более удобочитаемой формой задания запроса, чем его *основанный на методах* эквивалент. Предложения выражений запросов преобразуются в вызовы методов запросов во время компиляции.

Таблица синтаксиса выражений запросов

В следующей таблице приводится список стандартных операторов запросов, имеющих эквивалентные предложения выражений запросов.

МЕТОД	СИНТАКСИС ВЫРАЖЕНИЯ ЗАПРОСА C#
Cast	Использование явным образом типизированной переменной диапазона, например: `from int i in numbers` (Дополнительные сведения см. в разделе Предложение from.)

GroupBy	`group … by`
	-или-
	`group … by … into …`
	(Дополнительные сведения см. в разделе Предложение group.)

GroupJoin<TOuter,TInner,TKey,TResult> (IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>,TResult>)	`join … in … on … equals … into …` (Дополнительные сведения см. в разделе Предложение join.)
Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>)	`join … in … on … equals …` (Дополнительные сведения см. в разделе Предложение join.)
OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	`orderby` (Дополнительные сведения см. в разделе Предложение orderby.)

МЕТОД	СИНТАКСИС ВЫРАЖЕНИЯ ЗАПРОСА C#
OrderByDescending<TSource,TKey> (IEnumerable<TSource>, Func<TSource,TKey>)	`orderby … descending` (Дополнительные сведения см. в разделе Предложение orderby.)
Select	`select` (Дополнительные сведения см. в разделе Предложение select.)
SelectMany	Несколько предложений `from`. (Дополнительные сведения см. в разделе Предложение from.)
ThenBy<TSource,TKey>(IOrderedEnumerable<TSource>, Func<TSource,TKey>)	`orderby …, …` (Дополнительные сведения см. в разделе Предложение orderby.)
ThenByDescending<TSource,TKey> (IOrderedEnumerable<TSource>, Func<TSource,TKey>)	`orderby …, … descending` (Дополнительные сведения см. в разделе Предложение orderby.)
Where	`where` (Дополнительные сведения см. в разделе Предложение where.)

См. также

- Enumerable
- Queryable
- Общие сведения о стандартных операторах запроса (C#)
- Классификация стандартных операторов запросов по способу выполнения (C#)

Классификация стандартных операторов запросов по способу выполнения (C#)

18.12.2021 • 2 minutes to read

В реализации LINQ to Objects выполнение методов стандартных операторов запросов бывает немедленным и отложенным. Операторы запросов, использующие отложенное выполнение, можно дополнительно разделить на две категории: потоковые и непотоковые. Если вы знаете, каким образом выполняются разные операторы запросов, это может помочь понять результаты, полученные из данного запроса. Это особенно справедливо при изменении источника данных или создании одного запроса поверх другого. В этом разделе представлена классификация стандартных операторов запросов по способу выполнения.

Способ выполнения

Интерпретация

Немедленное выполнение означает, что источник данных считывается и операция выполняется в той точке кода, где объявлен запрос. Все стандартные операторы запросов, возвращающие один, неперечислимый результат, выполняются немедленно.

Отложено

Отложенное выполнение означает, что операция не выполняется в той точке кода, где объявлен запрос. Она выполняется только после перечисления переменной запроса, например с помощью оператора `foreach`. Это означает, что результаты выполнения запроса зависят от содержимого источника данных при выполнении запроса, а не при его определении. Если переменная запроса перечисляется несколько раз, результаты могут каждый раз отличаться. Практически все стандартные операторы запроса, которые возвращают значения типа IEnumerable<T> или IOrderedEnumerable<TElement>, выполняются отложенным способом.

Операторы запросов, использующие отложенное выполнение, можно дополнительно разделить на потоковые и непотоковые.

Потоковые операторы

Потоковые операторы не считывают все исходные данные до создания элементов. Во время выполнения потоковый оператор выполняет свою операцию с каждым исходным элементом по мере считывания и при необходимости создает элемент. Потоковый оператор продолжает считывание исходных элементов до того момента, когда можно будет создать итоговый элемент. Это означает, что для получения одного итогового элемента может быть считано несколько исходных элементов.

Непотоковые

Непотоковые операторы должны считать все исходные данные до создания итогового элемента. В эту категорию попадают операции сортировки и группировки. Во время выполнения непотоковые операторы запросов считывают все исходные данные, помещают их в структуру данных, выполняют операцию и создают итоговые элементы.

Таблица классификации

В следующей таблице приведена классификация всех методов стандартных операторов запросов по способу выполнения.

NOTE

Если оператор помечен в двух столбцах, в операции участвуют две входные последовательности, и каждая последовательность вычисляется по-разному. В таких случаях первая последовательность в списке параметров всегда вычисляется отложенным потоковым способом.

СТАНДАРТНЫЙ ОПЕРАТОР ЗАПРОСА	ВОЗВРАЩАЕМЫЙ ТИП	НЕМЕДЛЕННОЕ ВЫПОЛНЕНИЕ	ОТЛОЖЕННОЕ ПОТОКОВОЕ ВЫПОЛНЕНИЕ	ОТЛОЖЕННОЕ НЕПОТОКОВОЕ ВЫПОЛНЕНИЕ
Aggregate	TSource	X		
All	Boolean	X		
Any	Boolean	X		
AsEnumerable	IEnumerable<T>		X	
Average	Одно числовое значение	X		
Cast	IEnumerable<T>		X	
Concat	IEnumerable<T>		X	
Contains	Boolean	X		
Count	Int32	X		
DefaultIfEmpty	IEnumerable<T>		X	
Distinct	IEnumerable<T>		X	
ElementAt	TSource	X		
ElementAtOrDefault	TSource	X		
Empty	IEnumerable<T>	X		
Except	IEnumerable<T>		X	X
First	TSource	X		
FirstOrDefault	TSource	X		
GroupBy	IEnumerable<T>			X
GroupJoin	IEnumerable<T>		X	X
Intersect	IEnumerable<T>		X	X
Join	IEnumerable<T>		X	X

СТАНДАРТНЫЙ ОПЕРАТОР ЗАПРОСА	ВОЗВРАЩАЕМЫЙ ТИП	НЕМЕДЛЕННОЕ ВЫПОЛНЕНИЕ	ОТЛОЖЕННОЕ ПОТОКОВОЕ ВЫПОЛНЕНИЕ	ОТЛОЖЕННОЕ НЕПОТОКОВОЕ ВЫПОЛНЕНИЕ
Last	TSource	X		
LastOrDefault	TSource	X		
LongCount	Int64	X		
Max	Одно числовое значение, TSource или TResult	X		
Min	Одно числовое значение, TSource или TResult	X		
OfType	IEnumerable<T>		X	
OrderBy	IOrderedEnumerable <TElement>			X
OrderByDescending	IOrderedEnumerable <TElement>			X
Range	IEnumerable<T>		X	
Repeat	IEnumerable<T>		X	
Reverse	IEnumerable<T>			X
Select	IEnumerable<T>		X	
SelectMany	IEnumerable<T>		X	
SequenceEqual	Boolean	X		
Single	TSource	X		
SingleOrDefault	TSource	X		
Skip	IEnumerable<T>		X	
SkipWhile	IEnumerable<T>		X	
Sum	Одно числовое значение	X		
Take	IEnumerable<T>		X	
TakeWhile	IEnumerable<T>		X	

СТАНДАРТНЫЙ ОПЕРАТОР ЗАПРОСА	ВОЗВРАЩАЕМЫЙ ТИП	НЕМЕДЛЕННОЕ ВЫПОЛНЕНИЕ	ОТЛОЖЕННОЕ ПОТОКОВОЕ ВЫПОЛНЕНИЕ	ОТЛОЖЕННОЕ НЕПОТОКОВОЕ ВЫПОЛНЕНИЕ
ThenBy	IOrderedEnumerable<TElement>			X
ThenByDescending	IOrderedEnumerable<TElement>			X
ToArray	Массив TSource	X		
ToDictionary	Dictionary<TKey,TValue>	X		
ToList	IList<T>	X		
ToLookup	ILookup<TKey,TElement>	X		
Union	IEnumerable<T>		X	
Where	IEnumerable<T>		X	

См. также

- Enumerable
- Общие сведения о стандартных операторах запроса (C#)
- Синтаксис выражений запроса для стандартных операторов запроса (C#)
- LINQ to Objects (C#)

Сортировка данных (C#)

18.12.2021 • 2 minutes to read

Операция сортировки упорядочивает элементы последовательности на основе одного или нескольких атрибутов. Первый критерий сортировки выполняет первичную сортировку элементов. Указав второй критерий поиска, можно сортировать элементы внутри каждой группы первичной сортировки.

На следующем рисунке показаны результаты операции сортировки в алфавитном порядке в последовательности символов:

Далее перечислены методы стандартных операторов запроса, которые выполняют сортировку данных.

Методы

ИМЯ МЕТОДА	ОПИСАНИЕ	СИНТАКСИС ВЫРАЖЕНИЯ ЗАПРОСА C#	ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ
OrderBy	Сортировка значений в возрастающем порядке.	`orderby`	Enumerable.OrderBy Queryable.OrderBy
OrderByDescending	Сортировка значений в убывающем порядке.	`orderby … descending`	Enumerable.OrderByDescending Queryable.OrderByDescending
ThenBy	Дополнительная сортировка по возрастанию.	`orderby …, …`	Enumerable.ThenBy Queryable.ThenBy
ThenByDescending	Дополнительная сортировка по убыванию.	`orderby …, … descending`	Enumerable.ThenByDescending Queryable.ThenByDescending
Reverse	Изменение порядка элементов в коллекции на обратный.	Не применяется	Enumerable.Reverse Queryable.Reverse

Примеры синтаксиса выражений запросов

Примеры основной сортировки

Основная сортировка по возрастанию

В следующем примере показано использование предложения `orderby` в запросе LINQ для сортировки строк в массиве по длине строки в порядке возрастания.

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                            orderby word.Length
                            select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    the
    fox
    quick
    brown
    jumps
*/
```

Основная сортировка по убыванию

В следующем примере показано использование предложения `orderby descending` в запросе LINQ для сортировки строк по их первой букве в порядке убывания.

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                            orderby word.Substring(0, 1) descending
                            select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    the
    quick
    jumps
    fox
    brown
*/
```

Примеры дополнительной сортировки

Дополнительная сортировка по возрастанию

В следующем примере показано использование предложения `orderby` в запросе LINQ для выполнения основной и дополнительной сортировки строк в массиве. Строки сортируются основным образом по длине и дополнительно — по первой букве строки; в обоих случаях в возрастающем порядке.

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                            orderby word.Length, word.Substring(0, 1)
                            select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    fox
    the
    brown
    jumps
    quick
*/
```

Дополнительная сортировка по убыванию

В следующем примере показано использование предложения `orderby descending` в запросе LINQ для выполнения основной сортировки по возрастанию и дополнительной сортировки по убыванию. Строки сортируются основным образом по длине и дополнительно — по первой букве строки.

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                            orderby word.Length, word.Substring(0, 1) descending
                            select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    the
    fox
    quick
    jumps
    brown
*/
```

См. также

- System.Linq
- Общие сведения о стандартных операторах запроса (C#)
- предложение orderby
- Упорядочение результатов предложения соединения
- Практическое руководство. Сортировка или фильтрация текстовых данных по любому слову или полю (LINQ) (C#)

Операции над множествами (C#)

18.12.2021 • 6 minutes to read

Операции над множествами в LINQ — это операции запросов, результирующие наборы которых основываются на наличии или отсутствии эквивалентных элементов в одной или другой коллекции (или наборе).

Методы стандартных операторов запросов, которые выполняют операции над множествами, перечислены в следующем разделе.

Методы

ИМЕНА МЕТОДОВ	ОПИСАНИЕ	СИНТАКСИС ВЫРАЖЕНИЯ ЗАПРОСА C#	ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ
Distinct или DistinctBy	Удаляет повторяющиеся значения из коллекции.	Не применяется	Enumerable.Distinct Enumerable.DistinctBy Queryable.Distinct Queryable.DistinctBy
Except или ExceptBy	Возвращает разность множеств, т. е. элементы одной коллекции, которые отсутствуют во второй.	Не применяется	Enumerable.Except Enumerable.ExceptBy Queryable.Except Queryable.ExceptBy
Intersect или IntersectBy	Возвращает пересечение множеств, т. е. элементы, присутствующие в каждой из двух коллекций.	Не применяется	Enumerable.Intersect Enumerable.IntersectBy Queryable.Intersect Queryable.IntersectBy
Union или UnionBy	Возвращает объединение множеств, т. е. уникальные элементы, присутствующие в одной из двух коллекций.	Не применяется	Enumerable.Union Enumerable.UnionBy Queryable.Union Queryable.UnionBy

Примеры

В некоторых из приведенных ниже примеров используется тип `record`, который представляет планеты Солнечной системы.

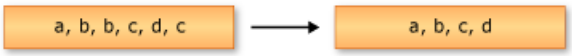
```
namespace SolarSystem;

record Planet(
    string Name,
    PlanetType Type,
    int OrderFromSun)
{
    public static readonly Planet Mercury =
        new(nameof(Mercury), PlanetType.Rock, 1);

    public static readonly Planet Venus =
        new(nameof(Venus), PlanetType.Rock, 2);

    public static readonly Planet Earth =
        new(nameof(Earth), PlanetType.Rock, 3);

    public static readonly Planet Mars =
        new(nameof(Mars), PlanetType.Rock, 4);

    public static readonly Planet Jupiter =
        new(nameof(Jupiter), PlanetType.Gas, 5);

    public static readonly Planet Saturn =
        new(nameof(Saturn), PlanetType.Gas, 6);

    public static readonly Planet Uranus =
        new(nameof(Uranus), PlanetType.Liquid, 7);

    public static readonly Planet Neptune =
        new(nameof(Neptune), PlanetType.Liquid, 8);

    // Yes, I know... not technically a planet anymore
    public static readonly Planet Pluto =
        new(nameof(Pluto), PlanetType.Ice, 9);
}
```

Для создания экземпляра позиционной записи `record Planet` требуются аргументы `Name`, `Type` и `OrderFromSun`. Существует несколько экземпляров планет `static readonly` типа `Planet`. Это удобные определения для хорошо известных планет. Элемент `Type` определяет тип планеты.

```
namespace SolarSystem;

enum PlanetType
{
    Rock,
    Ice,
    Gas,
    Liquid
};
```

`Distinct` и `DistinctBy`

В следующем примере показано поведение метода Enumerable.Distinct применительно к последовательности строк. Возвращаемая последовательность содержит уникальные элементы из входной последовательности.



```
string[] planets = { "Mercury", "Venus", "Venus", "Earth", "Mars", "Earth" };

IEnumerable<string> query = from planet in planets.Distinct()
                            select planet;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Mercury
 * Venus
 * Earth
 * Mars
 */
```

Вместо `Distinct` можно использовать `DistinctBy`, который принимает `keySelector`. `keySelector` используется как сравнительный дискриминатор для типа источника. Давайте рассмотрим следующий массив планет:

```
Planet[] planets =
{
    Planet.Mercury,
    Planet.Venus,
    Planet.Earth,
    Planet.Mars,
    Planet.Jupiter,
    Planet.Saturn,
    Planet.Uranus,
    Planet.Neptune,
    Planet.Pluto
};
```

Планеты различаются по значению `PlanetType`, и отображается первый из их типов:

```
foreach (Planet planet in planets.DistinctBy(p => p.Type))
{
    Console.WriteLine(planet);
}

// This code produces the following output:
//     Planet { Name = Mercury, Type = Rock, OrderFromSun = 1 }
//     Planet { Name = Jupiter, Type = Gas, OrderFromSun = 5 }
//     Planet { Name = Uranus, Type = Liquid, OrderFromSun = 7 }
//     Planet { Name = Pluto, Type = Ice, OrderFromSun = 9 }
```

В приведенном выше коде C#:

- Массив `Planet` фильтруется по первому вхождению каждого уникального типа плоскости.
- Полученный экземпляр `planet` записывается в консоль.

`Except` И `ExceptBy`

В следующем примере показано поведение Enumerable.Except. Возвращаемая последовательность содержит только те элементы из первой входной последовательности, которых нет во второй.

```
string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

IEnumerable<string> query = from planet in planets1.Except(planets2)
                            select planet;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Venus
 */
```

Вместо `Except` можно использовать `ExceptBy`, который принимает две последовательности разнородных типов и `keySelector`. Тип `keySelector` совпадает с типом второй коллекции и используется как сравнительный дискриминатор для типа источника. Давайте рассмотрим следующий массив планет:

```
Planet[] planets =
{
    Planet.Mercury,
    Planet.Venus,
    Planet.Earth,
    Planet.Jupiter
};

Planet[] morePlanets =
{
    Planet.Mercury,
    Planet.Earth,
    Planet.Mars,
    Planet.Jupiter
};
```

Чтобы найти планеты из первой коллекции, которые не входят во вторую коллекцию, можно проецировать имена планет в виде коллекции `second` с прежним значением `keySelector`:

```
// A shared "keySelector"
static string PlanetNameSelector(Planet planet) => planet.Name;

foreach (Planet planet in
    planets.ExceptBy(
        morePlanets.Select(PlanetNameSelector), PlanetNameSelector))
{
    Console.WriteLine(planet);
}

// This code produces the following output:
//     Planet { Name = Venus, Type = Rock, OrderFromSun = 2 }
```

В приведенном выше коде C#:

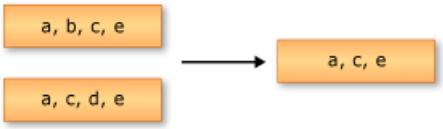
- `keySelector` определяется как локальная функция `static`, которая различает имена планет.
- Первый массив планет фильтруется, чтобы получить отсутствующие во втором массиве планеты, на

основе имен планет.

- Полученный экземпляр `planet` записывается в консоль.

`Intersect` и `IntersectBy`

В следующем примере показано поведение Enumerable.Intersect. Возвращаемая последовательность содержит элементы, общие для обеих входных последовательностей.




```
string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

IEnumerable<string> query = from planet in planets1.Intersect(planets2)
                            select planet;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Mercury
 * Earth
 * Jupiter
 */
```

Вместо `Intersect` можно использовать `IntersectBy`, который принимает две последовательности разнородных типов и `keySelector`. `keySelector` используется как сравнительный дискриминатор для типа второй коллекции. Давайте рассмотрим следующий массив планет:

```
Planet[] firstFivePlanetsFromTheSun =
{
    Planet.Mercury,
    Planet.Venus,
    Planet.Earth,
    Planet.Mars,
    Planet.Jupiter
};

Planet[] lastFivePlanetsFromTheSun =
{
    Planet.Mars,
    Planet.Jupiter,
    Planet.Saturn,
    Planet.Uranus,
    Planet.Neptune
};
```

Существует два массива планет, один из которых представляет первые пять планет от Солнца, а второй — последние пять планет. Так как тип `Planet` является позиционным типом `record`, вы можете использовать его семантику сравнения значений в формате `keySelector`:

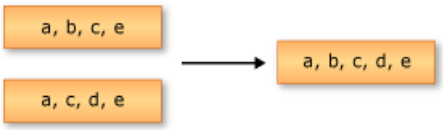
```
foreach (Planet planet in
    firstFivePlanetsFromTheSun.IntersectBy(
        lastFivePlanetsFromTheSun, planet => planet))
{
    Console.WriteLine(planet);
}

// This code produces the following output:
//     Planet { Name = Mars, Type = Rock, OrderFromSun = 4 }
//     Planet { Name = Jupiter, Type = Gas, OrderFromSun = 5 }
```

В приведенном выше коде C#:

- Два массива `Planet` пересекаются с использованием семантики сравнения их значений.
- В результирующей последовательности содержатся только те планеты, которые обнаружены в обоих массивах.
- Полученный экземпляр `planet` записывается в консоль.

`Union` И `UnionBy`

В следующем примере показана операция объединения двух последовательностей строк. Возвращаемая последовательность содержит уникальные элементы из обеих входных последовательностей.




```
string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

IEnumerable<string> query = from planet in planets1.Union(planets2)
                            select planet;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Mercury
 * Venus
 * Earth
 * Jupiter
 * Mars
 */
```

Вместо `Union` можно использовать `UnionBy`, который принимает две последовательности одного типа и `keySelector`. `keySelector` используется как сравнительный дискриминатор для типа источника. Давайте рассмотрим следующий массив планет:

```
Planet[] firstFivePlanetsFromTheSun =
{
    Planet.Mercury,
    Planet.Venus,
    Planet.Earth,
    Planet.Mars,
    Planet.Jupiter
};

Planet[] lastFivePlanetsFromTheSun =
{
    Planet.Mars,
    Planet.Jupiter,
    Planet.Saturn,
    Planet.Uranus,
    Planet.Neptune
};
```

Чтобы объединить эти две коллекции в одну последовательность, укажите `keySelector`:

```
foreach (Planet planet in
    firstFivePlanetsFromTheSun.UnionBy(
        lastFivePlanetsFromTheSun, planet => planet))
{
    Console.WriteLine(planet);
}

// This code produces the following output:
//     Planet { Name = Mercury, Type = Rock, OrderFromSun = 1 }
//     Planet { Name = Venus, Type = Rock, OrderFromSun = 2 }
//     Planet { Name = Earth, Type = Rock, OrderFromSun = 3 }
//     Planet { Name = Mars, Type = Rock, OrderFromSun = 4 }
//     Planet { Name = Jupiter, Type = Gas, OrderFromSun = 5 }
//     Planet { Name = Saturn, Type = Gas, OrderFromSun = 6 }
//     Planet { Name = Uranus, Type = Liquid, OrderFromSun = 7 }
//     Planet { Name = Neptune, Type = Liquid, OrderFromSun = 8 }
```

В приведенном выше коде C#:

- Эти два массива `Planet` объединяются с использованием семантики сравнения значений `record`.
- Полученный экземпляр `planet` записывается в консоль.

См. также

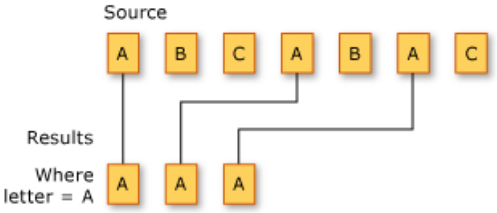
- System.Linq
- Общие сведения о стандартных операторах запроса (C#)
- Практическое руководство. Объединение и сравнение коллекций строк (LINQ) (C#)
- Нахождение разности множеств между двумя списками (LINQ) (C#)

Фильтрация данных (C#)

18.12.2021 • 2 minutes to read

Фильтрация — это операция по ограничению значений в результирующем наборе только элементами, соответствующими указанному условию. Это также называется выборкой.

На следующем рисунке показаны результаты операции фильтрации последовательности символов. Предикат для операции фильтрации указывает, что символ должен быть "A".



Методы стандартных операторов запросов, которые выполняют выборку, перечислены в следующем разделе.

Методы

ИМЯ МЕТОДА	ОПИСАНИЕ	СИНТАКСИС ВЫРАЖЕНИЯ ЗАПРОСА C#	ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ
OfType	Выбирает значения в зависимости от возможности приведения их к указанному типу.	Не применяется	Enumerable.OfType Queryable.OfType
Where	Выбирает значения, основанные на функции предиката.	`where`	Enumerable.Where Queryable.Where

Пример синтаксиса выражения запроса

В следующем примере для выбора из массива строк, имеющих определенную длину, используется предложение `where`.

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                            where word.Length == 3
                            select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    the
    fox
*/
```

См. также

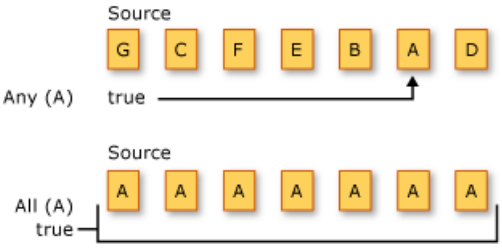
- System.Linq
- Общие сведения о стандартных операторах запроса (C#)
- предложение where
- Динамическое определение фильтров предикатов во время выполнения
- Практическое руководство. Выполнение запроса к метаданным сборки при помощи отражения (LINQ) (C#)
- Практическое руководство. Запрос файлов с указанными атрибутами или именем (C#)
- Практическое руководство. Сортировка или фильтрация текстовых данных по любому слову или полю (LINQ) (C#)

Операции, использующие квантификаторы (C#)

18.12.2021 • 2 minutes to read

Квантификаторы возвращают значение Boolean, которое указывает, удовлетворяют ли условию некоторые или все элементы в последовательности.

На приведенном ниже рисунке показаны два различных квантификатора, примененные к двум разным исходным последовательностям. Первая операция проверяет, является ли один или несколько элементов буквой "A"; результатом является `true`. Вторая операция проверяет, являются ли все элементы буквой "A"; результатом является `true`.



В следующем разделе перечислены методы стандартных операторов запросов, которые выполняют операции с использованием квантификаторов.

Методы

ИМЯ МЕТОДА	ОПИСАНИЕ	СИНТАКСИС ВЫРАЖЕНИЯ ЗАПРОСА C#	ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ
Все	Определяет, все ли элементы последовательности удовлетворяют условию.	Не применяется	Enumerable.All Queryable.All
Любой	Определяет, удовлетворяют ли условию какие-либо элементы последовательности.	Не применяется	Enumerable.Any Queryable.Any
Содержит	Определяет, содержит ли последовательность указанный элемент.	Не применяется	Enumerable.Contains Queryable.Contains

Примеры синтаксиса выражений запросов

Все

В следующем примере используется `All` для проверки того, что все строки имеют определенную длину.

```
class Market
{
    public string Name { get; set; }
    public string[] Items { get; set; }
}

public static void Example()
{
    List<Market> markets = new List<Market>
    {
        new Market { Name = "Emily's", Items = new string[] { "kiwi", "cheery", "banana" } },
        new Market { Name = "Kim's", Items = new string[] { "melon", "mango", "olive" } },
        new Market { Name = "Adam's", Items = new string[] { "kiwi", "apple", "orange" } },
    };

    // Determine which market have all fruit names length equal to 5
    IEnumerable<string> names = from market in markets
                                where market.Items.All(item => item.Length == 5)
                                select market.Name;

    foreach (string name in names)
    {
        Console.WriteLine($"{name} market");
    }

    // This code produces the following output:
    //
    // Kim's market
}
```

Любой

В следующем примере используется `Any` для проверки того, что все строки начинаются o.

```
class Market
{
    public string Name { get; set; }
    public string[] Items { get; set; }
}

public static void Example()
{
    List<Market> markets = new List<Market>
    {
        new Market { Name = "Emily's", Items = new string[] { "kiwi", "cheery", "banana" } },
        new Market { Name = "Kim's", Items = new string[] { "melon", "mango", "olive" } },
        new Market { Name = "Adam's", Items = new string[] { "kiwi", "apple", "orange" } },
    };

    // Determine which market have any fruit names start with 'o'
    IEnumerable<string> names = from market in markets
                                where market.Items.Any(item => item.StartsWith("o"))
                                select market.Name;

    foreach (string name in names)
    {
        Console.WriteLine($"{name} market");
    }

    // This code produces the following output:
    //
    // Kim's market
    // Adam's market
}
```

Содержит

В следующем примере используется `Contains` для проверки того, что массив имеет указанный элемент.

```
class Market
{
    public string Name { get; set; }
    public string[] Items { get; set; }
}

public static void Example()
{
    List<Market> markets = new List<Market>
    {
        new Market { Name = "Emily's", Items = new string[] { "kiwi", "cheery", "banana" } },
        new Market { Name = "Kim's", Items = new string[] { "melon", "mango", "olive" } },
        new Market { Name = "Adam's", Items = new string[] { "kiwi", "apple", "orange" } },
    };

    // Determine which market contains fruit names equal 'kiwi'
    IEnumerable<string> names = from market in markets
                                where market.Items.Contains("kiwi")
                                select market.Name;

    foreach (string name in names)
    {
        Console.WriteLine($"{name} market");
    }

    // This code produces the following output:
    //
    // Emily's market
    // Adam's market
}
```

См. также

- System.Linq
- Общие сведения о стандартных операторах запроса (C#)
- Динамическое определение фильтров предикатов во время выполнения
- Практическое руководство. Запрос к предложениям, содержащим указанный набор слов (LINQ) (C#)

Операции проекции (C#)

18.12.2021 • 5 minutes to read

Проекцией называют операцию преобразования объекта в новую форму, которая часто состоит только из тех его свойств, которые будут использоваться впоследствии. С помощью проекции можно создать новый тип, построенный из каждого объекта. Вы можете проецировать свойство и выполнять над ним математические функции. Также можно проецировать исходный объект, не изменяя его.

Методы стандартных операторов запросов, которые выполняют проецирование, перечислены в следующем разделе.

Методы

ИМЕНА МЕТОДОВ	ОПИСАНИЕ	СИНТАКСИС ВЫРАЖЕНИЯ ЗАПРОСА C#	ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ
Выбрать	Проецирует значения, основанные на функции преобразования.	`select`	Enumerable.Select Queryable.Select
SelectMany	Проецирует последовательности значений, основанных на функции преобразования, а затем выравнивает их в одну последовательность.	Использование нескольких предложений `from`	Enumerable.SelectMany Queryable.SelectMany
Почтовый индекс	Создает последовательность кортежей с элементами из указанных последовательностей и необязательного селектора результатов.	Неприменимо.	Enumerable.Zip Queryable.Zip

`Select`

В приведенном ниже примере предложение `select` используется для проецирования первой буквы из каждой строки в списке строк.

```
List<string> words = new() { "an", "apple", "a", "day" };

var query = from word in words
            select word.Substring(0, 1);

foreach (string s in query)
    Console.WriteLine(s);

/* This code produces the following output:

    a
    a
    a
    d
*/
```

`SelectMany`

В приведенном ниже примере несколько предложений `from` используются для проецирования каждого слова из каждой строки в списке строк.

```
List<string> phrases = new() { "an apple a day", "the quick brown fox" };

var query = from phrase in phrases
            from word in phrase.Split(' ')
            select word;

foreach (string s in query)
    Console.WriteLine(s);

/* This code produces the following output:

    an
    apple
    a
    day
    the
    quick
    brown
    fox
*/
```

`Zip`

Существует несколько перегрузок для оператора проекции `Zip`. Все методы `Zip` работают для последовательностей с двумя и более разнородными типами. Первые две перегрузки возвращают кортежи и соответствующий позиционный тип из заданных последовательностей.

Давайте рассмотрим следующие коллекции:

```
// An int array with 7 elements.
IEnumerable<int> numbers = new[]
{
    1, 2, 3, 4, 5, 6, 7
};
// A char array with 6 elements.
IEnumerable<char> letters = new[]
{
    'A', 'B', 'C', 'D', 'E', 'F'
};
```

Чтобы проецировать эти последовательности вместе, используйте оператор Enumerable.Zip<TFirst,TSecond>(IEnumerable<TFirst>, IEnumerable<TSecond>):

```
foreach ((int number, char letter) in numbers.Zip(letters))
{
    Console.WriteLine($"Number: {number} zipped with letter: '{letter}'");
}
// This code produces the following output:
//     Number: 1 zipped with letter: 'A'
//     Number: 2 zipped with letter: 'B'
//     Number: 3 zipped with letter: 'C'
//     Number: 4 zipped with letter: 'D'
//     Number: 5 zipped with letter: 'E'
//     Number: 6 zipped with letter: 'F'
```

IMPORTANT

Последовательность, полученная в результате операции объединения, никогда не будет длиннее кратчайшей последовательности. Иными словами, если взглянуть на коллекции `numbers` и `letters`, они отличаются по длине — результирующая последовательность опускает последний элемент из коллекции `letters`, так как ее не с чем объединять.

Вторая перегрузка принимает последовательность `third`. Давайте создадим еще одну коллекцию, а именно `emoji`:

```
// A string array with 8 elements.
IEnumerable<string> emoji = new[]
{
    "□", "□", "□", "□", "□", "□", "✔", "□"
};
```

Чтобы объединить эти последовательности с помощью проекции, примените оператор Enumerable.Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>):

```
foreach ((int number, char letter, string em) in numbers.Zip(letters, emoji))
{
    Console.WriteLine(
        $"Number: {number} is zipped with letter: '{letter}' and emoji: {em}");
}
// This code produces the following output:
//     Number: 1 is zipped with letter: 'A' and emoji: □
//     Number: 2 is zipped with letter: 'B' and emoji: □
//     Number: 3 is zipped with letter: 'C' and emoji: □
//     Number: 4 is zipped with letter: 'D' and emoji: □
//     Number: 5 is zipped with letter: 'E' and emoji: □
//     Number: 6 is zipped with letter: 'F' and emoji: □
```

Как и предыдущая перегрузка, метод `Zip` проецирует кортеж, но на этот раз с тремя элементами. И наконец, последняя перегрузка предоставляет объект `Func<TFirst, TSecond, TResult>`, который выступает в качестве селектора результатов. На основе двух типов из объединяемых последовательностей можно проецировать новую результирующую последовательность.

```
foreach (string result in
    numbers.Zip(letters, (number, letter) => $"{number} = {letter} ({(int)letter})"))
{
    Console.WriteLine(result);
}
// This code produces the following output:
//     1 = A (65)
//     2 = B (66)
//     3 = C (67)
//     4 = D (68)
//     5 = E (69)
//     6 = F (70)
```

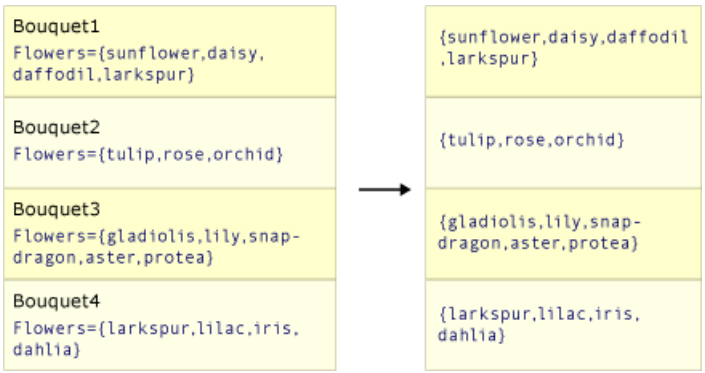
В предыдущей перегрузке `Zip` указанная функция применяется к соответствующим элементам `numbers` и `letter`, возвращая последовательность результатов `string`.

`Select` и `SelectMany`

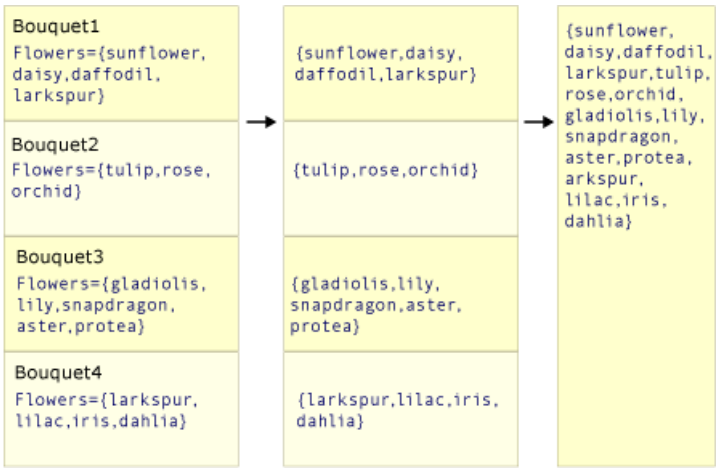
Задача обоих методов `Select` и `SelectMany` заключается в создании результирующего значения (или значений) из исходных значений. `Select` создает один результат для каждого исходного значения. Таким образом, общий результат является коллекцией, имеющей то же количество элементов, что и исходная коллекция. `SelectMany`, напротив, создает общий результат, содержащий соединенные подколлекции из каждого исходного значения. Функция преобразования, которая передается в качестве аргумента методу `SelectMany`, должна возвращать перечисляемую последовательность значений для каждого исходного значения. Эти перечисляемые последовательности затем объединяются `SelectMany` для создания одной большой последовательности.

На двух рисунках, приведенных ниже, показаны принципиальные различия между работой этих двух методов. В обоих случаях предполагается, что функция выбора (преобразования) выбирает массив цветов из каждого исходного значения.

На этом рисунке представлено, как `Select` возвращает коллекцию, которая имеет то же количество элементов, что и исходная коллекция.



На этом рисунке показано, как `SelectMany` объединяет промежуточные последовательности массивов в один конечный результат, содержащий все значения из промежуточных массивов.



Пример кода

В приведенном ниже примере сравнивается действие `Select` и `SelectMany`. Код создает "букет" из цветов путем получения первых двух элементов из каждого списка названий цветов в исходной коллекции. В этом примере отдельное значение, используемое функцией преобразования Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>), представляет собой коллекцию значений. Этот требует дополнительного цикла `foreach` для перечисления каждой строки в каждой подпоследовательности.

```
class Bouquet
{
    public List<string> Flowers { get; set; }
}

static void SelectVsSelectMany()
{
    List<Bouquet> bouquets = new()
    {
        new Bouquet { Flowers = new List<string> { "sunflower", "daisy", "daffodil", "larkspur" }},
        new Bouquet { Flowers = new List<string> { "tulip", "rose", "orchid" }},
        new Bouquet { Flowers = new List<string> { "gladiolis", "lily", "snapdragon", "aster", "protea" }},
        new Bouquet { Flowers = new List<string> { "larkspur", "lilac", "iris", "dahlia" }}
    };

    IEnumerable<List<string>> query1 = bouquets.Select(bq => bq.Flowers);

    IEnumerable<string> query2 = bouquets.SelectMany(bq => bq.Flowers);

    Console.WriteLine("Results by using Select():");
    // Note the extra foreach loop here.
    foreach (IEnumerable<String> collection in query1)
        foreach (string item in collection)
            Console.WriteLine(item);

    Console.WriteLine("\nResults by using SelectMany():");
    foreach (string item in query2)
        Console.WriteLine(item);

    /* This code produces the following output:

       Results by using Select():
        sunflower
        daisy
        daffodil
        larkspur
        tulip
        rose
        orchid
        gladiolis
        lily
        snapdragon
        aster
```

```
        aster
        protea
        larkspur
        lilac
        iris
        dahlia

      Results by using SelectMany():
        sunflower
        daisy
        daffodil
        larkspur
        tulip
        rose
        orchid
        gladiolis
        lily
        snapdragon
        aster
        protea
        larkspur
        lilac
        iris
        dahlia
    */
}
```

См. также

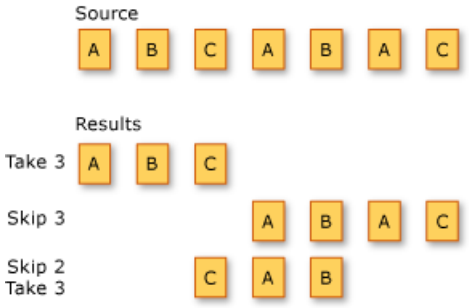
- System.Linq
- Общие сведения о стандартных операторах запроса (C#)
- предложение select
- Заполнение коллекций объектов из нескольких источников (LINQ) (C#)
- Разделение файла на несколько файлов с помощью групп (LINQ) (C#)

Секционирование данных (C#)

18.12.2021 • 2 minutes to read

Секционированием в LINQ называют операцию разделения входной последовательности на два раздела без изменения порядка элементов, а затем возвращения одного из разделов.

На следующем рисунке показаны результаты трех различных операций секционирования в последовательности символов. Первая операция возвращает первые три элемента в последовательности. Вторая операция пропускает первые три элемента и возвращает остальные элементы. Третья операция пропускает первые два элемента в последовательности и возвращает три следующих элемента.



Далее перечислены методы стандартных операторов запроса, которые секционируют последовательности.

Операторы

ИМЕНА МЕТОДОВ	ОПИСАНИЕ	СИНТАКСИС ВЫРАЖЕНИЯ ЗАПРОСА C#	ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ
Skip	Пропускает элементы до указанной позиции в последовательности.	Не применяется	Enumerable.Skip Queryable.Skip
SkipWhile	Пропускает элементы, пока элемент не удовлетворит условию функции предиката.	Не применяется	Enumerable.SkipWhile Queryable.SkipWhile
Take	Возвращает элементы на указанную позицию в последовательности.	Не применяется	Enumerable.Take Queryable.Take
TakeWhile	Принимает элементы, пока элемент не удовлетворит условию функции предиката.	Не применяется	Enumerable.TakeWhile Queryable.TakeWhile
Chunk	Разделите элементы последовательности на блоки максимально допустимого размера.	Неприменимо.	Enumerable.Chunk Queryable.Chunk

Пример

Оператор `Chunk` используется для разделения элементов последовательности с учетом заданного объекта `size`.

```
int chunkNumber = 1;
foreach (int[] chunk in Enumerable.Range(0, 8).Chunk(3))
{
    Console.WriteLine($"Chunk {chunkNumber++}:");
    foreach (int item in chunk)
    {
        Console.WriteLine($"    {item}");
    }

    Console.WriteLine();
}
// This code produces the following output:
// Chunk 1:
//    0
//    1
//    2
//
//Chunk 2:
//    3
//    4
//    5
//
//Chunk 3:
//    6
//    7
```

В приведенном выше коде C#:

- Использует Enumerable.Range(Int32, Int32) для создания последовательности чисел.
- Применяет оператор `Chunk`, разделяя последовательность на блоки, которые содержат максимум три элемента.

См. также

- System.Linq
- Общие сведения о стандартных операторах запроса (C#)

Операции Join (C#)

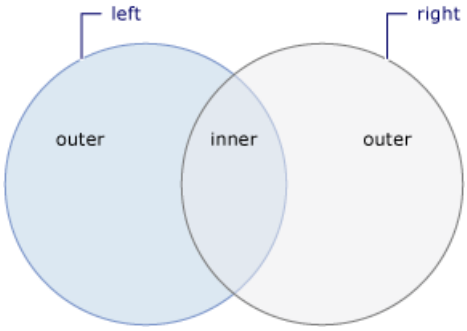
18.12.2021 • 4 minutes to read

Соединение двух источников данных — это связь объектов в одном источнике данных с объектами, которые имеют общий атрибут в другом источнике данных.

Join является важной операцией в запросах, направленных на источники данных, отношения которых друг к другу нельзя отследить напрямую. В объектно-ориентированном программировании оно может означать корреляцию между немоделируемыми объектами, например такими, как обратное направление одностороннего отношения. Примером одностороннего отношения является класс Customer, имеющий свойство типа City (город), в то время как класс City не имеет свойства, которое является коллекцией объектов Customer (клиент). В случае наличия списка объектов City для поиска всех клиентов в каждом городе можно использовать операцию соединения.

На платформе LINQ представлены методы объединения Join и GroupJoin. Они выполняют эквисоединения, или соединения, которые сопоставляют два источника данных на основе равенства их ключей. (Для сравнения, Transact-SQL поддерживает операции соединения, отличные от оператора "равно", например оператор "меньше, чем".) В терминах реляционных баз данных Join реализует внутреннее соединение — тип соединения, в котором возвращаются только те объекты, у которых есть совпадения в другом наборе данных. Метод GroupJoin не имеет прямого эквивалента в терминах реляционных баз данных, но реализует надмножество внутренних соединений и левых внешних соединений. Левое внешнее соединение — это соединение, которое возвращает каждый элемент первого (левого) источника данных, даже если в другом источнике данных не имеется соответствующих элементов.

На следующем рисунке показано концептуальное представление из двух наборов и элементов, входящих в эти наборы, которые включены либо во внутреннее соединение, либо в левое внешнее соединение.



Методы

ИМЯ МЕТОДА	ОПИСАНИЕ	СИНТАКСИС ВЫРАЖЕНИЯ ЗАПРОСА C#	ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ
Join	Join две последовательности на основании функций селектора ключа и извлекает пары значений.	`join … in … on … equals …`	Enumerable.Join Queryable.Join

ИМЯ МЕТОДА	ОПИСАНИЕ	СИНТАКСИС ВЫРАЖЕНИЯ ЗАПРОСА C#	ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ
GroupJoin	Join две последовательности на основании функций селектора ключа и группирует полученные при сопоставлении данные для каждого элемента.	`join … in … on … equals … into …`	Enumerable.GroupJoin Queryable.GroupJoin

Примеры синтаксиса выражений запросов

Join

В следующем примере предложение `join … in … on … equals …` используется для объединения двух последовательностей на основе конкретного значения.

```
class Product
{
    public string Name { get; set; }
    public int CategoryId { get; set; }
}

class Category
{
    public int Id { get; set; }
    public string CategoryName { get; set; }
}

public static void Example()
{
    List<Product> products = new List<Product>
    {
        new Product { Name = "Cola", CategoryId = 0 },
        new Product { Name = "Tea", CategoryId = 0 },
        new Product { Name = "Apple", CategoryId = 1 },
        new Product { Name = "Kiwi", CategoryId = 1 },
        new Product { Name = "Carrot", CategoryId = 2 },
    };

    List<Category> categories = new List<Category>
    {
        new Category { Id = 0, CategoryName = "Beverage" },
        new Category { Id = 1, CategoryName = "Fruit" },
        new Category { Id = 2, CategoryName = "Vegetable" }
    };

    // Join products and categories based on CategoryId
    var query = from product in products
                join category in categories on product.CategoryId equals category.Id
                select new { product.Name, category.CategoryName };

    foreach (var item in query)
    {
        Console.WriteLine($"{item.Name} - {item.CategoryName}");
    }

    // This code produces the following output:
    //
    // Cola - Beverage
    // Tea - Beverage
    // Apple - Fruit
    // Kiwi - Fruit
    // Carrot - Vegetable
}
```

GroupJoin

В следующем примере используется предложение `join … in … on … equals … into …` для объединения двух последовательностей на основе конкретного значения, а полученные совпадения для каждого элемента группируются.

```
class Product
{
    public string Name { get; set; }
    public int CategoryId { get; set; }
}

class Category
{
    public int Id { get; set; }
    public string CategoryName { get; set; }
}

public static void Example()
{
    List<Product> products = new List<Product>
    {
        new Product { Name = "Cola", CategoryId = 0 },
        new Product { Name = "Tea", CategoryId = 0 },
        new Product { Name = "Apple", CategoryId = 1 },
        new Product { Name = "Kiwi", CategoryId = 1 },
        new Product { Name = "Carrot", CategoryId = 2 },
    };

    List<Category> categories = new List<Category>
    {
        new Category { Id = 0, CategoryName = "Beverage" },
        new Category { Id = 1, CategoryName = "Fruit" },
        new Category { Id = 2, CategoryName = "Vegetable" }
    };

    // Join categories and product based on CategoryId and grouping result
    var productGroups = from category in categories
                        join product in products on category.Id equals product.CategoryId into productGroup
                        select productGroup;

    foreach (IEnumerable<Product> productGroup in productGroups)
    {
        Console.WriteLine("Group");
        foreach (Product product in productGroup)
        {
            Console.WriteLine($"{product.Name,8}");
        }
    }

    // This code produces the following output:
    //
    // Group
    //     Cola
    //      Tea
    // Group
    //    Apple
    //     Kiwi
    // Group
    //   Carrot
}
```

См. также

- System.Linq
- Общие сведения о стандартных операторах запроса (C#)
- Анонимные типы
- Формулировка Join и запросов перекрестного произведения
- предложение join

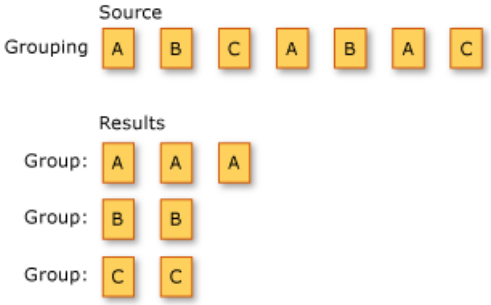
- Join с помощью составных ключей
- Объединение содержимого из файлов разных форматов (LINQ) (C#)
- Упорядочение результатов предложения соединения
- Выполнение пользовательских операций соединения
- Выполнение групповых соединений
- Выполнение внутренних соединений
- Выполнение левых внешних соединений
- Заполнение коллекций объектов из нескольких источников (LINQ) (C#)

Группирование данных (C#)

18.12.2021 • 2 minutes to read

Группированием называют операцию объединения данных в группы таким образом, чтобы у элементов в каждой группе был общий атрибут.

На следующем рисунке показаны результаты операции группирования последовательности символов. Ключ для каждой группы — это символ.



Далее перечислены методы стандартных операторов запроса, которые группируют элементы данных.

<a name="methods">Методы

ИМЯ МЕТОДА	ОПИСАНИЕ	СИНТАКСИС ВЫРАЖЕНИЯ ЗАПРОСА C#	ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ

GroupBy	Группирует элементы с общим атрибутом. Каждая группа представлена объектом IGrouping<TKey,TElement>.	`group … by`
		-или-
		`group … by … into …`
			Enumerable.GroupBy
			Queryable.GroupBy

ToLookup	Вставляет элементы в Lookup<TKey,TElement> (словарь "один ко многим") в зависимости от функции выбора ключа.	Не применяется	Enumerable.ToLookup

Пример синтаксиса выражения запроса

В следующем примере кода предложение `group by` используется для группирования целых чисел в списке на основании четности.

```
List<int> numbers = new List<int>() { 35, 44, 200, 84, 3987, 4, 199, 329, 446, 208 };

IEnumerable<IGrouping<int, int>> query = from number in numbers
                                         group number by number % 2;

foreach (var group in query)
{
    Console.WriteLine(group.Key == 0 ? "\nEven numbers:" : "\nOdd numbers:");
    foreach (int i in group)
        Console.WriteLine(i);
}

/* This code produces the following output:

    Odd numbers:
    35
    3987
    199
    329

    Even numbers:
    44
    200
    84
    4
    446
    208
*/
```

См. также

- System.Linq
- Общие сведения о стандартных операторах запроса (C#)
- предложение group
- Создание вложенной группы
- Группировка файлов по расширению (LINQ) (C#)
- Группировка результатов запросов
- Вложенный запрос в операции группирования
- Разделение файла на несколько файлов с помощью групп (LINQ) (C#)

Операции создания (C#)

18.12.2021 • 2 minutes to read

Под созданием понимается формирование новых последовательностей значений.

В следующем разделе перечислены методы стандартных операторов запроса, которые выполняют создание.

Методы

ИМЯ МЕТОДА	ОПИСАНИЕ	СИНТАКСИС ВЫРАЖЕНИЯ ЗАПРОСА C#	ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ
DefaultIfEmpty	Заменяет пустую коллекцию на одноэлементную коллекцию со значениями по умолчанию.	Не применяется	Enumerable.DefaultIfEmpty Queryable.DefaultIfEmpty
Empty	Возвращает пустую коллекцию.	Не применяется	Enumerable.Empty
Диапазон	Создает коллекцию, содержащую последовательность чисел.	Не применяется	Enumerable.Range
Повторить	Создает коллекцию, содержащую одно повторяющееся значение.	Не применяется	Enumerable.Repeat

См. также

- System.Linq
- Общие сведения о стандартных операторах запроса (C#)

Операции равенства (C#)

18.12.2021 • 2 minutes to read

Две последовательности, соответствующие элементы в которых совпадают и которые имеют одинаковое число элементов, считаются равными.

Методы

ИМЯ МЕТОДА	ОПИСАНИЕ	СИНТАКСИС ВЫРАЖЕНИЯ ЗАПРОСА C#	ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ
SequenceEqual	Определяет, равны ли две последовательности, сравнивая элементы попарно.	Не применяется	Enumerable.SequenceEqual Queryable.SequenceEqual

См. также

- System.Linq
- Общие сведения о стандартных операторах запроса (C#)
- Практическое руководство. Сравнение содержимого двух папок (LINQ) (C#)

Операции с элементами (C#)

18.12.2021 • 2 minutes to read

Операции с элементами возвращают один определенный элемент из последовательности.

В следующем разделе перечислены методы стандартных операторов запросов, которые выполняют операции с элементами.

Методы

ИМЯ МЕТОДА	ОПИСАНИЕ	СИНТАКСИС ВЫРАЖЕНИЯ ЗАПРОСА C#	ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ
ElementAt	Возвращает элемент коллекции с указанным индексом.	Не применяется	Enumerable.ElementAt Queryable.ElementAt
ElementAtOrDefault	Возвращает элемент коллекции с указанным индексом или значение по умолчанию, если индекс выходит за пределы допустимого диапазона.	Не применяется	Enumerable.ElementAtOrDefault Queryable.ElementAtOrDefault
First	Возвращает первый элемент коллекции или первый элемент, удовлетворяющий условию.	Не применяется	Enumerable.First Queryable.First
FirstOrDefault	Возвращает первый элемент коллекции или первый элемент, удовлетворяющий условию. Если такой элемент не существует, возвращает значение по умолчанию.	Не применяется	Enumerable.FirstOrDefault Queryable.FirstOrDefault Queryable.FirstOrDefault<TSource> (IQueryable<TSource>)
Последняя	Возвращает последний элемент коллекции или последний элемент, удовлетворяющий условию.	Не применяется	Enumerable.Last Queryable.Last
LastOrDefault	Возвращает последний элемент коллекции или последний элемент, удовлетворяющий условию. Если такой элемент не существует, возвращает значение по умолчанию.	Не применяется	Enumerable.LastOrDefault Queryable.LastOrDefault

ИМЯ МЕТОДА	ОПИСАНИЕ	СИНТАКСИС ВЫРАЖЕНИЯ ЗАПРОСА C#	ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ
Single	Возвращает единственный элемент коллекции или единственный элемент, удовлетворяющий условию. Создает исключение InvalidOperationException, если нет ни одного элемента для возврата или таких элементов несколько.	Не применяется	Enumerable.Single Queryable.Single
SingleOrDefault	Возвращает единственный элемент коллекции или единственный элемент, удовлетворяющий условию. Возвращает значение по умолчанию, если нет элементов для возврата. Создает исключение InvalidOperationException, если существует несколько элементов для возврата.	Не применяется	Enumerable.SingleOrDefault Queryable.SingleOrDefault

См. также

- System.Linq
- Общие сведения о стандартных операторах запроса (C#)
- Практическое руководство. Запрос самого большого файла или файлов в дереве папок (LINQ) (C#)

Преобразование типов данных (C#)

18.12.2021 • 2 minutes to read

Методы преобразования изменяют тип входных объектов.

Операции преобразования в запросах LINQ удобны в различных ситуациях. Ниже приводятся некоторые примеры.

- Метод Enumerable.AsEnumerable можно использовать, чтобы скрыть настраиваемую реализацию типа стандартного оператора запроса.
- Метод Enumerable.OfType позволяет использовать непараметризованные коллекции для запросов LINQ.
- Методы Enumerable.ToArray, Enumerable.ToDictionary, Enumerable.ToList и Enumerable.ToLookup можно использовать для принудительного немедленного выполнения запроса, не дожидаясь, пока этот запрос будет перечислен.

Методы

В следующей таблице перечислены методы стандартных операторов запросов, выполняющие преобразование типов данных.

Методы преобразования в этой таблице, имена которых начинаются с "As", изменяют статический тип исходной коллекции, но не выполняют перечисление. Методы, имена которых начинаются с "To", перечисляют исходную коллекцию и помещают элементы в соответствующий тип коллекции.

ИМЯ МЕТОДА	ОПИСАНИЕ	СИНТАКСИС ВЫРАЖЕНИЯ ЗАПРОСА C#	ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ
AsEnumerable	Возвращает входное значение, типизированное как IEnumerable<T>.	Не применяется	Enumerable.AsEnumerable
AsQueryable	Преобразует IEnumerable (универсальный шаблон) в IQueryable (универсальный шаблон).	Не применяется	Queryable.AsQueryable
Cast	Приводит элементы коллекции к указанному типу.	Используйте явно типизированную переменную диапазона. Пример: `from string str in words`	Enumerable.Cast Queryable.Cast
OfType	Фильтрует значения в зависимости от возможности их приведения к указанному типу.	Не применяется	Enumerable.OfType Queryable.OfType

ИМЯ МЕТОДА	ОПИСАНИЕ	СИНТАКСИС ВЫРАЖЕНИЯ ЗАПРОСА C#	ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ
ToArray	Преобразует коллекцию в массив. Этот метод принудительно выполняет запрос.	Не применяется	Enumerable.ToArray
ToDictionary	Помещает элементы в Dictionary<TKey,TValue> в зависимости от функции выбора ключа. Этот метод принудительно выполняет запрос.	Не применяется	Enumerable.ToDictionary
ToList	Преобразует коллекцию в List<T>. Этот метод принудительно выполняет запрос.	Не применяется	Enumerable.ToList
ToLookup	Помещает элементы в Lookup<TKey,TElement> (словарь "один ко многим") в зависимости от функции выбора ключа. Этот метод принудительно выполняет запрос.	Не применяется	Enumerable.ToLookup

Пример синтаксиса выражения запроса

В следующем примере кода явным образом типизированная переменная диапазона используется для приведения типа к подтипу перед обращением к элементу, доступному только в подтипе.

```
class Plant
{
    public string Name { get; set; }
}

class CarnivorousPlant : Plant
{
    public string TrapType { get; set; }
}

static void Cast()
{
    Plant[] plants = new Plant[] {
        new CarnivorousPlant { Name = "Venus Fly Trap", TrapType = "Snap Trap" },
        new CarnivorousPlant { Name = "Pitcher Plant", TrapType = "Pitfall Trap" },
        new CarnivorousPlant { Name = "Sundew", TrapType = "Flypaper Trap" },
        new CarnivorousPlant { Name = "Waterwheel Plant", TrapType = "Snap Trap" }
    };

    var query = from CarnivorousPlant cPlant in plants
                where cPlant.TrapType == "Snap Trap"
                select cPlant;

    foreach (Plant plant in query)
        Console.WriteLine(plant.Name);

    /* This code produces the following output:

        Venus Fly Trap
        Waterwheel Plant
    */
}
```

См. также

- System.Linq
- Общие сведения о стандартных операторах запроса (C#)
- предложение from
- Выражения запросов LINQ
- Выполнение запроса к ArrayList с помощью LINQ (C#)

Операции объединения (C#)

18.12.2021 • 2 minutes to read

Объединением называют операцию добавления одной последовательности к другой.

На рисунке ниже показана операция объединения двух последовательностей символов.

Далее перечислены методы стандартных операторов запроса, которые выполняют объединение.

Методы

ИМЯ МЕТОДА	ОПИСАНИЕ	СИНТАКСИС ВЫРАЖЕНИЯ ЗАПРОСА C#	ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ
Concat	Объединяет две последовательности в одну.	Не применяется	Enumerable.Concat Queryable.Concat

См. также

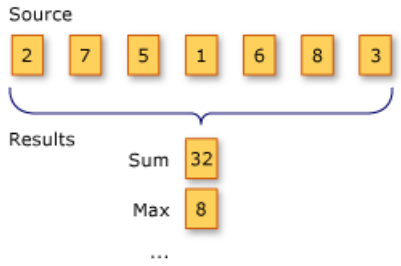
- System.Linq
- Общие сведения о стандартных операторах запроса (C#)
- Практическое руководство. Объединение и сравнение коллекций строк (LINQ) (C#)

Операции агрегирования (C#)

18.12.2021 • 2 minutes to read

Статистическая операция вычисляет одно значение по коллекции значений. Например, статистической обработкой является вычисление средней дневной температуры с использованием значений дневной температуры за месяц.

На приведенном ниже рисунке показаны результаты двух различных операций агрегирования с последовательностью чисел. Первая операция суммирует числа. Вторая операция возвращает максимальное значение в последовательности.



В следующем разделе перечислены методы стандартных операторов запросов, которые выполняют операции агрегирования.

Методы

ИМЯ МЕТОДА	DESCRIPTION	СИНТАКСИС ВЫРАЖЕНИЯ ЗАПРОСА C#	ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ
Статистическое выражение	Выполняет пользовательскую операцию агрегирования со значениями коллекции.	Не применяется	Enumerable.Aggregate Queryable.Aggregate
Метод Average	Вычисляет среднее значение коллекции значений.	Не применяется	Enumerable.Average Queryable.Average
Count	Подсчитывает число элементов в коллекции (при необходимости только те элементы, которые удовлетворяют функции предиката).	Не применяется	Enumerable.Count Queryable.Count
LongCount	Подсчитывает число элементов в большой коллекции (при необходимости только те элементы, которые удовлетворяют функции предиката).	Не применяется	Enumerable.LongCount Queryable.LongCount

ИМЯ МЕТОДА	DESCRIPTION	СИНТАКСИС ВЫРАЖЕНИЯ ЗАПРОСА C#	ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ
Max или MaxBy	Определяет максимальное значение в коллекции.	Не применяется	Enumerable.Max Enumerable.MaxBy Queryable.Max Queryable.MaxBy
Min или MinBy	Определяет минимальное значение в коллекции.	Не применяется	Enumerable.Min Enumerable.MinBy Queryable.Min Queryable.MinBy
Sum	Вычисляет сумму значений в коллекции.	Не применяется	Enumerable.Sum Queryable.Sum

См. также

- System.Linq
- Общие сведения о стандартных операторах запроса (C#)
- Практическое руководство. Вычисление значений столбцов в текстовом CSV-файле (LINQ) (C#)
- Практическое руководство. Запрос самого большого файла или файлов в дереве папок (LINQ) (C#)
- Получение общего числа байтов в наборе папок (LINQ) (C#)

LINQ to Objects (C#)

18.12.2021 • 2 minutes to read

Термин "LINQ to Objects" означает использование запросов LINQ с любой коллекцией IEnumerable или IEnumerable<T> напрямую, без привлечения промежуточного поставщика LINQ, API LINQ to SQL или LINQ to XML. Вы можете выполнить запрос LINQ к любой перечислимой коллекции, такой как List<T>, Array или Dictionary<TKey,TValue>. Коллекция может быть определена пользователем или возвращена API .NET.

В общем смысле LINQ to Objects представляет собой новый подход к коллекциям. Раньше нужно было написать сложные циклы `foreach`, определяющие порядок извлечения данных из коллекции. При использовании LINQ пишется декларативный код, описывающий, какие данные необходимо извлечь.

Кроме того, запросы LINQ предлагают три основных преимущества по сравнению с традиционными циклами `foreach`:

- Они более краткие и удобочитаемые, особенно при фильтрации нескольких условий.
- Они предоставляют широкие возможности фильтрации, упорядочивания и группировки с минимумом кода приложения.
- Они могут переноситься в другие источники данных практически без изменений.

В общем, чем сложнее операция, которую нужно выполнить с данными, тем больше преимуществ вы получаете при использовании LINQ вместо традиционных способов итерации.

Целью этого раздела является демонстрация подхода LINQ с помощью нескольких примеров. Он не претендует на исчерпывающий характер.

В этом разделе

LINQ и строки (C#)
Использование LINQ для запроса и преобразования строк и коллекций строк. Ссылки на статьи, демонстрирующие эти принципы.

LINQ и отражение (C#)
Ссылки на пример, демонстрирующий, как LINQ использует отражение.

LINQ и каталоги файлов (C#)
Использование LINQ для взаимодействия с файловыми системами. Ссылки на статьи, демонстрирующие эти понятия.

Практическое руководство. Выполнение запроса к ArrayList с помощью LINQ (C#)
Выполнение запроса ArrayList в C#.

Практическое руководство. Добавление настраиваемых методов для запросов LINQ (C#)
Расширение набора методов, которые можно использовать для запросов LINQ путем добавления методов расширения в интерфейс IEnumerable<T>.

LINQ (C#)
Ссылки на статьи, рассказывающие LINQ и содержащие примеры кода выполнения запросов.

LINQ и строки (C#)

18.12.2021 • 2 minutes to read

LINQ можно использовать для запроса и преобразования строк и коллекций строк. При этом лучше всего его потенциал раскрывается при работе с частично структурированными данными в текстовых файлах. Запросы LINQ можно комбинировать с традиционными строковыми функциями и регулярными выражениями. Например, используя метод String.Split или Regex.Split, можно создать массив строк, который затем можно запрашивать или изменять с помощью LINQ. Метод Regex.IsMatch можно использовать в предложении `where` запроса LINQ. Также LINQ можно использовать для запроса или изменения результатов MatchCollection, возвращаемых регулярным выражением.

Методы, описанные в этом разделе, позволяют преобразовать частично структурированные текстовые данные в XML. См. сведения в руководстве по созданию кода XML из CSV-файлов.

Примеры в этом разделе делятся на две категории:

Запрос блока текста

Вы можете запрашивать, анализировать и изменять блоки текста, разбивая их на запрашиваемый массив строк меньшего размера с помощью методов String.Split или Regex.Split. Исходный текст можно разбить на слова, предложения, абзацы, страницы или другие фрагменты, а затем применить другие способы фрагментации, необходимые для вашего запроса.

- Практическое руководство. Подсчет вхождений слова в строке (LINQ) (C#)
  Показывает, как использовать LINQ для простых запросов текста.
- Практическое руководство. Запрос к предложениям, содержащим указанный набор слов (LINQ) (C#)

  Показывает, как разбивать текстовые файлы на произвольные фрагменты и выполнять запросы к каждой части.
- Практическое руководство. Запрос символов в строке (LINQ) (C#)

  Показывает, что строка является запрашиваемым типом.
- Практическое руководство. Объединение запросов LINQ с помощью регулярных выражений (C#)

  Показывает, как выполнять сопоставление комплексных шаблонов с отфильтрованными результатами запросов, используя регулярные выражения в запросах LINQ.

Запрос частично структурированных данных в текстовом формате

Многие типы текстовых файлов состоят из серии строк, которые часто имеют одинаковый формат, например, из файлов с разделителями табуляцией или запятыми либо из строк фиксированной длины. После того как текстовый файл будет считан в память, можно использовать LINQ для запроса и (или) изменения строк. Кроме того, запросы LINQ упрощают задачу объединения данных из различных источников.

- Нахождение разности множеств между двумя списками (LINQ) (C#)

  Показывает, как найти все строки, которые есть в одном списке, но отсутствуют в другом.

- Практическое руководство. Сортировка или фильтрация текстовых данных по любому слову или полю (LINQ) (C#)

  Показывает, как сортировать текстовые строки по какому-либо слову или полю.

- Практическое руководство. Изменение порядка полей файла с разделителями (LINQ) (C#)

  Показывает, как изменить порядок полей в строке CSV-файла.

- Практическое руководство. Объединение и сравнение коллекций строк (LINQ) (C#)

  Показывает различные способы объединения списков строк.

- Заполнение коллекций объектов из нескольких источников (LINQ) (C#)

  Показывает, как создавать коллекции объектов, используя в качестве источников данных сразу несколько текстовых файлов.

- Объединение содержимого из файлов разных форматов (LINQ) (C#)

  Показывает, как объединить строки из двух списков в одну строку, используя ключ сопоставления.

- Разделение файла на несколько файлов с помощью групп (LINQ) (C#)

  Показывает, как создавать файлы, используя в качестве источника данных одиночный файл.

- Практическое руководство. Вычисление значений столбцов в текстовом CSV-файле (LINQ) (C#)

  Показывает, как выполнять математические расчеты на основе текстовых данных в CSV-файлах.

См. также

- LINQ (C#)
- Создание кода XML из CSV-файлов

Практическое руководство. Подсчет вхождений слова в строке (LINQ) (C#)

18.12.2021 • 2 minutes to read

В этом примере показано, как с помощью запроса LINQ определить, сколько раз то или иное слово встречается в строке. Обратите внимание на то, что для этого сначала вызывается метод Split, который создает массив слов. Использование метода Split связано с определенным снижением производительности. Если для строки выполняется только подсчет слов, рекомендуется вместо него использовать метод Matches или IndexOf. Если же производительность не критична или вы уже разбили предложение, чтобы выполнить с ним другие типы запросов, имеет смысл подсчитать слова или фразы с помощью LINQ.

Пример

```
class CountWords
{
    static void Main()
    {
        string text = @"Historically, the world of data and the world of objects" +
          @" have not been well integrated. Programmers work in C# or Visual Basic" +
          @" and also in SQL or XQuery. On the one side are concepts such as classes," +
          @" objects, fields, inheritance, and .NET APIs. On the other side" +
          @" are tables, columns, rows, nodes, and separate languages for dealing with" +
          @" them. Data types often require translation between the two worlds; there are" +
          @" different standard functions. Because the object world has no notion of query, a" +
          @" query can only be represented as a string without compile-time type checking or" +
          @" IntelliSense support in the IDE. Transferring data from SQL tables or XML trees to" +
          @" objects in memory is often tedious and error-prone.";

        string searchTerm = "data";

        //Convert the string into an array of words
        string[] source = text.Split(new char[] { '.', '?', '!', ' ', ';', ':', ',' },
StringSplitOptions.RemoveEmptyEntries);

        // Create the query.  Use ToLowerInvariant to match "data" and "Data"
        var matchQuery = from word in source
                         where word.ToLowerInvariant() == searchTerm.ToLowerInvariant()
                         select word;

        // Count the matches, which executes the query.
        int wordCount = matchQuery.Count();
        Console.WriteLine("{0} occurrences(s) of the search term \"{1}\" were found.", wordCount,
searchTerm);

        // Keep console window open in debug mode
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
   3 occurrences(s) of the search term "data" were found.
*/
```

Компиляция кода

Создайте проект консольного приложения C# с директивами `using` для пространств имен System.Linq и System.IO.

См. также

- LINQ и строки (C#)

Как запросить предложения с указанным набором слов (LINQ) (C#)

18.12.2021 • 2 minutes to read

В этом примере показан поиск предложений, содержащих совпадения для каждого из указанного набора слов в текстовом файле. Хотя массив терминов для поиска в этом примере жестко закодирован, его можно заполнять динамически во время выполнения. В этом примере запрос возвращает предложения, которые содержат слова "Historically", "data" и "integrated".

Пример

```
class FindSentences
{
    static void Main()
    {
        string text = @"Historically, the world of data and the world of objects " +
        @"have not been well integrated. Programmers work in C# or Visual Basic " +
        @"and also in SQL or XQuery. On the one side are concepts such as classes, " +
        @"objects, fields, inheritance, and .NET APIs. On the other side " +
        @"are tables, columns, rows, nodes, and separate languages for dealing with " +
        @"them. Data types often require translation between the two worlds; there are " +
        @"different standard functions. Because the object world has no notion of query, a " +
        @"query can only be represented as a string without compile-time type checking or " +
        @"IntelliSense support in the IDE. Transferring data from SQL tables or XML trees to " +
        @"objects in memory is often tedious and error-prone.";

        // Split the text block into an array of sentences.
        string[] sentences = text.Split(new char[] { '.', '?', '!' });

        // Define the search terms. This list could also be dynamically populated at run time.
        string[] wordsToMatch = { "Historically", "data", "integrated" };

        // Find sentences that contain all the terms in the wordsToMatch array.
        // Note that the number of terms to match is not specified at compile time.
        var sentenceQuery = from sentence in sentences
                            let w = sentence.Split(new char[] { '.', '?', '!', ' ', ';', ':', ',' },
                                                   StringSplitOptions.RemoveEmptyEntries)
                            where w.Distinct().Intersect(wordsToMatch).Count() == wordsToMatch.Count()
                            select sentence;

        // Execute the query. Note that you can explicitly type
        // the iteration variable here even though sentenceQuery
        // was implicitly typed.
        foreach (string str in sentenceQuery)
        {
            Console.WriteLine(str);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
Historically, the world of data and the world of objects have not been well integrated
*/
```

Запрос сначала разделяет текст на предложения, а затем разделяет предложения на массив строк,

содержащих слова. Для каждого из этих массивов метод Distinct удаляет все повторяющиеся слова, после чего выполняет операцию Intersect в отношении массива слов и массива `wordsToMatch`. Если число пересечений совпадает с размером массива `wordsToMatch`, все слова были найдены и возвращается исходное предложение.

В вызове Split знаки пунктуации используются как разделители для того, чтобы удалить их из строки. Если этого не сделать, то, например, можно получить строку "Historically," которая не будет совпадать с "Historically" в массиве `wordsToMatch`. В зависимости от знаков препинания в исходном тексте может потребоваться использовать дополнительные разделители.

Компиляция кода

Создайте проект консольного приложения C# с директивами `using` для пространств имен System.Linq и System.IO.

См. также

- LINQ и строки (C#)

Запрос символов в строке (LINQ) (C#)

18.12.2021 • 2 minutes to read

Поскольку класс String реализует универсальный интерфейс IEnumerable<T>, любая строка может запрашиваться как последовательность символов. Однако это не слишком распространенный пример использования LINQ. Для сложных операций сопоставления шаблонов используйте класс Regex.

Пример

В следующем примере запрашивается строка, чтобы определить количество содержащихся в ней цифр. Обратите внимание, что запрос "используется повторно" после первоначального выполнения. Это становится возможным, поскольку сам запрос не хранит фактические результаты.

```
class QueryAString
{
    static void Main()
    {
        string aString = "ABCDE99F-J74-12-89A";

        // Select only those characters that are numbers
        IEnumerable<char> stringQuery =
          from ch in aString
          where Char.IsDigit(ch)
          select ch;

        // Execute the query
        foreach (char c in stringQuery)
            Console.Write(c + " ");

        // Call the Count method on the existing query.
        int count = stringQuery.Count();
        Console.WriteLine("Count = {0}", count);

        // Select all characters before the first '-'
        IEnumerable<char> stringQuery2 = aString.TakeWhile(c => c != '-');

        // Execute the second query
        foreach (char c in stringQuery2)
            Console.Write(c);

        Console.WriteLine(System.Environment.NewLine + "Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
  Output: 9 9 7 4 1 2 8 9
  Count = 8
  ABCDE99F
*/
```

Компиляция кода

Создайте проект консольного приложения C# с директивами `using` для пространств имен System.Linq и System.IO.

См. также

- LINQ и строки (C#)
- Практическое руководство. Объединение запросов LINQ с помощью регулярных выражений (C#)

Практическое руководство. Объединение запросов LINQ с помощью регулярных выражений (C#)

18.12.2021 • 2 minutes to read

В этом примере показано, как использовать класс Regex при создании регулярного выражения для более сложных сопоставлений в текстовых строках. Запрос LINQ упрощает фильтрацию именно тех файлов, которые требуется найти с помощью регулярного выражения, и формирование результатов.

Пример

```
class QueryWithRegEx
{
    public static void Main()
    {
        // Modify this path as necessary so that it accesses your version of Visual Studio.
        string startFolder = @"C:\Program Files (x86)\Microsoft Visual Studio 14.0\";
        // One of the following paths may be more appropriate on your computer.
        //string startFolder = @"C:\Program Files (x86)\Microsoft Visual Studio\2017\";

        // Take a snapshot of the file system.
        IEnumerable<System.IO.FileInfo> fileList = GetFiles(startFolder);

        // Create the regular expression to find all things "Visual".
        System.Text.RegularExpressions.Regex searchTerm =
            new System.Text.RegularExpressions.Regex(@"Visual (Basic|C#|C\+\+|Studio)");

        // Search the contents of each .htm file.
        // Remove the where clause to find even more matchedValues!
        // This query produces a list of files where a match
        // was found, and a list of the matchedValues in that file.
        // Note: Explicit typing of "Match" in select clause.
        // This is required because MatchCollection is not a
        // generic IEnumerable collection.
        var queryMatchingFiles =
            from file in fileList
            where file.Extension == ".htm"
            let fileText = System.IO.File.ReadAllText(file.FullName)
            let matches = searchTerm.Matches(fileText)
            where matches.Count > 0
            select new
            {
                name = file.FullName,
                matchedValues = from System.Text.RegularExpressions.Match match in matches
                                select match.Value
            };

        // Execute the query.
        Console.WriteLine("The term \"{0}\" was found in:", searchTerm.ToString());

        foreach (var v in queryMatchingFiles)
        {
            // Trim the path a bit, then write
            // the file name in which a match was found.
            string s = v.name.Substring(startFolder.Length - 1);
            Console.WriteLine(s);

            // For this file, write out all the matching strings
```

```
            foreach (var v2 in v.matchedValues)
            {
                Console.WriteLine("   " + v2);
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // This method assumes that the application has discovery
    // permissions for all folders under the specified path.
    static IEnumerable<System.IO.FileInfo> GetFiles(string path)
    {
        if (!System.IO.Directory.Exists(path))
            throw new System.IO.DirectoryNotFoundException();

        string[] fileNames = null;
        List<System.IO.FileInfo> files = new List<System.IO.FileInfo>();

        fileNames = System.IO.Directory.GetFiles(path, "*.*", System.IO.SearchOption.AllDirectories);
        foreach (string name in fileNames)
        {
            files.Add(new System.IO.FileInfo(name));
        }
        return files;
    }
}
```

Обратите внимание, что можно также запросить объект MatchCollection, возвращаемый поиском `RegEx`. В этом примере в результатах создается только значение каждого совпадения. Тем не менее вы можете использовать LINQ для выполнения всех видов фильтрации, сортировки и группировки в этой коллекции. Так как MatchCollection является неуниверсальной коллекцией IEnumerable, необходимо явно указать тип переменной диапазона в запросе.

Компиляция кода

Создайте проект консольного приложения C# с директивами `using` для пространств имен System.Linq и System.IO.

См. также

- LINQ и строки (C#)
- LINQ и каталоги файлов (C#)

Нахождение разности множеств между двумя списками (LINQ) (C#)

18.12.2021 • 2 minutes to read

В этом примере показано, как использовать LINQ для сравнения двух списков строк и вывода тех строк, которые содержатся в файле names1.txt, но не в файле names2.txt.

Создание файлов данных

1. Скопируйте файлы names1.txt и names2.txt в папку решения, как показано в разделе Практическое руководство. Объединение и сравнение коллекций строк (LINQ) (C#).

Пример

```
class CompareLists
{
    static void Main()
    {
        // Create the IEnumerable data sources.
        string[] names1 = System.IO.File.ReadAllLines(@"../../../names1.txt");
        string[] names2 = System.IO.File.ReadAllLines(@"../../../names2.txt");

        // Create the query. Note that method syntax must be used here.
        IEnumerable<string> differenceQuery =
          names1.Except(names2);

        // Execute the query.
        Console.WriteLine("The following lines are in names1.txt but not names2.txt");
        foreach (string s in differenceQuery)
            Console.WriteLine(s);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
     The following lines are in names1.txt but not names2.txt
    Potra, Cristina
    Noriega, Fabricio
    Aw, Kam Foo
    Toyoshima, Tim
    Guy, Wey Yuan
    Garcia, Debra
     */
```

Некоторые типы операторов запроса в C#, например Except, Distinct, Union и Concat, могут выражаться только с использованием синтаксиса на основе методов.

Компиляция кода

Создайте проект консольного приложения C# с директивами `using` для пространств имен System.Linq и System.IO.

См. также

- LINQ и строки (C#)

Сортировка или фильтрация текстовых данных по любому слову или полю (LINQ)

18.12.2021 • 2 minutes to read

В следующем примере демонстрируется сортировка строк структурированного текста, таких как значения, разделенные запятыми, по любому полю в строке. Поле можно указывать в среде выполнения динамически. Допустим, поля в файле scores.csv содержат идентификационные номера учащихся и баллы, которые они набрали в результате четырех тестов.

Создание файла с данными

1. Скопируйте данные из файла scores.csv в папку решения. См. статью Практическое руководство по C#. Объединение содержимого из файлов разных форматов (LINQ).

Пример

```
public class SortLines
{
    static void Main()
    {
        // Create an IEnumerable data source
        string[] scores = System.IO.File.ReadAllLines(@"../../../scores.csv");

        // Change this to any value from 0 to 4.
        int sortField = 1;

        Console.WriteLine("Sorted highest to lowest by field [{0}]:", sortField);

        // Demonstrates how to return query from a method.
        // The query is executed here.
        foreach (string str in RunQuery(scores, sortField))
        {
            Console.WriteLine(str);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // Returns the query variable, not query results!
    static IEnumerable<string> RunQuery(IEnumerable<string> source, int num)
    {
        // Split the string and sort on field[num]
        var scoreQuery = from line in source
                         let fields = line.Split(',')
                         orderby fields[num] descending
                         select line;

        return scoreQuery;
    }
}
/* Output (if sortField == 1):
   Sorted highest to lowest by field [1]:
    116, 99, 86, 90, 94
    120, 99, 82, 81, 79
    111, 97, 92, 81, 60
    114, 97, 89, 85, 82
    121, 96, 85, 91, 60
    122, 94, 92, 91, 91
    117, 93, 92, 80, 87
    118, 92, 90, 83, 78
    113, 88, 94, 65, 91
    112, 75, 84, 91, 39
    119, 68, 79, 88, 92
    115, 35, 72, 91, 70
 */
```

Это пример показывает также, как вернуть переменную запроса из метода.

Компиляция кода

Создайте проект консольного приложения C# с директивами `using` для пространств имен System.Linq и System.IO.

См. также

- LINQ и строки (C#)

Как изменить порядок полей файла с разделителями (LINQ) (C#)

18.12.2021 • 2 minutes to read

CSV-файл — это текстовый файл, который часто используется для хранения данных электронных таблиц или других табличных данных, представленных строками и столбцами. Использование метода Split для разделения полей упрощает создание запросов к CSV-файлам и управление ими с помощью LINQ. Фактически та же технология может использоваться для изменения порядка частей любой структурированной строки текста, а не только CSV-файлов.

В следующем примере предполагается, что три столбца представляют "фамилию", "имя" и "идентификатор" учащихся. Поля группируются в алфавитном порядке по фамилии учащихся. Запрос создает новую последовательность, в которой столбец идентификатора отображается первым, за ним следует второй столбец, который объединяет имя и фамилию учащегося. Порядок строк изменен в соответствии с полем идентификатора. Результаты сохраняются в новый файл, и исходные данные не изменяются.

Создание файла данных

1. Скопируйте следующие строки в обычный текстовый файл с именем spreadsheet1.csv. Сохраните файл в папке проекта.

```
Adams,Terry,120
Fakhouri,Fadi,116
Feng,Hanying,117
Garcia,Cesar,114
Garcia,Debra,115
Garcia,Hugo,118
Mortensen,Sven,113
O'Donnell,Claire,112
Omelchenko,Svetlana,111
Tucker,Lance,119
Tucker,Michael,122
Zabokritski,Eugene,121
```

Пример

```
class CSVFiles
{
    static void Main(string[] args)
    {
        // Create the IEnumerable data source
        string[] lines = System.IO.File.ReadAllLines(@"../../../spreadsheet1.csv");

        // Create the query. Put field 2 first, then
        // reverse and combine fields 0 and 1 from the old field
        IEnumerable<string> query =
            from line in lines
            let x = line.Split(',')
            orderby x[2]
            select x[2] + ", " + (x[1] + " " + x[0]);

        // Execute the query and write out the new file. Note that WriteAllLines
        // takes a string[], so ToArray is called on the query.
        System.IO.File.WriteAllLines(@"../../../spreadsheet2.csv", query.ToArray());

        Console.WriteLine("Spreadsheet2.csv written to disk. Press any key to exit");
        Console.ReadKey();
    }
}
/* Output to spreadsheet2.csv:
111, Svetlana Omelchenko
112, Claire O'Donnell
113, Sven Mortensen
114, Cesar Garcia
115, Debra Garcia
116, Fadi Fakhouri
117, Hanying Feng
118, Hugo Garcia
119, Lance Tucker
120, Terry Adams
121, Eugene Zabokritski
122, Michael Tucker
 */
```

Компиляция кода

Создайте проект консольного приложения C# с директивами `using` для пространств имен System.Linq и System.IO.

См. также

- LINQ и строки (C#)
- LINQ и каталоги файлов (C#)
- Создание XML из CSV-файлов (C#)

Практическое руководство. Объединение и сравнение коллекций строк (LINQ) (C#)

18.12.2021 • 2 minutes to read

В этом примере показано объединение файлов, содержащих строки текста, и последующая сортировка результатов. В частности, показано выполнение простого сцепления, объединения и пересечения в двух наборах строк текста.

Настройка проекта и текстовых файлов

1. Скопируйте эти имена в текстовый файл с именем names1.txt и сохраните его в папке проекта:

```
Bankov, Peter
Holm, Michael
Garcia, Hugo
Potra, Cristina
Noriega, Fabricio
Aw, Kam Foo
Beebe, Ann
Toyoshima, Tim
Guy, Wey Yuan
Garcia, Debra
```

2. Скопируйте эти имена в текстовый файл с именем names2.txt и сохраните его в папке проекта. Обратите внимание, что в этих двух файлах имеются общие имена.

```
Liu, Jinghao
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
Gilchrist, Beth
Myrcha, Jacek
Giakoumakis, Leo
McLin, Nkenge
El Yassir, Mehdi
```

Пример

```
class MergeStrings
    {
        static void Main(string[] args)
        {
            //Put text files in your solution folder
            string[] fileA = System.IO.File.ReadAllLines(@"../../../names1.txt");
            string[] fileB = System.IO.File.ReadAllLines(@"../../../names2.txt");

            //Simple concatenation and sort. Duplicates are preserved.
            IEnumerable<string> concatQuery =
                fileA.Concat(fileB).OrderBy(s => s);

            // Pass the query variable to another function for execution.
            OutputQueryResults(concatQuery, "Simple concatenate and sort. Duplicates are preserved:");

            // Concatenate and remove duplicate names based on
            // default string comparer.
```

```
            // default string comparer.
            IEnumerable<string> uniqueNamesQuery =
                fileA.Union(fileB).OrderBy(s => s);
            OutputQueryResults(uniqueNamesQuery, "Union removes duplicate names:");

            // Find the names that occur in both files (based on
            // default string comparer).
            IEnumerable<string> commonNamesQuery =
                fileA.Intersect(fileB);
            OutputQueryResults(commonNamesQuery, "Merge based on intersect:");

            // Find the matching fields in each list. Merge the two
            // results by using Concat, and then
            // sort using the default string comparer.
            string nameMatch = "Garcia";

            IEnumerable<String> tempQuery1 =
                from name in fileA
                let n = name.Split(',')
                where n[0] == nameMatch
                select name;

            IEnumerable<string> tempQuery2 =
                from name2 in fileB
                let n2 = name2.Split(',')
                where n2[0] == nameMatch
                select name2;

            IEnumerable<string> nameMatchQuery =
                tempQuery1.Concat(tempQuery2).OrderBy(s => s);
            OutputQueryResults(nameMatchQuery, $"Concat based on partial name match \"{nameMatch}\":");

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to exit");
            Console.ReadKey();
        }

        static void OutputQueryResults(IEnumerable<string> query, string message)
        {
            Console.WriteLine(System.Environment.NewLine + message);
            foreach (string item in query)
            {
                Console.WriteLine(item);
            }
            Console.WriteLine("{0} total names in list", query.Count());
        }
    }
    /* Output:
        Simple concatenate and sort. Duplicates are preserved:
        Aw, Kam Foo
        Bankov, Peter
        Bankov, Peter
        Beebe, Ann
        Beebe, Ann
        El Yassir, Mehdi
        Garcia, Debra
        Garcia, Hugo
        Garcia, Hugo
        Giakoumakis, Leo
        Gilchrist, Beth
        Guy, Wey Yuan
        Holm, Michael
        Holm, Michael
        Liu, Jinghao
        McLin, Nkenge
        Myrcha, Jacek
        Noriega, Fabricio
        Potra, Cristina
        Toyoshima, Tim
        20 total names in list
```

```
        20 total names in list

        Union removes duplicate names:
        Aw, Kam Foo
        Bankov, Peter
        Beebe, Ann
        El Yassir, Mehdi
        Garcia, Debra
        Garcia, Hugo
        Giakoumakis, Leo
        Gilchrist, Beth
        Guy, Wey Yuan
        Holm, Michael
        Liu, Jinghao
        McLin, Nkenge
        Myrcha, Jacek
        Noriega, Fabricio
        Potra, Cristina
        Toyoshima, Tim
        16 total names in list

        Merge based on intersect:
        Bankov, Peter
        Holm, Michael
        Garcia, Hugo
        Beebe, Ann
        4 total names in list

        Concat based on partial name match "Garcia":
        Garcia, Debra
        Garcia, Hugo
        Garcia, Hugo
        3 total names in list
*/
```

Компиляция кода

Создайте проект консольного приложения C# с директивами `using` для пространств имен System.Linq и System.IO.

См. также

- LINQ и строки (C#)
- LINQ и каталоги файлов (C#)

Заполнение коллекций объектов из нескольких источников (LINQ) (C#)

18.12.2021 • 3 minutes to read

В этом примере показано, как объединить данные из разных источников в последовательность новых типов.

> **NOTE**
> Не пытайтесь объединить данные в памяти или данные в файловой системе с данными, которые остаются в базе данных. Такие междоменные соединения могут повлечь за собой непредвиденные результаты из-за разных способов, с помощью которых операции соединения могут определяться для запросов к базам данных и другим типам источников. Кроме того, существует риск, что такая операция может вызвать исключение нехватки памяти, если объем данных в базе данных достаточно большой. Чтобы объединить данные из базы данных с данными в памяти, сначала вызовите `ToList` или `ToArray` для запроса к базе данных, а затем выполните присоединение для возвращенной коллекции.

Создание файла данных

Скопируйте файлы names.csv и scores.csv в папку проекта, как описано в руководстве по объединению содержимого из файлов разных форматов (LINQ) (C#).

Пример

Следующий пример демонстрирует использование именованного типа `Student` для хранения объединенных данных из двух коллекций строк в памяти, которые имитируют данные электронной таблицы в формате CSV. Первая коллекция строк представляет имена и идентификаторы учащихся, а вторая коллекция — идентификатор учащегося (в первом столбце) и результаты четырех экзаменов. Идентификатор используется в качестве внешнего ключа.

```
using System;
using System.Collections.Generic;
using System.Linq;

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int ID { get; set; }
    public List<int> ExamScores { get; set; }
}

class PopulateCollection
{
    static void Main()
    {
        // These data files are defined in How to join content from
        // dissimilar files (LINQ).

        // Each line of names.csv consists of a last name, a first name, and an
        // ID number, separated by commas. For example, Omelchenko,Svetlana,111
        string[] names = System.IO.File.ReadAllLines(@"../../../names.csv");

        // Each line of scores.csv consists of an ID number and four test
```

```
        // Each line of scores.csv consists of an ID number and four test
        // scores, separated by commas. For example, 111, 97, 92, 81, 60
        string[] scores = System.IO.File.ReadAllLines(@"../../../scores.csv");

        // Merge the data sources using a named type.
        // var could be used instead of an explicit type. Note the dynamic
        // creation of a list of ints for the ExamScores member. The first item
        // is skipped in the split string because it is the student ID,
        // not an exam score.
        IEnumerable<Student> queryNamesScores =
            from nameLine in names
            let splitName = nameLine.Split(',')
            from scoreLine in scores
            let splitScoreLine = scoreLine.Split(',')
            where Convert.ToInt32(splitName[2]) == Convert.ToInt32(splitScoreLine[0])
            select new Student()
            {
                FirstName = splitName[0],
                LastName = splitName[1],
                ID = Convert.ToInt32(splitName[2]),
                ExamScores = (from scoreAsText in splitScoreLine.Skip(1)
                              select Convert.ToInt32(scoreAsText)).
                              ToList()
            };

        // Optional. Store the newly created student objects in memory
        // for faster access in future queries. This could be useful with
        // very large data files.
        List<Student> students = queryNamesScores.ToList();

        // Display each student's name and exam score average.
        foreach (var student in students)
        {
            Console.WriteLine("The average score of {0} {1} is {2}.",
                student.FirstName, student.LastName,
                student.ExamScores.Average());
        }

        //Keep console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    The average score of Omelchenko Svetlana is 82.5.
    The average score of O'Donnell Claire is 72.25.
    The average score of Mortensen Sven is 84.5.
    The average score of Garcia Cesar is 88.25.
    The average score of Garcia Debra is 67.
    The average score of Fakhouri Fadi is 92.25.
    The average score of Feng Hanying is 88.
    The average score of Garcia Hugo is 85.75.
    The average score of Tucker Lance is 81.75.
    The average score of Adams Terry is 85.25.
    The average score of Zabokritski Eugene is 83.
    The average score of Tucker Michael is 92.
 */
```

В предложении select инициализатор объектов используется для создания каждого нового объекта `Student` на основе данных из двух источников.

Если не требуется хранить результаты запроса, анонимные типы могут быть более удобными, чем именованные типы. Именованные типы необходимы, если результаты запроса передаются за пределы метода, в котором выполняется запрос. Следующий пример кода выполняет ту же задачу, что и в предыдущем примере, но использует анонимные типы вместо именованных типов:

```
// Merge the data sources by using an anonymous type.
// Note the dynamic creation of a list of ints for the
// ExamScores member. We skip 1 because the first string
// in the array is the student ID, not an exam score.
var queryNamesScores2 =
    from nameLine in names
    let splitName = nameLine.Split(',')
    from scoreLine in scores
    let splitScoreLine = scoreLine.Split(',')
    where Convert.ToInt32(splitName[2]) == Convert.ToInt32(splitScoreLine[0])
    select new
    {
        First = splitName[0],
        Last = splitName[1],
        ExamScores = (from scoreAsText in splitScoreLine.Skip(1)
                      select Convert.ToInt32(scoreAsText))
                      .ToList()
    };

// Display each student's name and exam score average.
foreach (var student in queryNamesScores2)
{
    Console.WriteLine("The average score of {0} {1} is {2}.",
        student.First, student.Last, student.ExamScores.Average());
}
```

См. также

- LINQ и строки (C#)
- Инициализаторы объектов и коллекций
- Анонимные типы

Разделение файла на несколько файлов с помощью групп (LINQ) (C#)

18.12.2021 • 2 minutes to read

В этом примере показан один из способов объединения содержимого двух файлов и последующего создания набора новых файлов, данные в котором будут организованы иначе.

Создание файлов данных

1. Скопируйте эти имена в текстовый файл с именем names1.txt и сохраните его в папке проекта:

```
Bankov, Peter
Holm, Michael
Garcia, Hugo
Potra, Cristina
Noriega, Fabricio
Aw, Kam Foo
Beebe, Ann
Toyoshima, Tim
Guy, Wey Yuan
Garcia, Debra
```

2. Скопируйте эти имена в текстовый файл с именем names2.txt и сохраните его в папке проекта. Обратите внимание на то, что имена этих двух файлов похожи.

```
Liu, Jinghao
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
Gilchrist, Beth
Myrcha, Jacek
Giakoumakis, Leo
McLin, Nkenge
El Yassir, Mehdi
```

Пример

```
class SplitWithGroups
{
    static void Main()
    {
        string[] fileA = System.IO.File.ReadAllLines(@"../../../names1.txt");
        string[] fileB = System.IO.File.ReadAllLines(@"../../../names2.txt");

        // Concatenate and remove duplicate names based on
        // default string comparer
        var mergeQuery = fileA.Union(fileB);

        // Group the names by the first letter in the last name.
        var groupQuery = from name in mergeQuery
                         let n = name.Split(',')
                         group name by n[0][0] into g
                         orderby g.Key
                         select g;
```

```
        // Create a new file for each group that was created
        // Note that nested foreach loops are required to access
        // individual items with each group.
        foreach (var g in groupQuery)
        {
            // Create the new file name.
            string fileName = @"../../../testFile_" + g.Key + ".txt";

            // Output to display.
            Console.WriteLine(g.Key);

            // Write file.
            using (System.IO.StreamWriter sw = new System.IO.StreamWriter(fileName))
            {
                foreach (var item in g)
                {
                    sw.WriteLine(item);
                    // Output to console for example purposes.
                    Console.WriteLine("   {0}", item);
                }
            }
        }
        // Keep console window open in debug mode.
        Console.WriteLine("Files have been written. Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
    A
       Aw, Kam Foo
    B
       Bankov, Peter
       Beebe, Ann
    E
       El Yassir, Mehdi
    G
       Garcia, Hugo
       Guy, Wey Yuan
       Garcia, Debra
       Gilchrist, Beth
       Giakoumakis, Leo
    H
       Holm, Michael
    L
       Liu, Jinghao
    M
       Myrcha, Jacek
       McLin, Nkenge
    N
       Noriega, Fabricio
    P
       Potra, Cristina
    T
       Toyoshima, Tim
 */
```

Программа записывает отдельный файл для каждой группы в ту же папку, где находятся файлы данных.

Компиляция кода

Создайте проект консольного приложения C# с директивами `using` для пространств имен System.Linq и System.IO.

См. также

- LINQ и строки (C#)
- LINQ и каталоги файлов (C#)

Объединение содержимого из файлов разных форматов (LINQ) (C#)

18.12.2021 • 2 minutes to read

В этом примере показано, как объединить данные из двух файлов с разделителями-запятыми, которые имеют общее значение, используемое в качестве совпадающего ключа. Этот способ может оказаться полезным, если необходимо объединить данные из двух электронных таблиц или из электронной таблицы и файла, имеющего другой формат, в новый файл. Можно изменить пример для обработки любого типа структурированного текста.

Создание файлов данных

1. Скопируйте следующие строки в файл с именем *scores.csv* и сохраните его в папке проекта. Этот файл представляет данные электронной таблицы. Первый столбец представляет идентификатор учащегося, а столбцы со второго по пятый представляют результаты тестирования.

```
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

2. Скопируйте следующие строки в файл с именем *names.csv* и сохраните его в папке проекта. Этот файл представляет электронную таблицу, содержащую фамилию, имя и идентификатор учащегося.

```
Omelchenko,Svetlana,111
O'Donnell,Claire,112
Mortensen,Sven,113
Garcia,Cesar,114
Garcia,Debra,115
Fakhouri,Fadi,116
Feng,Hanying,117
Garcia,Hugo,118
Tucker,Lance,119
Adams,Terry,120
Zabokritski,Eugene,121
Tucker,Michael,122
```

Пример

```
using System;
using System.Collections.Generic;
using System.Linq;

class JoinStrings
```

```
class JoinStrings
{
    static void Main()
    {
        // Join content from dissimilar files that contain
        // related information. File names.csv contains the student
        // name plus an ID number. File scores.csv contains the ID
        // and a set of four test scores. The following query joins
        // the scores to the student names by using ID as a
        // matching key.

        string[] names = System.IO.File.ReadAllLines(@"../../../names.csv");
        string[] scores = System.IO.File.ReadAllLines(@"../../../scores.csv");

        // Name:    Last[0],       First[1],  ID[2]
        //          Omelchenko,    Svetlana,  11
        // Score:   StudentID[0],  Exam1[1]   Exam2[2],  Exam3[3],  Exam4[4]
        //          111,           97,        92,        81,        60

        // This query joins two dissimilar spreadsheets based on common ID value.
        // Multiple from clauses are used instead of a join clause
        // in order to store results of id.Split.
        IEnumerable<string> scoreQuery1 =
            from name in names
            let nameFields = name.Split(',')
            from id in scores
            let scoreFields = id.Split(',')
            where Convert.ToInt32(nameFields[2]) == Convert.ToInt32(scoreFields[0])
            select nameFields[0] + "," + scoreFields[1] + "," + scoreFields[2]
                   + "," + scoreFields[3] + "," + scoreFields[4];

        // Pass a query variable to a method and execute it
        // in the method. The query itself is unchanged.
        OutputQueryResults(scoreQuery1, "Merge two spreadsheets:");

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    static void OutputQueryResults(IEnumerable<string> query, string message)
    {
        Console.WriteLine(System.Environment.NewLine + message);
        foreach (string item in query)
        {
            Console.WriteLine(item);
        }
        Console.WriteLine("{0} total names in list", query.Count());
    }
}
/* Output:
Merge two spreadsheets:
Omelchenko, 97, 92, 81, 60
O'Donnell, 75, 84, 91, 39
Mortensen, 88, 94, 65, 91
Garcia, 97, 89, 85, 82
Garcia, 35, 72, 91, 70
Fakhouri, 99, 86, 90, 94
Feng, 93, 92, 80, 87
Garcia, 92, 90, 83, 78
Tucker, 68, 79, 88, 92
Adams, 99, 82, 81, 79
Zabokritski, 96, 85, 91, 60
Tucker, 94, 92, 91, 91
12 total names in list
 */
```

См. также

- LINQ и строки (C#)
- LINQ и каталоги файлов (C#)

Практическое руководство. Вычисление значений столбцов в текстовом CSV-файле (LINQ) (C#)

18.12.2021 • 3 minutes to read

В этом примере демонстрируется выполнение статистических вычислений, таких как сумма, среднее, минимальное и максимальное для столбцов в CSV-файле. Приведенные в примере принципы могут применяться к другим типам структурированного текста.

Создание исходного файла

1. Скопируйте следующие строки в файл с именем scores.csv и сохраните его в папке проекта. Предположим, что первый столбец представляет идентификатор учащегося, а последующие столбцы — результаты четырех экзаменов.

```
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

Пример

```
class SumColumns
{
    static void Main(string[] args)
    {
        string[] lines = System.IO.File.ReadAllLines(@"../../../scores.csv");

        // Specifies the column to compute.
        int exam = 3;

        // Spreadsheet format:
        // Student ID    Exam#1  Exam#2  Exam#3  Exam#4
        // 111,          97,     92,     81,     60

        // Add one to exam to skip over the first column,
        // which holds the student ID.
        SingleColumn(lines, exam + 1);
        Console.WriteLine();
        MultiColumns(lines);

        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    static void SingleColumn(IEnumerable<string> strs, int examNum)
    {
        Console.WriteLine("Single Column Query:");
```

```
        // Parameter examNum specifies the column to
        // run the calculations on. This value could be
        // passed in dynamically at run time.

        // Variable columnQuery is an IEnumerable<int>.
        // The following query performs two steps:
        // 1) use Split to break each row (a string) into an array
        //    of strings,
        // 2) convert the element at position examNum to an int
        //    and select it.
        var columnQuery =
            from line in strs
            let elements = line.Split(',')
            select Convert.ToInt32(elements[examNum]);

        // Execute the query and cache the results to improve
        // performance. This is helpful only with very large files.
        var results = columnQuery.ToList();

        // Perform aggregate calculations Average, Max, and
        // Min on the column specified by examNum.
        double average = results.Average();
        int max = results.Max();
        int min = results.Min();

        Console.WriteLine("Exam #{0}: Average:{1:##.##} High Score:{2} Low Score:{3}",
                  examNum, average, max, min);
    }

    static void MultiColumns(IEnumerable<string> strs)
    {
        Console.WriteLine("Multi Column Query:");

        // Create a query, multiColQuery. Explicit typing is used
        // to make clear that, when executed, multiColQuery produces
        // nested sequences. However, you get the same results by
        // using 'var'.

        // The multiColQuery query performs the following steps:
        // 1) use Split to break each row (a string) into an array
        //    of strings,
        // 2) use Skip to skip the "Student ID" column, and store the
        //    rest of the row in scores.
        // 3) convert each score in the current row from a string to
        //    an int, and select that entire sequence as one row
        //    in the results.
        IEnumerable<IEnumerable<int>> multiColQuery =
            from line in strs
            let elements = line.Split(',')
            let scores = elements.Skip(1)
            select (from str in scores
                    select Convert.ToInt32(str));

        // Execute the query and cache the results to improve
        // performance.
        // ToArray could be used instead of ToList.
        var results = multiColQuery.ToList();

        // Find out how many columns you have in results.
        int columnCount = results[0].Count();

        // Perform aggregate calculations Average, Max, and
        // Min on each column.
        // Perform one iteration of the loop for each column
        // of scores.
        // You can use a for loop instead of a foreach loop
        // because you already executed the multiColQuery
        // query by calling ToList.
        for (int column = 0; column < columnCount; column++)
```

```
        for (int column = 0; column < columnCount; column++)
        {
            var results2 = from row in results
                           select row.ElementAt(column);
            double average = results2.Average();
            int max = results2.Max();
            int min = results2.Min();

            // Add one to column because the first exam is Exam #1,
            // not Exam #0.
            Console.WriteLine("Exam #{0} Average: {1:##.##} High Score: {2} Low Score: {3}",
                              column + 1, average, max, min);
        }
    }
}
/* Output:
    Single Column Query:
    Exam #4: Average:76.92 High Score:94 Low Score:39

    Multi Column Query:
    Exam #1 Average: 86.08 High Score: 99 Low Score: 35
    Exam #2 Average: 86.42 High Score: 94 Low Score: 72
    Exam #3 Average: 84.75 High Score: 91 Low Score: 65
    Exam #4 Average: 76.92 High Score: 94 Low Score: 39
 */
```

Запрос работает с использованием метода Split для преобразования каждой строки текста в массив. Каждый элемент массива представляет столбец. И наконец, текст в каждом столбце преобразуется в свое числовое представление. Если файл является файлом с разделением знаками табуляции, просто измените аргумент в методе `Split` на `\t`.

Компиляция кода

Создайте проект консольного приложения C# с директивами `using` для пространств имен System.Linq и System.IO.

См. также

- LINQ и строки (C#)
- LINQ и каталоги файлов (C#)

Выполнение запроса к метаданным сборки при помощи отражения (LINQ) (C#)

18.12.2021 • 2 minutes to read

API отражения .NET можно использовать для просмотра метаданных в сборке .NET и создания коллекций типов, элементов типов, параметров и т. д., присутствующих в этой сборке. Поскольку эти коллекции поддерживают универсальный интерфейс IEnumerable<T>, их можно запрашивать с помощью LINQ.

В следующем примере показано использование LINQ с отражением для извлечения определенных метаданных о методах, соответствующих условиям поиска. В этом примере запрос будет искать имена всех методов в сборке, которые возвращают перечислимые типы, такие как массивы.

Пример

```
using System;
using System.Linq;
using System.Reflection;

class ReflectionHowTO
{
    static void Main()
    {
        Assembly assembly = Assembly.Load("System.Core, Version=3.5.0.0, Culture=neutral, PublicKeyToken=
b77a5c561934e089");
        var pubTypesQuery = from type in assembly.GetTypes()
                    where type.IsPublic
                        from method in type.GetMethods()
                        where method.ReturnType.IsArray == true
                            || ( method.ReturnType.GetInterface(
                                typeof(System.Collections.Generic.IEnumerable<>).FullName ) != null
                            && method.ReturnType.FullName != "System.String" )
                        group method.ToString() by type.ToString();

        foreach (var groupOfMethods in pubTypesQuery)
        {
            Console.WriteLine("Type: {0}", groupOfMethods.Key);
            foreach (var method in groupOfMethods)
            {
                Console.WriteLine("  {0}", method);
            }
        }

        Console.WriteLine("Press any key to exit... ");
        Console.ReadKey();
    }
}
```

В примере используется метод Assembly.GetTypes для возвращения массива типов в указанной сборке. Чтобы вернуть только открытые типы, применяется фильтр where. Для каждого открытого типа создается вложенный запрос с использованием массива MethodInfo, который возвращается из вызова Type.GetMethods. Эти результаты фильтруются для возвращения только тех методов, возвращаемый тип которых является массивом или типом, который реализует IEnumerable<T>. Наконец, эти результаты группируются с помощью имени типа в качестве ключа.

См. также

- LINQ to Objects (C#)

Выполнение запроса к метаданным сборки при помощи отражения (LINQ) (C#)

18.12.2021 • 2 minutes to read

API отражения .NET можно использовать для просмотра метаданных в сборке .NET и создания коллекций типов, элементов типов, параметров и т. д., присутствующих в этой сборке. Поскольку эти коллекции поддерживают универсальный интерфейс IEnumerable<T>, их можно запрашивать с помощью LINQ.

В следующем примере показано использование LINQ с отражением для извлечения определенных метаданных о методах, соответствующих условиям поиска. В этом примере запрос будет искать имена всех методов в сборке, которые возвращают перечислимые типы, такие как массивы.

Пример

```
using System;
using System.Linq;
using System.Reflection;

class ReflectionHowTO
{
    static void Main()
    {
        Assembly assembly = Assembly.Load("System.Core, Version=3.5.0.0, Culture=neutral, PublicKeyToken=
b77a5c561934e089");
        var pubTypesQuery = from type in assembly.GetTypes()
                    where type.IsPublic
                        from method in type.GetMethods()
                        where method.ReturnType.IsArray == true
                            || ( method.ReturnType.GetInterface(
                                typeof(System.Collections.Generic.IEnumerable<>).FullName ) != null
                            && method.ReturnType.FullName != "System.String" )
                        group method.ToString() by type.ToString();

        foreach (var groupOfMethods in pubTypesQuery)
        {
            Console.WriteLine("Type: {0}", groupOfMethods.Key);
            foreach (var method in groupOfMethods)
            {
                Console.WriteLine("  {0}", method);
            }
        }

        Console.WriteLine("Press any key to exit... ");
        Console.ReadKey();
    }
}
```

В примере используется метод Assembly.GetTypes для возвращения массива типов в указанной сборке. Чтобы вернуть только открытые типы, применяется фильтр where. Для каждого открытого типа создается вложенный запрос с использованием массива MethodInfo, который возвращается из вызова Type.GetMethods. Эти результаты фильтруются для возвращения только тех методов, возвращаемый тип которых является массивом или типом, который реализует IEnumerable<T>. Наконец, эти результаты группируются с помощью имени типа в качестве ключа.

См. также

- LINQ to Objects (C#)

LINQ и каталоги файлов (C#)

18.12.2021 • 2 minutes to read

Многие операции файловой системы по существу являются запросами и, следовательно, соответствуют свойствам LINQ.

Запросы в этом разделе являются безопасными. Они не используются для изменения содержимого исходных файлов или папок. Это соответствует правилу о том, что запросы не должны вызывать никаких побочных эффектов. Как правило, любой код (включая запросы, выполняющие операторы создания, изменения и удаления), изменяющий исходные данные, должен храниться отдельно от кода, который просто запрашивает данные.

В этом разделе рассматриваются следующие вопросы.

Практическое руководство. Запрос файлов с указанными атрибутами или именем (C#)
Демонстрация поиска файлов путем проверки одного или нескольких свойств его объекта FileInfo.

Группировка файлов по расширению (LINQ) (C#)
Демонстрация возвращения групп объектов FileInfo в зависимости от расширения имени файла.

Получение общего числа байтов в наборе папок (LINQ) (C#)
Демонстрация возвращения общего числа байтов во всех файлах в указанном дереве каталогов.

Практическое руководство. Сравнение содержимого двух папок (LINQ) (C#)
Демонстрация возвращения всех файлов, присутствующих в двух указанных папках, а также всех файлов, которые присутствуют в одной папке, но отсутствуют в другой.

Практическое руководство. Запрос самого большого файла или файлов в дереве папок (LINQ) (C#)
Демонстрация возвращения наибольшего или наименьшего файла либо определенного числа файлов в дереве каталогов.

Практическое руководство. Запрос повторяющихся файлов в дереве папок (LINQ) (C#)
Демонстрация группирования всех имен файлов, находящихся в нескольких расположениях в указанном дереве каталогов. Также показано, как выполнять более сложные сравнения на основе пользовательской функции сравнения.

Получение содержимого файлов в папке (LINQ) (C#)
Демонстрация итерации по папкам в дереве, открытия каждого файла и запроса содержимого файла.

Комментарии

Существуют некоторые сложности, связанные с созданием источника данных, который точно представляет содержимое файловой системы и корректно обрабатывает исключения. В примерах этого раздела создается моментальный снимок коллекции объектов FileInfo, представляющих все файлы в указанной корневой папке и всех ее подпапках. Фактическое состояние каждого объекта FileInfo может измениться в период между началом и окончанием выполнения запроса. Например, можно создать список объектов FileInfo для использования в качестве источника данных. При попытке доступа к свойству `Length` в запросе объект FileInfo попытается получить доступ к файловой системе для обновления значения `Length`. Если файл больше не существует, вы получите исключение FileNotFoundException в запросе, даже если не запрашиваете файловую систему напрямую. Некоторые запросы в этом разделе используют отдельный метод, который использует эти конкретные исключения в определенных случаях. Другой возможностью является поддержка источника данных, обновляемого

динамически с помощью FileSystemWatcher.

См. также

- LINQ to Objects (C#)

Как запросить файлы с указанными атрибутами или именем (C#)

18.12.2021 • 2 minutes to read

В этом примере показано, как обнаружить все файлы с указанным расширением (например, TXT) в заданном дереве каталогов. Кроме того, он показывает, как обнаружить самый новый или самый старый файл в дереве, используя время создания.

Пример

```
class FindFileByExtension
{
    // This query will produce the full path for all .txt files
    // under the specified folder including subfolders.
    // It orders the list according to the file name.
    static void Main()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

        //Create the query
        IEnumerable<System.IO.FileInfo> fileQuery =
            from file in fileList
            where file.Extension == ".txt"
            orderby file.Name
            select file;

        //Execute the query. This might write out a lot of files!
        foreach (System.IO.FileInfo fi in fileQuery)
        {
            Console.WriteLine(fi.FullName);
        }

        // Create and execute a new query by using the previous
        // query as a starting point. fileQuery is not
        // executed again until the call to Last()
        var newestFile =
            (from file in fileQuery
             orderby file.CreationTime
             select new { file.FullName, file.CreationTime })
            .Last();

        Console.WriteLine("\r\nThe newest .txt file is {0}. Creation time: {1}",
            newestFile.FullName, newestFile.CreationTime);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
```

Компиляция кода

Создайте проект консольного приложения C# с директивами `using` для пространств имен System.Linq и System.IO.

См. также

- LINQ to Objects (C#)
- LINQ и каталоги файлов (C#)

Группировка файлов по расширению (LINQ) (C#)

18.12.2021 • 2 minutes to read

В этом примере показано, как можно использовать LINQ для выполнения расширенного группирования и сортировки списков файлов или папок. Кроме того, здесь показывается, как разбить на страницы выходные данные в окне консоли с помощью методов Skip и Take.

Пример

Следующий запрос демонстрирует группирование содержимого указанного дерева каталогов по расширению файла.

```
class GroupByExtension
{
    // This query will sort all the files under the specified folder
    //  and subfolder into groups keyed by the file extension.
    private static void Main()
    {
        // Take a snapshot of the file system.
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\Common7";

        // Used in WriteLine to trim output lines.
        int trimLength = startFolder.Length;

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

        // Create the query.
        var queryGroupByExt =
            from file in fileList
            group file by file.Extension.ToLower() into fileGroup
            orderby fileGroup.Key
            select fileGroup;

        // Display one group at a time. If the number of
        // entries is greater than the number of lines
        // in the console window, then page the output.
        PageOutput(trimLength, queryGroupByExt);
    }

    // This method specifically handles group queries of FileInfo objects with string keys.
    // It can be modified to work for any long listings of data. Note that explicit typing
    // must be used in method signatures. The groupbyExtList parameter is a query that produces
    // groups of FileInfo objects with string keys.
    private static void PageOutput(int rootLength,
                                   IEnumerable<System.Linq.IGrouping<string, System.IO.FileInfo>>
groupByExtList)
    {
        // Flag to break out of paging loop.
        bool goAgain = true;

        // "3" = 1 line for extension + 1 for "Press any key" + 1 for input cursor.
        int numLines = Console.WindowHeight - 3;

        // Iterate through the outer collection of groups.
```

```
        foreach (var filegroup in groupByExtList)
        {
            // Start a new extension at the top of a page.
            int currentLine = 0;

            // Output only as many lines of the current group as will fit in the window.
            do
            {
                Console.Clear();
                Console.WriteLine(filegroup.Key == String.Empty ? "[none]" : filegroup.Key);

                // Get 'numLines' number of items starting at number 'currentLine'.
                var resultPage = filegroup.Skip(currentLine).Take(numLines);

                //Execute the resultPage query
                foreach (var f in resultPage)
                {
                    Console.WriteLine("\t{0}", f.FullName.Substring(rootLength));
                }

                // Increment the line counter.
                currentLine += numLines;

                // Give the user a chance to escape.
                Console.WriteLine("Press any key to continue or the 'End' key to break...");
                ConsoleKey key = Console.ReadKey().Key;
                if (key == ConsoleKey.End)
                {
                    goAgain = false;
                    break;
                }
            } while (currentLine < filegroup.Count());

            if (goAgain == false)
                break;
        }
    }
}
```

Вывод этой программы может быть длинным в зависимости от объема данных локальной файловой системы и значения `startFolder`. В этом примере демонстрируется постраничный просмотр, который позволяет просматривать все результаты. Те же методы могут применяться для приложений Windows и веб-приложений. Обратите внимание, что поскольку код разбивает элементы в группе на страницы, требуется вложенный цикл `foreach`. Также существует некоторая дополнительная логика для вычисления текущей позиции в списке и предоставления пользователю возможности остановить разбиение по страницам и выйти из программы. В данном конкретном случае запрос разбиения на страницы выполняется для кэшированных результатов исходного запроса. В других контекстах, например LINQ to SQL, подобное кэширование не требуется.

Компиляция кода

Создайте проект консольного приложения C# с директивами `using` для пространств имен System.Linq и System.IO.

См. также

- LINQ to Objects (C#)
- LINQ и каталоги файлов (C#)

Получение общего числа байтов в наборе папок (LINQ) (C#)

18.12.2021 • 2 minutes to read

В этом примере показано, как получить общее число байтов, используемое всеми файлами в указанной папке и всех ее вложенных папках.

Пример

Метод Sum суммирует значения всех элементов, выбранных в предложении `select`. Этот запрос можно легко изменить, чтобы получить самый большой или маленький файл в заданном дереве каталогов, вызвав метод Min или Max вместо Sum.

```
class QuerySize
{
    public static void Main()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\VC#";

        // Take a snapshot of the file system.
        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<string> fileList = System.IO.Directory.GetFiles(startFolder, "*.*",
System.IO.SearchOption.AllDirectories);

        var fileQuery = from file in fileList
                        select GetFileLength(file);

        // Cache the results to avoid multiple trips to the file system.
        long[] fileLengths = fileQuery.ToArray();

        // Return the size of the largest file
        long largestFile = fileLengths.Max();

        // Return the total number of bytes in all the files under the specified folder.
        long totalBytes = fileLengths.Sum();

        Console.WriteLine("There are {0} bytes in {1} files under {2}",
            totalBytes, fileList.Count(), startFolder);
        Console.WriteLine("The largest files is {0} bytes.", largestFile);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    // This method is used to swallow the possible exception
    // that can be raised when accessing the System.IO.FileInfo.Length property.
    static long GetFileLength(string filename)
    {
        long retval;
        try
        {
            System.IO.FileInfo fi = new System.IO.FileInfo(filename);
            retval = fi.Length;
        }
        catch (System.IO.FileNotFoundException)
        {
            // If a file is no longer present,
            // just add zero bytes to the total.
            retval = 0;
        }
        return retval;
    }
}
```

Если требуется только подсчитать число байтов в указанном дереве каталогов, можно сделать это более эффективно без создания запроса LINQ, который вносит дополнительные издержки, связанные с созданием коллекции списков в качестве источника данных. Практичность подхода LINQ увеличивается по мере усложнения запроса или при необходимости выполнения нескольких запросов к одному источнику данных.

Для получения длины файла запрос вызывает отдельный метод. Это необходимо для обработки возможных исключений, которые могут возникнуть из-за удаления файла в другом потоке после создания объекта FileInfo вызовом `GetFiles`. Даже если объект FileInfo уже создан, может возникнуть исключение, так как объект FileInfo будет пытаться обновить свойство Length, используя самую последнюю длину при первом обращении к свойству. При помещении этой операции в блок try-catch вне

запроса будет выполнено правило исключения использования в запросах операций, которые могут вызвать побочные эффекты. В целом необходимо соблюдать осторожность при перехвате исключений, чтобы убедиться, что приложение не остается в неизвестном состоянии.

Компиляция кода

Создайте проект консольного приложения C# с директивами `using` для пространств имен System.Linq и System.IO.

См. также

- LINQ to Objects (C#)
- LINQ и каталоги файлов (C#)

Практическое руководство. Сравнение содержимого двух папок (LINQ) (C#)

18.12.2021 • 2 minutes to read

В этом примере демонстрируются три способа сравнения двух списков файлов:

- путем запроса логического значения, указывающего, являются ли два списка файлов идентичными;
- путем запроса пересечения для извлечения файлов, находящихся в обеих папках;
- путем запроса разности множеств для извлечения файлов, находящихся в одной папке, но отсутствующих в другой.

> **NOTE**
> Методы, представленные в этом примере, можно адаптировать для сравнения последовательностей объектов любого типа.

Класс `FileComparer`, показанный здесь, демонстрирует применение пользовательского класса сравнения вместе со стандартными операторами запросов. Класс не предназначен для использования в реальных сценариях. Он использует только имя и длину каждого файла в байтах для определения идентичности содержимого папок. В реальном сценарии для выполнения более строгой проверки равенства этот класс следует изменить.

Пример

```
namespace QueryCompareTwoDirs
{
    class CompareDirs
    {

        static void Main(string[] args)
        {

            // Create two identical or different temporary folders
            // on a local drive and change these file paths.
            string pathA = @"C:\TestDir";
            string pathB = @"C:\TestDir2";

            System.IO.DirectoryInfo dir1 = new System.IO.DirectoryInfo(pathA);
            System.IO.DirectoryInfo dir2 = new System.IO.DirectoryInfo(pathB);

            // Take a snapshot of the file system.
            IEnumerable<System.IO.FileInfo> list1 = dir1.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);
            IEnumerable<System.IO.FileInfo> list2 = dir2.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

            //A custom file comparer defined below
            FileCompare myFileCompare = new FileCompare();

            // This query determines whether the two folders contain
            // identical file lists, based on the custom file comparer
            // that is defined in the FileCompare class.
            // The query executes immediately because it returns a bool.
            bool areIdentical = list1.SequenceEqual(list2, myFileCompare);
```

```
            if (areIdentical == true)
            {
                Console.WriteLine("the two folders are the same");
            }
            else
            {
                Console.WriteLine("The two folders are not the same");
            }

            // Find the common files. It produces a sequence and doesn't
            // execute until the foreach statement.
            var queryCommonFiles = list1.Intersect(list2, myFileCompare);

            if (queryCommonFiles.Any())
            {
                Console.WriteLine("The following files are in both folders:");
                foreach (var v in queryCommonFiles)
                {
                    Console.WriteLine(v.FullName); //shows which items end up in result list
                }
            }
            else
            {
                Console.WriteLine("There are no common files in the two folders.");
            }

            // Find the set difference between the two folders.
            // For this example we only check one way.
            var queryList1Only = (from file in list1
                                  select file).Except(list2, myFileCompare);

            Console.WriteLine("The following files are in list1 but not list2:");
            foreach (var v in queryList1Only)
            {
                Console.WriteLine(v.FullName);
            }

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }

    // This implementation defines a very simple comparison
    // between two FileInfo objects. It only compares the name
    // of the files being compared and their length in bytes.
    class FileCompare : System.Collections.Generic.IEqualityComparer<System.IO.FileInfo>
    {
        public FileCompare() { }

        public bool Equals(System.IO.FileInfo f1, System.IO.FileInfo f2)
        {
            return (f1.Name == f2.Name &&
                    f1.Length == f2.Length);
        }

        // Return a hash that reflects the comparison criteria. According to the
        // rules for IEqualityComparer<T>, if Equals is true, then the hash codes must
        // also be equal. Because equality as defined here is a simple value equality, not
        // reference identity, it is possible that two or more objects will produce the same
        // hash code.
        public int GetHashCode(System.IO.FileInfo fi)
        {
            string s = $"{fi.Name}{fi.Length}";
            return s.GetHashCode();
        }
    }
}
```

```
}
```

Компиляция кода

Создайте проект консольного приложения C# с директивами `using` для пространств имен System.Linq и System.IO.

См. также

- LINQ to Objects (C#)
- LINQ и каталоги файлов (C#)

Как запросить самый большой файл или файлы в дереве папок (LINQ) (C#)

18.12.2021 • 3 minutes to read

В этом примере показано пять запросов, связанных с размером файла в байтах.

- Извлечение размера в байтах наибольшего файла.
- Извлечение размера в байтах наименьшего файла.
- Извлечение объекта FileInfo наибольшего или наименьшего файла из одной или нескольких папок в указанной корневой папке.
- Извлечение последовательности, например 10 наибольших файлов.
- Упорядочение файлов в группы на основании их размера в байтах, игнорируя файлы меньше заданного размера.

Пример

Следующий пример содержит пять отдельных запросов, которые показывают, как запрашивать и группировать файлы на основании их размера в байтах. Эти примеры можно легко изменить, чтобы создать запрос на основании других свойств объекта FileInfo.

```
class QueryBySize
{
    static void Main(string[] args)
    {
        QueryFilesBySize();
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    private static void QueryFilesBySize()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

        //Return the size of the largest file
        long maxSize =
            (from file in fileList
             let len = GetFileLength(file)
             select len)
             .Max();

        Console.WriteLine("The length of the largest file under {0} is {1}",
            startFolder, maxSize);

        // Return the FileInfo object for the largest file
        // by sorting and selecting from beginning of list
        System.IO.FileInfo longestFile =
```

```
            (from file in fileList
             let len = GetFileLength(file)
             where len > 0
             orderby len descending
             select file)
            .First();

        Console.WriteLine("The largest file under {0} is {1} with a length of {2} bytes",
                          startFolder, longestFile.FullName, longestFile.Length);

        //Return the FileInfo of the smallest file
        System.IO.FileInfo smallestFile =
            (from file in fileList
             let len = GetFileLength(file)
             where len > 0
             orderby len ascending
             select file).First();

        Console.WriteLine("The smallest file under {0} is {1} with a length of {2} bytes",
                          startFolder, smallestFile.FullName, smallestFile.Length);

        //Return the FileInfos for the 10 largest files
        // queryTenLargest is an IEnumerable<System.IO.FileInfo>
        var queryTenLargest =
            (from file in fileList
             let len = GetFileLength(file)
             orderby len descending
             select file).Take(10);

        Console.WriteLine("The 10 largest files under {0} are:", startFolder);

        foreach (var v in queryTenLargest)
        {
            Console.WriteLine("{0}: {1} bytes", v.FullName, v.Length);
        }

        // Group the files according to their size, leaving out
        // files that are less than 200000 bytes.
        var querySizeGroups =
            from file in fileList
            let len = GetFileLength(file)
            where len > 0
            group file by (len / 100000) into fileGroup
            where fileGroup.Key >= 2
            orderby fileGroup.Key descending
            select fileGroup;

        foreach (var filegroup in querySizeGroups)
        {
            Console.WriteLine(filegroup.Key.ToString() + "00000");
            foreach (var item in filegroup)
            {
                Console.WriteLine("\t{0}: {1}", item.Name, item.Length);
            }
        }
    }

    // This method is used to swallow the possible exception
    // that can be raised when accessing the FileInfo.Length property.
    // In this particular case, it is safe to swallow the exception.
    static long GetFileLength(System.IO.FileInfo fi)
    {
        long retval;
        try
        {
            retval = fi.Length;
        }
        catch (System.IO.FileNotFoundException)
        {
```

```
            // If a file is no longer present,
            // just add zero bytes to the total.
            retval = 0;
        }
        return retval;
    }

}
```

Чтобы вернуть один или несколько полных объектов FileInfo, запрос должен сначала проверить каждый из них в источнике данных, а затем отсортировать их по значению свойства Length. Затем он может вернуть один объект или последовательность объектов с максимальной длиной. First возвращает первый элемент в списке. Take возвращает первые n элементов. Укажите порядок сортировки по убыванию для размещения наименьших элементов в начале списка.

Запрос вызывает отдельный метод, чтобы получить размер файла в байтах для обработки возможных исключений, которые будут вызваны при удалении файла в другом потоке за период времени с момента создания объекта FileInfo в вызове `GetFiles`. Даже если объект FileInfo уже создан, может возникнуть исключение, так как объект FileInfo будет пытаться обновить свойство Length, используя самый последний размер в байтах при первом обращении к свойству. Поместив эту операцию в блок try-catch вне запроса, будет выполнено правило исключения использования операций в запросах, которые могут вызвать побочные эффекты. В целом, необходимо соблюдать осторожность при перехвате исключений, чтобы убедиться, что приложение не остается в неизвестном состоянии.

Компиляция кода

Создайте проект консольного приложения C# с директивами `using` для пространств имен System.Linq и System.IO.

См. также

- LINQ to Objects (C#)
- LINQ и каталоги файлов (C#)

Как запросить повторяющиеся файлы в дереве папок (LINQ) (C#)

18.12.2021 • 3 minutes to read

Иногда файлы с одинаковыми именами могут находиться в нескольких папках. Например, в папке установки Visual Studio несколько папок содержат файл readme.htm. В этом примере показано, как запросить такие повторяющиеся имена файлов в указанной корневой папке. Во втором примере показано, как запросить файлы, размер и время последней записи которых также совпадают.

Пример

```
class QueryDuplicateFileNames
{
    static void Main(string[] args)
    {
        // Uncomment QueryDuplicates2 to run that query.
        QueryDuplicates();
        // QueryDuplicates2();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    static void QueryDuplicates()
    {
        // Change the root drive or folder if necessary
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

        // used in WriteLine to keep the lines shorter
        int charsToSkip = startFolder.Length;

        // var can be used for convenience with groups.
        var queryDupNames =
            from file in fileList
            group file.FullName.Substring(charsToSkip) by file.Name into fileGroup
            where fileGroup.Count() > 1
            select fileGroup;

        // Pass the query to a method that will
        // output one page at a time.
        PageOutput<string, string>(queryDupNames);
    }

    // A Group key that can be passed to a separate method.
    // Override Equals and GetHashCode to define equality for the key.
    // Override ToString to provide a friendly name for Key.ToString()
    class PortableKey
    {
        public string Name { get; set; }
        public DateTime LastWriteTime { get; set; }
```

```
        public long Length { get; set; }

        public override bool Equals(object obj)
        {
            PortableKey other = (PortableKey)obj;
            return other.LastWriteTime == this.LastWriteTime &&
                   other.Length == this.Length &&
                   other.Name == this.Name;
        }

        public override int GetHashCode()
        {
            string str = $"{this.LastWriteTime}{this.Length}{this.Name}";
            return str.GetHashCode();
        }
        public override string ToString()
        {
            return $"{this.Name} {this.Length} {this.LastWriteTime}";
        }
    }
    static void QueryDuplicates2()
    {
        // Change the root drive or folder if necessary.
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\Common7";

        // Make the lines shorter for the console display
        int charsToSkip = startFolder.Length;

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

        // Note the use of a compound key. Files that match
        // all three properties belong to the same group.
        // A named type is used to enable the query to be
        // passed to another method. Anonymous types can also be used
        // for composite keys but cannot be passed across method boundaries
        //
        var queryDupFiles =
            from file in fileList
            group file.FullName.Substring(charsToSkip) by
                new PortableKey { Name = file.Name, LastWriteTime = file.LastWriteTime, Length = file.Length
} into fileGroup
            where fileGroup.Count() > 1
            select fileGroup;

        var list = queryDupFiles.ToList();

        int i = queryDupFiles.Count();

        PageOutput<PortableKey, string>(queryDupFiles);
    }

    // A generic method to page the output of the QueryDuplications methods
    // Here the type of the group must be specified explicitly. "var" cannot
    // be used in method signatures. This method does not display more than one
    // group per page.
    private static void PageOutput<K, V>(IEnumerable<System.Linq.IGrouping<K, V>> groupByExtList)
    {
        // Flag to break out of paging loop.
        bool goAgain = true;

        // "3" = 1 line for extension + 1 for "Press any key" + 1 for input cursor.
        int numLines = Console.WindowHeight - 3;

        // Iterate through the outer collection of groups.
        foreach (var filegroup in groupByExtList)
        {
```

```
            // Start a new extension at the top of a page.
            int currentLine = 0;

            // Output only as many lines of the current group as will fit in the window.
            do
            {
                Console.Clear();
                Console.WriteLine("Filename = {0}", filegroup.Key.ToString() == String.Empty ? "[none]" :
filegroup.Key.ToString());

                // Get 'numLines' number of items starting at number 'currentLine'.
                var resultPage = filegroup.Skip(currentLine).Take(numLines);

                //Execute the resultPage query
                foreach (var fileName in resultPage)
                {
                    Console.WriteLine("\t{0}", fileName);
                }

                // Increment the line counter.
                currentLine += numLines;

                // Give the user a chance to escape.
                Console.WriteLine("Press any key to continue or the 'End' key to break...");
                ConsoleKey key = Console.ReadKey().Key;
                if (key == ConsoleKey.End)
                {
                    goAgain = false;
                    break;
                }
            } while (currentLine < filegroup.Count());

            if (goAgain == false)
                break;
        }
    }
}
```

В первом запросе используется простой ключ для определения соответствия; при этом выполняется поиск файлов, которые имеют то же имя, однако их содержимое может быть другим. Второй запрос использует составной ключ для сравнения трех свойств объекта FileInfo. Этот запрос дает гораздо больше шансов найти файлы, имеющие одинаковые имена и схожее или идентичное содержимое.

Компиляция кода

Создайте проект консольного приложения C# с директивами `using` для пространств имен System.Linq и System.IO.

См. также

- LINQ to Objects (C#)
- LINQ и каталоги файлов (C#)

Как запросить содержимое текстовых файлов в папке (LINQ) (C#)

18.12.2021 • 2 minutes to read

В этом примере показано, как запросить все файлы в указанном дереве каталогов, открыть каждый файл и проверить его содержимое. Этот способ позволяет создать индексы для содержимого дерева каталогов или обратить их порядок. В этом примере выполняется простой поиск строки. Более сложные типы сопоставления шаблонов можно выполнять с помощью регулярных выражений. Дополнительные сведения см. в разделе Практическое руководство. Объединение запросов LINQ с помощью регулярных выражений (C#).

Пример

```
class QueryContents
{
    public static void Main()
    {
        // Modify this path as necessary.
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

        string searchTerm = @"Visual Studio";

        // Search the contents of each file.
        // A regular expression created with the RegEx class
        // could be used instead of the Contains method.
        // queryMatchingFiles is an IEnumerable<string>.
        var queryMatchingFiles =
            from file in fileList
            where file.Extension == ".htm"
            let fileText = GetFileText(file.FullName)
            where fileText.Contains(searchTerm)
            select file.FullName;

        // Execute the query.
        Console.WriteLine("The term \"{0}\" was found in:", searchTerm);
        foreach (string filename in queryMatchingFiles)
        {
            Console.WriteLine(filename);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // Read the contents of the file.
    static string GetFileText(string name)
    {
        string fileContents = String.Empty;

        // If the file has been deleted since we took
        // the snapshot, ignore it and return the empty string.
        if (System.IO.File.Exists(name))
        {
            fileContents = System.IO.File.ReadAllText(name);
        }
        return fileContents;
    }
}
```

Компиляция кода

Создайте проект консольного приложения C# с директивами `using` для пространств имен System.Linq и System.IO.

См. также

- LINQ и каталоги файлов (C#)

- LINQ to Objects (C#)

Выполнение запроса к ArrayList с помощью LINQ (C#)

18.12.2021 • 2 minutes to read

При использовании LINQ для запросов к неуниверсальным коллекциям IEnumerable, таким как ArrayList, необходимо явно объявить тип переменной диапазона, чтобы отразить конкретный тип объектов в коллекции. Например, если у вас есть список ArrayList объектов `Student`, предложение from должно иметь следующий вид:

```
var query = from Student s in arrList
//...
```

Указав тип переменной диапазона, вы приводите каждый элемент в ArrayList к `Student`.

Использование явным образом типизированной переменной диапазона в выражении запроса эквивалентно вызову метода Cast. Если выполнить приведение не удается, Cast создает исключение. Методы Cast и OfType стандартного оператора запроса используются для работы с неуниверсальными типами IEnumerable. Дополнительные сведения см. в разделе Связи типов в операциях запроса LINQ.

Пример

В следующем примере показан простой запрос к объекту ArrayList. Обратите внимание на то, что в этом примере инициализаторы объектов используются, когда код вызывает метод Add, но это не обязательно.

```
using System;
using System.Collections;
using System.Linq;

namespace NonGenericLINQ
{
    public class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int[] Scores { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            ArrayList arrList = new ArrayList();
            arrList.Add(
                new Student
                    {
                        FirstName = "Svetlana", LastName = "Omelchenko", Scores = new int[] { 98, 92, 81, 60
}
                    });
            arrList.Add(
                new Student
                    {
                        FirstName = "Claire", LastName = "O'Donnell", Scores = new int[] { 75, 84, 91, 39 }
                    });
            arrList.Add(
                new Student
                    {
                        FirstName = "Sven", LastName = "Mortensen", Scores = new int[] { 88, 94, 65, 91 }
                    });
            arrList.Add(
                new Student
                    {
                        FirstName = "Cesar", LastName = "Garcia", Scores = new int[] { 97, 89, 85, 82 }
                    });

            var query = from Student student in arrList
                        where student.Scores[0] > 95
                        select student;

            foreach (Student s in query)
                Console.WriteLine(s.LastName + ": " + s.Scores[0]);

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }
}
/* Output:
    Omelchenko: 98
    Garcia: 97
*/
```

См. также

- LINQ to Objects (C#)

Практическое руководство. Добавление настраиваемых методов для запросов LINQ (C#)

18.12.2021 • 4 minutes to read

Вы можете расширить набор методов, которые можно использовать для запросов LINQ, путем добавления методов расширения в интерфейс IEnumerable<T>. Например, помимо стандартных операций вычисления среднего или максимального значения, можно создать настраиваемый метод агрегирования для вычисления одного значения на основе последовательности значений. Также можно создать метод, который работает как настраиваемый фильтр или особое преобразование данных для последовательности значений и возвращает новую последовательность. Примерами таких методов являются Distinct, Skip и Reverse.

При расширении интерфейса IEnumerable<T> настраиваемые методы можно применять к любой перечислимой коллекции. Дополнительные сведения см. в разделе Методы расширения.

Добавление метода агрегирования

Метод агрегирования вычисляет одно значение на основе набора значений. LINQ реализует несколько методов агрегирования, включая Average, Min и Max. Вы можете создать собственный метод агрегирования, добавив метод расширения в интерфейс IEnumerable<T>.

В следующем примере кода показано, как создать метод расширения `Median` для вычисления срединного значения последовательности чисел типа `double`.

```
public static class EnumerableExtension
{
    public static double Median(this IEnumerable<double>? source)
    {
        if (source is null || !source.Any())
        {
            throw new InvalidOperationException("Cannot compute median for a null or empty set.");
        }

        var sortedList =
            source.OrderBy(number => number).ToList();

        int itemIndex = sortedList.Count / 2;

        if (sortedList.Count % 2 == 0)
        {
            // Even number of items.
            return (sortedList[itemIndex] + sortedList[itemIndex - 1]) / 2;
        }
        else
        {
            // Odd number of items.
            return sortedList[itemIndex];
        }
    }
}
```

Этот метод расширения вызывается для любых перечислимых коллекций так же, как другие методы агрегирования из интерфейса IEnumerable<T>.

В следующем примере кода показано использование метода `Median` для массива типа `double`.

```
double[] numbers = { 1.9, 2, 8, 4, 5.7, 6, 7.2, 0 };
var query = numbers.Median();

Console.WriteLine($"double: Median = {query}");
// This code produces the following output:
//     double: Median = 4.85
```

Перегрузка метода агрегирования для принятия различных типов

Метод агрегирования можно перегрузить, чтобы он принимал последовательности различных типов. Стандартный способ — создание перегрузки для каждого типа. Другой подход заключается в создании перегрузки, которая будет принимать универсальный тип и преобразовывать его в конкретный тип с помощью делегата. Вы можете сочетать оба способа.

Создание перегрузки для каждого типа

Вы можете создать конкретную перегрузку для каждого типа, который требуется поддерживать. В следующем примере кода показана перегрузка метода `Median` для типа `int`.

```
// int overload
public static double Median(this IEnumerable<int> source) =>
    (from number in source select (double)number).Median();
```

Теперь можно вызвать перегрузки `Median` для типов `integer` и `double`, как показано в следующем примере кода:

```
double[] numbers1 = { 1.9, 2, 8, 4, 5.7, 6, 7.2, 0 };
var query1 = numbers1.Median();

Console.WriteLine($"double: Median = {query1}");

int[] numbers2 = { 1, 2, 3, 4, 5 };
var query2 = numbers2.Median();

Console.WriteLine($"int: Median = {query2}");
// This code produces the following output:
//     double: Median = 4.85
//     int: Median = 3
```

Создание универсальной перегрузки

Вы также можете создать перегрузку, которая принимает последовательность универсальных объектов. Эта перегрузка принимает делегат в качестве параметра и использует его для преобразования последовательности объектов универсального типа в конкретный тип.

В следующем примере кода демонстрируется перегрузка метода `Median`, принимающая делегат Func<T,TResult> в качестве параметра. Этот делегат принимает объект универсального типа T и возвращает объект типа `double`.

```
// generic overload
public static double Median<T>(
    this IEnumerable<T> numbers, Func<T, double> selector) =>
    (from num in numbers select selector(num)).Median();
```

Теперь вы можете вызвать метод `Median` для последовательности объектов любого типа. Если тип не содержит собственную перегрузку метода, вам потребуется передать параметр делегата. В C# для этой цели можно использовать лямбда-выражение. Кроме того, только в Visual Basic, если вы используете предложение `Aggregate` или `Group By` вместо вызова метода, вы можете передать любое значение или выражение, которое находится в области этого предложения.

В следующем примере кода показано, как вызвать метод `Median` для массива целых чисел и массива строк. Для строк вычисляется срединное значение длин строк в массиве. В примере демонстрируется передача параметра делегата Func<T,TResult> в метод `Median` для каждого из случаев.

```
int[] numbers3 = { 1, 2, 3, 4, 5 };

/*
    You can use the num => num lambda expression as a parameter for the Median method
    so that the compiler will implicitly convert its value to double.
    If there is no implicit conversion, the compiler will display an error message.
*/
var query3 = numbers3.Median(num => num);

Console.WriteLine($"int: Median = {query3}");

string[] numbers4 = { "one", "two", "three", "four", "five" };

// With the generic overload, you can also use numeric properties of objects.
var query4 = numbers4.Median(str => str.Length);

Console.WriteLine($"string: Median = {query4}");
// This code produces the following output:
//     int: Median = 3
//     string: Median = 4
```

Добавление метода, возвращающего последовательность

Вы можете расширить интерфейс IEnumerable<T> с помощью метода настраиваемого запроса, который возвращает последовательность значений. В этом случае метод должен возвращать коллекцию типа IEnumerable<T>. Такие методы можно использовать для применения фильтров или преобразований данных к последовательности значений.

В следующем примере показано, как создать метод расширения с именем `AlternateElements`, который возвращает каждый второй элемент в коллекции, начиная с первого элемента.

```
// Extension method for the IEnumerable<T> interface.
// The method returns every other element of a sequence.
public static IEnumerable<T> AlternateElements<T>(this IEnumerable<T> source)
{
    int index = 0;
    foreach (T element in source)
    {
        if (index % 2 == 0)
        {
            yield return element;
        }

        index++;
    }
}
```

Этот метод расширения вызывается для любых перечислимых коллекций так же, как другие методы из интерфейса IEnumerable<T>, как показано в следующем примере кода:

```
string[] strings = { "a", "b", "c", "d", "e" };

var query5 = strings.AlternateElements();

foreach (var element in query5)
{
    Console.WriteLine(element);
}
// This code produces the following output:
//     a
//     c
//     e
```

См. также

- IEnumerable<T>
- Методы расширения

LINQ to ADO.NET (Страница портала)

18.12.2021 • 2 minutes to read

LINQ to ADO.NET позволяет запрашивать любой перечисляемый объект в ADO.NET с помощью модели программирования LINQ.

> **NOTE**
> Документация по LINQ to ADO.NET находится в разделе ADO.NET пакета SDK для .NET Framework: LINQ и ADO.NET.

Существуют три отдельные технологии ADO.NET LINQ: LINQ to DataSet, LINQ to SQL и LINQ to Entities. Технология LINQ to DataSet обеспечивает расширенные и оптимизированные запросы к DataSet, LINQ to SQL позволяет запрашивать непосредственно схемы базы данных SQL Server, а LINQ to Entities — выполнять запросы к EDM.

LINQ to DataSet

DataSet является одним из наиболее широко используемых компонентов в ADO.NET и ключевым элементом модели автономного программирования, на основе которой создан ADO.NET. Несмотря на все это, объект DataSet имеет ограниченные возможности запросов.

LINQ to DataSet позволяет использовать расширенные возможности запросов в DataSet — те же функции, что и для многих других источников данных.

Дополнительные сведения см. в разделе LINQ to DataSet.

LINQ to SQL

LINQ to SQL предоставляет инфраструктуру времени выполнения для управления реляционными данными в виде объектов. В LINQ to SQL модель данных реляционной базы данных сопоставляется объектной модели, выраженной в языке программирования разработчика. При выполнении приложения LINQ to SQL преобразует интегрированные в язык запросы из объектной модели в SQL и отправляет их в базу данных для выполнения. Когда база данных возвращает результаты, LINQ to SQL переводит их обратно в объекты, которыми можно управлять.

LINQ to SQL включает поддержку хранимых процедур, определяемых пользователем функций в базе данных и наследования в объектной модели.

Дополнительные сведения см. в разделе LINQ to SQL.

LINQ to Entities

В модели EDM реляционные данные представлены в виде объектов в среде .NET. Благодаря этому поддержка LINQ эффективно реализуется на уровне объектов, что позволяет составлять запросы баз данных на языке, используемом для сборки бизнес-логики. Эта функция называется LINQ to Entities. Дополнительные сведения см. в разделе LINQ to Entities.

См. также

- LINQ и ADO.NET

- LINQ (C#)

Включение источника данных для запросов LINQ

18.12.2021 • 2 minutes to read

Существуют различные способы расширения LINQ для поддержки запросов к любому источнику данных в шаблоне LINQ. Источник данных может быть, например, структурой данных, веб-службой, файловой системой или базой данных. Шаблон LINQ позволяет клиентам легко выполнять запросы к источнику данных, для которого включен LINQ-запрос, поскольку синтаксис и шаблон запроса не меняются. Далее перечислены способы расширения LINQ для этих источников данных.

- Реализация интерфейса IEnumerable<T> в типе для выполнения запросов LINQ to Objects этого типа.
- Создание методов стандартных операторов запроса, таких как Where и Select, расширяющих тип, для поддержки настраиваемых запросов LINQ этого типа.
- Создание поставщика для источника данных, который реализует интерфейс IQueryable<T>. Поставщик, реализующий этот интерфейс, получает запросы LINQ в виде деревьев выражения, которые он может выполнять любым способом, например удаленно.
- Создание поставщика для источника данных, который использует преимущества существующей технологии LINQ. Такой поставщик позволит выполнять не только запросы, но и операции вставки, обновления, удаления и сопоставления для определяемых пользователем типов.

Эти варианты рассматриваются далее.

Включение поддержки запросов LINQ к пользовательскому источнику данных

Данные в памяти

Существует два способа включения поддержки запросов LINQ к данным, находящимся в памяти. Если данные имеют тип, который реализует IEnumerable<T>, можно выполнить запрос данных с помощью LINQ to Objects. Если добавление перечисления к типу с помощью реализации интерфейса IEnumerable<T> не имеет смысла, можно определить методы стандартных операторов запроса LINQ в типе или создать методы стандартных операторов запроса LINQ, которые расширяют тип. Пользовательские реализации стандартных операторов запроса должны использовать отложенное выполнение для возврата результатов.

Удаленные данные

Лучшим вариантом для поддержки запросов LINQ к удаленному источнику данных является реализация интерфейса IQueryable<T>. Однако это отличается от расширения поставщика, например LINQ to SQL для источника данных.

Поставщики IQueryable LINQ

Поставщики LINQ, реализующие IQueryable<T>, могут существенно различаться по сложности. В этом разделе рассматриваются различные уровни сложности.

Менее сложные поставщики `IQueryable` могут взаимодействовать с одним методом веб-службы. Этот тип поставщиков является очень специализированным, так как он требует наличия определенных сведений в обрабатываемых запросах. Он имеет закрытую систему типов, возможно, предоставляя один тип результата. Большая часть выполнения запроса происходит локально, например с помощью

реализации Enumerable стандартных операторов запроса. Менее сложный поставщик может проверять только одно выражение вызова метода в дереве выражений, которое представляет запрос, и оставляет оставшуюся логику запроса обрабатываться в другом месте.

Поставщик `IQueryable` средней сложности может использовать источник данных, который имеет ограниченно выразительный язык запросов. Если он предназначен для веб-службы, он может взаимодействовать с несколькими методами веб-службы и выбирать метод для вызова на основании вопроса, содержащегося в запросе. Поставщик средней сложности будет иметь более богатую систему типов, чем простой поставщик, но она по-прежнему будет фиксированной. Например, поставщик может предоставлять типы, имеющие связи "один ко многим", которые можно обойти, но он не предоставляет технологии отображения для определяемых пользователем типов.

Сложный поставщик `IQueryable`, например поставщик LINQ to SQL, может переводить завершенные запросы LINQ на выразительный язык запросов, например SQL. Сложные поставщики более общие, чем менее сложные, поскольку они могут обрабатывать более широкий диапазон вопросов в запросе. Он также имеет открытую систему типов и поэтому должен содержать расширяемую инфраструктуру для сопоставления определяемых пользователем типов. Разработка сложного поставщика требует значительных усилий.

См. также

- IQueryable<T>
- IEnumerable<T>
- Enumerable
- Общие сведения о стандартных операторах запроса (C#)
- LINQ to Objects (C#)

Интегрированная среда разработки Visual Studio и поддержка средств для LINQ (C#)

18.12.2021 • 2 minutes to read

Интегрированная среда разработки Visual Studio (IDE) предоставляет следующие возможности, поддерживающие разработку приложений LINQ.

Object Relational Designer

Реляционный конструктор объектов — это средство визуального проектирования, которое можно использовать в приложениях LINQ to SQL для создания классов в C#, представляющих реляционные данные в основной базе данных. Дополнительные сведения см. в разделе Средства LINQ to SQL в Visual Studio.

Средство командной строки SQLMetal

SQLMetal — это средство командной строки, которое может использоваться в процессах сборки для создания классов из существующей базы данных, предназначенных для работы в приложениях LINQ to SQL. Дополнительные сведения см. в разделе SQLMetal.exe (средство создания кода).

Редактор кода с поддержкой LINQ

Редактор кода C# поддерживает LINQ с новой технологией IntelliSense и возможностями форматирования.

Поддержка отладчика Visual Studio

Отладчик Visual Studio поддерживает отладку выражений запросов. Дополнительные сведения см. в разделе Отладка LINQ.

См. также

- LINQ (C#)

Отражение (C#)

18.12.2021 • 2 minutes to read

Механизм отражения позволяет получать объекты (типа Type), которые описывают сборки, модули и типы. Отражение можно использовать для динамического создания экземпляра типа, привязки типа к существующему объекту, а также получения типа из существующего объекта и вызова его методов или доступа к его полям и свойствам. Если в коде используются атрибуты, отражение обеспечивает доступ к ним. Дополнительные сведения см. в разделе Атрибуты.

Вот простой пример отражения, в котором для получения типа переменной используется метод GetType(), наследуемый всеми типами от базового класса `Object`:

> **NOTE**
> Убедитесь, что вы добавили `using System;` и `using System.Reflection;` в верхней части файла *.cs*.

```
// Using GetType to obtain type information:
int i = 42;
Type type = i.GetType();
Console.WriteLine(type);
```

Выходные данные: `System.Int32`.

В этом примере отражение используется для получения полного имени загруженной сборки.

```
// Using Reflection to get information of an Assembly:
Assembly info = typeof(int).Assembly;
Console.WriteLine(info);
```

Выходные данные: `System.Private.CoreLib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=7cec85d7bea7798e`

.

> **NOTE**
> Ключевые слова C# `protected` и `internal` не имеют никакого значения в промежуточном языке и не используются в интерфейсах API отражения. Соответствующими терминами в промежуточном языке являются *Family* и *Assembly*. Для идентификации метода `internal` с помощью отражения используйте свойство IsAssembly. Для идентификации метода `protected internal` используйте IsFamilyOrAssembly.

Общие сведения об отражении

Отражение удобно использовать в следующих ситуациях:

- При необходимости доступа к атрибутам в метаданных программы. Дополнительные сведения см. в разделе Извлечение информации, сохраненной в атрибуте.
- Для проверки и создания экземпляров типов в сборке.
- Для создания типов во время выполнения. Используйте классы в System.Reflection.Emit.
- Для выполнения позднего связывания, которое обеспечивает доступ к методам в типах, созданных во время выполнения. См. раздел Динамическая загрузка и использование типов.

Связанные разделы

Дополнительные сведения:

- Отражение
- Просмотр сведений о типах
- Отражение и универсальные типы
- System.Reflection.Emit
- Извлечение информации, сохраненной в атрибуте

См. также

- Руководство по программированию на C#
- Сборки в .NET

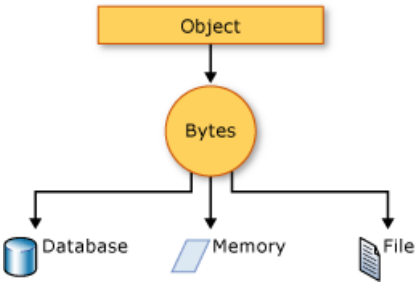
Сериализация (C#)

18.12.2021 • 3 minutes to read

Сериализация — это процесс преобразования объекта в поток байтов для сохранения или передачи в память, базу данных или файл. Эта операция предназначена для того, чтобы сохранить состояния объекта для последующего воссоздания при необходимости. Обратный процесс называется десериализацией.

Как работает сериализация

На этом рисунке показан общий процесс сериализации.



Объект сериализуется в поток, который служит для передачи данных. Поток также может содержать сведения о типе объекта, в том числе о его версии, языке и региональных параметрах, а также имени сборки. В этом формате потока объект можно сохранить в базе данных, файле или памяти.

Применение сериализации

Сериализация позволяет разработчику сохранять состояние объекта и воссоздавать его при необходимости. Это полезно для длительного хранения объектов или для обмена данными. Посредством сериализации разработчик может выполнять следующие действия:

- Отправка объекта в удаленное приложение с помощью веб-службы
- Передача объекта из одного домена в другой
- Передача объекта через брандмауэр в виде строки JSON или XML
- Хранение сведений о безопасности и пользователях между приложениями

Сериализация JSON

В пространстве имен System.Text.Json содержатся классы для сериализации и десериализации в формате JSON. JSON — это открытый стандарт, который широко применяется для обмена данными через Интернет.

В формате JSON открытые свойства объекта сериализуются в строку, массив байтов или поток в соответствии со спецификацией JSON RFC 8259. Для управления способами сериализации или десериализации с использованием JsonSerializer экземпляр класса:

- использует объект JsonSerializerOptions;
- применяет к классам или свойствам атрибуты из пространства имен System.Text.Json.Serialization;
- реализует пользовательские преобразователи.

Двоичная сериализация и сериализация XML

В пространстве имен System.Runtime.Serialization содержатся классы для сериализации и десериализации в двоичном формате и формате XML.

Двоичная сериализация использует двоичное кодирование, создавая компактные потоки для хранения или передачи через сетевые сокеты. Процесс двоичной сериализации сериализует все элементы, даже доступные только для чтения. Также этот вариант имеет более высокую производительность.

> **WARNING**
> Двоичная сериализация может быть опасной. Дополнительные сведения см. в разделе BinaryFormatter Security Guide.

При XML-сериализации все открытые поля и свойства объекта (или параметры и возвращаемые значения метода) сериализуются в XML-поток по правилам определенного документа XSD (язык определения схемы XML). XML-сериализация создает строго типизированные классы с открытыми свойствами и полями, которые преобразуются в формат XML. Пространство имен System.Xml.Serialization содержит классы для сериализации и десериализации XML. Чтобы контролировать сериализацию и десериализацию экземпляров класса, осуществляемую XmlSerializer, вы можете применять к классам и их членам специальные атрибуты.

Превращение объекта в сериализуемый

Для двоичной сериализации или сериализации XML вам потребуются:

- сериализуемый объект;
- поток, содержащий сериализованный объект;
- экземпляр System.Runtime.Serialization.Formatter.

Примените к типу атрибут SerializableAttribute, чтобы указать возможность сериализации экземпляров этого типа. Если в типе нет атрибута SerializableAttribute, при попытке сериализации выдается исключение.

Чтобы запретить сериализацию поля, примените атрибут NonSerializedAttribute. Если поле сериализуемого типа содержит указатель, дескриптор или специальные структуры данных для определенной среды, и содержимое этого поле невозможно разумно воссоздать в другой среде, такое поле лучше сделать несериализуемым.

Если сериализуемый класс содержит ссылки на объекты других классов, имеющие пометку SerializableAttribute, эти объекты тоже будут сериализованы.

Базовая и пользовательская сериализации

Существует два способа выполнить двоичную сериализацию или сериализацию XML — базовый и пользовательский.

Базовая сериализация использует платформу .NET для автоматической сериализации объекта. Единственное требование заключается в необходимости применить атрибут SerializableAttribute к классу. Атрибут NonSerializedAttribute также можно использовать для исключения из сериализации определенных полей.

При использовании базовой сериализации управление версиями объектов может привести к ошибкам. Если проблемы с управлением версиями важны, рекомендуется использовать пользовательскую сериализацию. Базовая сериализация является самым простым способом сериализации, но не дает почти никакого контроля над процессом.

Используя пользовательскую сериализацию, вы можете точно указать, какие объекты и как будут сериализованы. Класс должен иметь отметку SerializableAttribute и реализовывать интерфейс ISerializable. Если вы хотите настраивать и десериализацию объекта, следует использовать пользовательский

конструктор.

Сериализация конструктора

Сериализация конструктора — это особая форма сериализации, при которой применяется способ постоянного хранения объектов, используемый в средствах разработки. Сериализация конструктора выполняет преобразование графа объекта в файл исходного кода, с помощью которого впоследствии можно восстановить граф объекта. Этот файл исходного кода может содержать программный код, разметку или даже информацию из таблицы SQL.

Связанные разделы и примеры

В разделе Обзор System.Text.Json описывается способ получить библиотеку `System.Text.Json`.

Практическое руководство. Сериализация и десериализация JSON в .NET. Содержит сведения о том, как считывать и записывать данные объекта в формате JSON с использованием класса JsonSerializer.

Пошаговое руководство: Сохранение объекта в Visual Studio (C#)
Демонстрирует, как с помощью сериализации сохранить данные объекта между экземплярами, чтобы сохранять значения и извлекать их при следующем создании экземпляра объекта.

Практическое руководство. Чтение данных объекта из XML-файла (C#)
Показывает считывание данных объекта, которые ранее были записаны в XML-файл с помощью класса XmlSerializer.

Практическое руководство. Запись данных объекта в XML-файл (C#)
Показывает, как записать объект из класса в XML-файл с помощью класса XmlSerializer.

Запись данных объекта в XML-файл (C#)

18.12.2021 • 2 minutes to read

Показывает, как записать объект из класса в XML-файл с помощью класса XmlSerializer.

Пример

```
public class XMLWrite
{

   static void Main(string[] args)
    {
        WriteXML();
    }

    public class Book
    {
        public String title;
    }

    public static void WriteXML()
    {
        Book overview = new Book();
        overview.title = "Serialization Overview";
        System.Xml.Serialization.XmlSerializer writer =
            new System.Xml.Serialization.XmlSerializer(typeof(Book));

        var path = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments) +
"//SerializationOverview.xml";
        System.IO.FileStream file = System.IO.File.Create(path);

        writer.Serialize(file, overview);
        file.Close();
    }
}
```

Компиляция кода

В сериализуемом классе должен быть открытый конструктор без параметров.

Отказоустойчивость

При следующих условиях возможно возникновение исключения:

- В сериализуемом классе нет открытого конструктора без параметров.
- Файл существует и является файлом только для чтения (IOException).
- Слишком длинный путь (PathTooLongException).
- Диск заполнен (IOException).

Безопасность .NET

В этом примере создается файл (если файл отсутствует). Если приложению требуется создать файл, оно должно иметь доступ к каталогу для создания файлов (`Create`). Если файл уже существует, то

приложению достаточно иметь лишь доступ для записи файлов (`Write`), т. е. меньшие привилегии. Безопаснее создавать файл во время развертывания, если это возможно, а также предоставлять доступ `Read` к отдельному файлу вместо доступа `Create` к папке.

См. также

- StreamWriter
- Практическое руководство. Чтение данных объекта из XML-файла (C#)
- Сериализация (C#)

Чтение данных объекта из XML-файла (C#)

18.12.2021 • 2 minutes to read

В этом примере демонстрируется считывание данных объекта, которые ранее были записаны в XML-файл с помощью класса XmlSerializer.

Пример

```
public class Book
{
    public String title;
}

public void ReadXML()
{
    // First write something so that there is something to read ...
    var b = new Book { title = "Serialization Overview" };
    var writer = new System.Xml.Serialization.XmlSerializer(typeof(Book));
    var wfile = new System.IO.StreamWriter(@"c:\temp\SerializationOverview.xml");
    writer.Serialize(wfile, b);
    wfile.Close();

    // Now we can read the serialized book ...
    System.Xml.Serialization.XmlSerializer reader =
        new System.Xml.Serialization.XmlSerializer(typeof(Book));
    System.IO.StreamReader file = new System.IO.StreamReader(
        @"c:\temp\SerializationOverview.xml");
    Book overview =  (Book)reader.Deserialize(file);
    file.Close();

    Console.WriteLine(overview.title);

}
```

Компиляция кода

Замените имя файла "c:\temp\SerializationOverview.xml" на имя файла, в котором содержатся сериализованные данные. Дополнительные сведения о сериализации данных см. в руководстве по записи данных объекта в XML-файл (C#).

У класса должен быть открытый конструктор без параметров.

Десериализуются только открытые свойства и поля.

Отказоустойчивость

При следующих условиях возможно возникновение исключения:

- В сериализуемом классе нет открытого конструктора без параметров.
- Данные в файле не являются данными из класса, который десериализуется.
- Файл не существует (IOException).

Безопасность .NET

Всегда проверяйте входные данные и никогда не десериализуйте данные из непроверенных источников. Созданный заново объект выполняется на локальном компьютере с разрешениями кода, который его десериализовал. Следует проверять все входные данные перед использованием их в приложении.

См. также

- StreamWriter
- Практическое руководство. Запись данных объекта в XML-файл (C#)
- Сериализация (C#)
- Руководство по программированию на C#

Пошаговое руководство. Сохранение объекта с помощью C#

18.12.2021 • 4 minutes to read

С помощью сериализации можно сохранить данные объекта между экземплярами, что позволит сохранять значения и извлекать их при следующем создании экземпляра объекта.

В этом пошаговом руководстве будет создан базовый объект `Loan`, а его данные сохранены в файл. Затем при повторном создании объекта вы получите данные из файла.

> **IMPORTANT**
> В этом примере создается файл (если файл отсутствует). Если приложение должно создать файл, ему необходимо разрешение `Create` для папки. Для задания разрешений используются списки управления доступом. Если файл уже существует, приложению требуется лишь разрешение `Write` (с более низким уровнем). Если возможно, безопаснее создать файл во время развертывания и предоставить только разрешения `Read` для одного файла (вместо разрешения Create для папки). По тем же соображениям рекомендуется записывать данные в пользовательские папки, а не в корневую папку или папку Program Files.

> **IMPORTANT**
> В этом примере данные сохраняются в двоичном формате. Эти форматы не следует использовать для конфиденциальных данных, таких как пароли или сведения о кредитных картах.

Предварительные требования

- Для сборки и запуска установите пакет SDK для .NET.
- Установите любой привычный для вас редактор кода, если еще не сделали этого.

> **TIP**
> Нужно ли устанавливать редактор кода? Попробуйте использовать Visual Studio.

- Этот пример требует C# версии 7.3. См. сведения о выборе версии языка C#.

Можно просмотреть образец кода онлайн в репозитории GitHub с примерами .NET.

Создание объекта Loan

Первым шагом является создание класса `Loan` и консольного приложения, которое использует этот класс:

1. Создание нового приложения. Введите `dotnet new console -o serialization` для создания нового консольного приложения в подкаталоге `serialization`.
2. Откройте приложение в редакторе и добавьте новый класс с именем `Loan.cs`.
3. Добавьте приведенный ниже код к классу `Loan`:

```
public class Loan : INotifyPropertyChanged
 {
     public double LoanAmount { get; set; }
     public double InterestRatePercent { get; set; }

     [field:NonSerialized()]
     public DateTime TimeLastLoaded { get; set; }

     public int Term { get; set; }

     private string customer;
     public string Customer
     {
         get { return customer; }
         set
         {
             customer = value;
             PropertyChanged?.Invoke(this,
               new PropertyChangedEventArgs(nameof(Customer)));
         }
     }

     [field: NonSerialized()]
     public event System.ComponentModel.PropertyChangedEventHandler PropertyChanged;

     public Loan(double loanAmount,
                 double interestRate,
                 int term,
                 string customer)
     {
         this.LoanAmount = loanAmount;
         this.InterestRatePercent = interestRate;
         this.Term = term;
         this.customer = customer;
     }
 }
```

Также потребуется создать приложение, которое использует класс `Loan`.

Сериализация объекта Loan

1. Откройте `Program.cs`. Добавьте следующий код:

```
Loan TestLoan = new Loan(10000.0, 7.5, 36, "Neil Black");
```

Добавьте обработчик событий для события `PropertyChanged` и несколько строк, чтобы изменить объект `Loan` и отобразить изменения. Дополнения отображаются в следующем коде:

```
TestLoan.PropertyChanged += (_, __) => Console.WriteLine($"New customer value: {TestLoan.Customer}");

TestLoan.Customer = "Henry Clay";
Console.WriteLine(TestLoan.InterestRatePercent);
TestLoan.InterestRatePercent = 7.1;
Console.WriteLine(TestLoan.InterestRatePercent);
```

На этом этапе можно запустить код и посмотреть текущие выходные данные:

```
New customer value: Henry Clay
7.5
7.1
```

При повторном запуске приложения выводятся одни и те же значения. При каждом запуске программы создается новый объект Loan. В реальной жизни процентная ставка время от времени меняется, но не при каждом запуске этого приложения. Код сериализации позволяет сохранять последние процентные ставки между экземплярами приложения. На следующем шаге для этого в класс Loan будет добавлена сериализация.

Использование сериализации для хранения объекта

Чтобы сохранить значения для класса Loan, прежде всего необходимо отметить класс атрибутом `Serializable`. Добавьте следующий код непосредственно перед определением класса Loan:

```
[Serializable()]
```

Атрибут SerializableAttribute сообщает компилятору, что все находящееся в классе может быть сохранено в файле. Поскольку событие `PropertyChanged` не представляет часть графа объекта, который должен быть сохранен, оно не подлежит сериализации. В противном случае будут сериализованы все объекты, присоединенные к этому событию. Можно добавить NonSerializedAttribute к объявлению поля для обработчика событий `PropertyChanged`.

```
[field: NonSerialized()]
public event System.ComponentModel.PropertyChangedEventHandler PropertyChanged;
```

Начиная с C# 7.3 можно прикреплять атрибуты к резервному полю автоматически реализуемого свойства с помощью значения целевого объекта `field`. Следующий код добавляет свойство `TimeLastLoaded` и помечает его как не подлежащее сериализации:

```
[field:NonSerialized()]
public DateTime TimeLastLoaded { get; set; }
```

Следующим шагом будет добавление кода сериализации в приложение LoanApp. Чтобы выполнить сериализацию класса и записать данные в файл, используйте пространства имен System.IO и System.Runtime.Serialization.Formatters.Binary. Во избежание ввода полных имен можно добавить ссылки на необходимые пространства имен, как показано в следующем коде:

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
```

Следующим шагом является добавление кода для десериализации объекта из файла при создании объекта. Добавьте в класс константу для имени файла сериализованных данных, как показано в следующем коде:

```
const string FileName = @"../../../SavedLoan.bin";
```

Затем добавьте следующий код после строки, которая создает объект `TestLoan`:

```
if (File.Exists(FileName))
{
    Console.WriteLine("Reading saved file");
    Stream openFileStream = File.OpenRead(FileName);
    BinaryFormatter deserializer = new BinaryFormatter();
    TestLoan = (Loan)deserializer.Deserialize(openFileStream);
    TestLoan.TimeLastLoaded = DateTime.Now;
    openFileStream.Close();
}
```

Прежде всего убедитесь, что файл существует. Если он существует, создайте класс Stream для чтения двоичного файла и класс BinaryFormatter для преобразования файла. Кроме того, необходимо преобразовать тип потока в тип объекта Loan.

Далее необходимо добавить код для сериализации класса в файл. Добавьте следующий код после существующего кода в методе `Main`:

```
Stream SaveFileStream = File.Create(FileName);
BinaryFormatter serializer = new BinaryFormatter();
serializer.Serialize(SaveFileStream, TestLoan);
SaveFileStream.Close();
```

Теперь можно снова собрать и запустить приложение. При первом запуске вы заметите, что процентная ставка имеет значение 7,5, а затем меняется на 7,1. Закройте приложение, а затем снова запустите его. Теперь приложение выводит сообщение о том, что сохраненный файл прочитан и процентная ставка составляет 7,1, даже раньше кода, который меняет ее.

См. также

- Сериализация (C#)
- Руководство по программированию на C#

Операторы и выражения (Руководство по программированию на C#)

18.12.2021 • 2 minutes to read

Код приложения на C# состоит из операторов, которые включают ключевые слова, выражения и символьные операторы. В этом разделе содержатся сведения об этих базовых элементах программы на C#.

Дополнительные сведения можно найти в разделе

- Операторы
- Операторы и выражения
- Элементы, воплощающие выражение
- Сравнения на равенство

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Руководство по программированию на C#
- Приведение и преобразование типов

Инструкции (Руководство по программированию на C#)

18.12.2021 • 5 minutes to read

Действия программы выражаются в инструкциях. В общие действия включено объявление переменных, присвоение значений, вызов методов, проход по коллекциям и ветвление на один или другой блок кода, в зависимости от заданного условия. Порядок выполнения инструкций в программе называется потоком управления или потоком выполнения. Поток управления может отличаться при каждом запуске программы, в зависимости от реакции программы на входные данные, которые она получает во время выполнения.

Инструкция может состоять из одной строки кода, которая заканчивается точкой с запятой, или из ряда однострочных инструкций в блоке. Блок инструкций заключен в скобки {} и может содержать вложенные блоки. В следующем коде показаны два примера однострочных инструкций и блок многострочных инструкций:

```
    static void Main()
    {
        // Declaration statement.
        int counter;

        // Assignment statement.
        counter = 1;

        // Error! This is an expression, not an expression statement.
        // counter + 1;

        // Declaration statements with initializers are functionally
        // equivalent to  declaration statement followed by assignment statement:
        int[] radii = { 15, 32, 108, 74, 9 }; // Declare and initialize an array.
        const double pi = 3.14159; // Declare and initialize  constant.

        // foreach statement block that contains multiple statements.
        foreach (int radius in radii)
        {
            // Declaration statement with initializer.
            double circumference = pi * (2 * radius);

            // Expression statement (method invocation). A single-line
            // statement can span multiple text lines because line breaks
            // are treated as white space, which is ignored by the compiler.
            System.Console.WriteLine("Radius of circle #{0} is {1}. Circumference = {2:N2}",
                                     counter, radius, circumference);

            // Expression statement (postfix increment).
            counter++;
        } // End of foreach statement block
    } // End of Main method body.
} // End of SimpleStatements class.
/*
   Output:
    Radius of circle #1 = 15. Circumference = 94.25
    Radius of circle #2 = 32. Circumference = 201.06
    Radius of circle #3 = 108. Circumference = 678.58
    Radius of circle #4 = 74. Circumference = 464.96
    Radius of circle #5 = 9. Circumference = 56.55
*/
```

Типы инструкций

В приведенной ниже таблице перечислены различные типы инструкций в C# и связанные с ними ключевые слова со ссылками на разделы, в которых содержатся дополнительные сведения.

КАТЕГОРИЯ	КЛЮЧЕВЫЕ СЛОВА / ПРИМЕЧАНИЯ C#
Инструкции объявления	Инструкция объявления представляет новую переменную или константу. Объявление переменной может при необходимости присвоить значение переменной. В объявлении константы необходимо назначение.
Инструкции выражений	Инструкции выражений, вычисляющие значение, должны сохранить его в переменной.
Операторы выбора	Инструкции выбора позволяют ветвление на разные разделы кода, в зависимости от одного или нескольких заданных условий. Дополнительные сведения см. в следующих разделах: • if • switch
Операторы итерации	Инструкции итерации позволяют просмотр коллекций как массивов или многократное выполнение того же набора инструкций до выполнения заданного условия. Дополнительные сведения см. в следующих разделах: • do • for • foreach • while
Операторы перехода	Инструкции перехода передают управление другому разделу кода. Дополнительные сведения см. в следующих разделах: • break • continue • goto • return • yield
Операторы обработки исключений	Инструкции обработки исключений позволяют аккуратно выполнить восстановление после исключительных условий, возникающих во время выполнения. Дополнительные сведения см. в следующих разделах: • throw • try-catch • try-finally • try-catch-finally

КАТЕГОРИЯ	КЛЮЧЕВЫЕ СЛОВА / ПРИМЕЧАНИЯ C#
Инструкции checked и unchecked	Инструкции checked и unchecked позволяют указать, позволено ли числовым операциям вызывать переполнение, когда результат сохраняется в переменной, которая слишком мала для хранения результирующего значения. Дополнительные сведения см. в разделах checked и unchecked.
Инструкция `await`	Если пометить метод с помощью модификатора async, можно использовать в этом методе инструкцию await. Когда управление достигает выражения `await` в асинхронном методе, управление возвращается вызывающему объекту, и выполнение метода приостанавливается до завершения выполнения ожидающей задачи. После завершения задачи можно возобновить выполнение в методе. Простой пример см. в подразделе "Асинхронные методы" раздела Методы. Дополнительные сведения см. в разделе Асинхронное программирование с использованием ключевых слов async и await.
Инструкция `yield return`	Итератор выполняет настраиваемую итерацию по коллекции, например по списку или массиву. Итератор использует инструкцию yield return для возврата всех элементов по одному. При достижении инструкции `yield return` текущее расположение в коде запоминается. При следующем вызове итератора выполнение возобновляется с этого места. Дополнительные сведения см. в разделе Итераторы.
Инструкция `fixed`	Инструкция fixed не позволяет сборщику мусора переносить перемещаемую переменную. Дополнительные сведения см. в разделе fixed.
Инструкция `lock`	Инструкция lock позволяет ограничить одновременный доступ к блокам кода только до одного потока. Дополнительные сведения см. в разделе lock.
Инструкции с метками	Инструкцию можно пометить и затем использовать ключевое слово goto для перехода к инструкции с меткой. (См. пример в следующей строке.)
Пустая инструкция	Пустая инструкция состоит из точки с запятой. Она ничего не делает, и ее можно использовать там, где инструкция необходима, но не нужно выполнять никаких действий.

Инструкции объявления

В следующем коде приведены примеры объявления переменных с начальным значением и без него, а также пример объявления константы с необходимой инициализацией.

```
// Variable declaration statements.
double area;
double radius = 2;

// Constant declaration statement.
const double pi = 3.14159;
```

Инструкции выражений

В следующем коде показаны примеры инструкций выражений, включая присваивание, создание объекта с помощью присваивания и вызов метода.

```
// Expression statement (assignment).
area = 3.14 * (radius * radius);

// Error. Not  statement because no assignment:
//circ * 2;

// Expression statement (method invocation).
System.Console.WriteLine();

// Expression statement (new object creation).
System.Collections.Generic.List<string> strings =
    new System.Collections.Generic.List<string>();
```

Пустая инструкция

В следующих примерах показаны два способа использования пустой инструкции:

```
void ProcessMessages()
{
    while (ProcessMessage())
        ; // Statement needed here.
}

void F()
{
    //...
    if (done) goto exit;
//...
exit:
    ; // Statement needed here.
}
```

Внедренные инструкции

В некоторых инструкциях, например инструкциях итерации, всегда есть внедренная инструкция, следующая за ними. Эта внедренная инструкция может быть либо одной инструкцией, либо несколькими инструкциями, заключенными в скобки {} в блоке инструкций. Даже однострочные внедренные инструкции могут быть заключены в скобки {}, как показано в следующем примере:

```
// Recommended style. Embedded statement in  block.
foreach (string s in System.IO.Directory.GetDirectories(
                        System.Environment.CurrentDirectory))
{
    System.Console.WriteLine(s);
}

// Not recommended.
foreach (string s in System.IO.Directory.GetDirectories(
                        System.Environment.CurrentDirectory))
    System.Console.WriteLine(s);
```

Внедренная инструкция, не заключенная в скобки {}, не может быть инструкцией объявления или инструкцией с меткой. Это показано в следующем примере:

```
if(pointB == true)
    //Error CS1023:
    int radius = 5;
```

Чтобы устранить ошибку, поместите внедренную инструкцию в блок:

```
if (b == true)
{
    // OK:
    System.DateTime d = System.DateTime.Now;
    System.Console.WriteLine(d.ToLongDateString());
}
```

Вложенные блоки инструкций

Блоки инструкций могут быть вложенными, как показано в следующем коде:

```
foreach (string s in System.IO.Directory.GetDirectories(
    System.Environment.CurrentDirectory))
{
    if (s.StartsWith("CSharp"))
    {
        if (s.EndsWith("TempFolder"))
        {
            return s;
        }
    }
}
return "Not found.";
```

Недостижимые инструкции

Если компилятор определяет, что поток управления ни при каких обстоятельствах не сможет достичь определенного оператора, то он выдаст предупреждение CS0162, как показано в следующем примере:

```
// An over-simplified example of unreachable code.
const int val = 5;
if (val < 4)
{
    System.Console.WriteLine("I'll never write anything."); //CS0162
}
```

Спецификация языка C#

Дополнительные сведения см. в разделе Операторы в спецификации языка C#.

См. также

- Руководство по программированию на C#
- Ключевые слова операторов
- Операторы и выражения C#

Члены, воплощающие выражения (руководство по программированию на C#)

18.12.2021 • 3 minutes to read

Определения тела выражений позволяют предоставлять реализацию члена самым быстрым и удобочитаемым способом. Определение тела выражения можно использовать, когда логика для любого поддерживаемого члена, такого как метод или свойство, состоит из одного выражения. Определение тела выражения имеет следующий общий синтаксис:

```
member => expression;
```

здесь *expression* является допустимым выражением.

В C# 6 была представлена поддержка для определений тела выражений для методов и свойств только для чтения. В C# 7.0 эта поддержка была расширена. Определения тела выражений можно использовать с членами типа, указанными в следующей таблице.

ЧЛЕН	ПОДДЕРЖИВАЕТСЯ КАК...
Метод	C# 6
Свойство только для чтения	C# 6
Property	C# 7.0
Конструктор	C# 7.0
Метод завершения	C# 7.0
Индексатор	C# 7.0

Методы

Метод, воплощающий выражение, состоит из одного выражения, возвращающего значение, тип которого соответствует возвращаемому типу метода, или методов, возвращающих `void`, который выполняет некоторые операции. Например, типы, переопределяющие метод ToString, обычно содержат одно выражение, которое возвращает строковое представление текущего объекта.

В следующем примере определяется класс `Person`, который переопределяет метод ToString с помощью определения тела выражения. Он также определяет метод `DisplayName`, который отображает имя в консоли. Обратите внимание, что ключевое слово `return` не используется в определении тела выражения `ToString`.

```
using System;

public class Person
{
   public Person(string firstName, string lastName)
   {
      fname = firstName;
      lname = lastName;
   }

   private string fname;
   private string lname;

   public override string ToString() => $"{fname} {lname}".Trim();
   public void DisplayName() => Console.WriteLine(ToString());
}

class Example
{
   static void Main()
   {
      Person p = new Person("Mandy", "Dejesus");
      Console.WriteLine(p);
      p.DisplayName();
   }
}
```

Дополнительные сведения см. в разделе Методы (руководство по программированию на C#).

Свойства только для чтения

Начиная с C# 6 определение тела выражения можно использовать для реализации свойства только для чтения. Для этого используйте следующий синтаксис:

```
PropertyType PropertyName => expression;
```

В следующем примере определяется класс `Location`, свойство `Name` только для чтения которого реализуется как определение тела выражения, возвращающее значение закрытого поля `locationName`:

```
public class Location
{
   private string locationName;

   public Location(string name)
   {
      locationName = name;
   }

   public string Name => locationName;
}
```

Дополнительные сведения о свойствах см. в разделе Свойства (руководство по программированию на C#).

Свойства

Начиная с C# 7.0 определения тела выражения можно использовать для реализации методов доступа `get` и `set` свойства. В следующем примере показано, как это сделать:

```
public class Location
{
   private string locationName;

   public Location(string name) => Name = name;

   public string Name
   {
      get => locationName;
      set => locationName = value;
   }
}
```

Дополнительные сведения о свойствах см. в разделе Свойства (руководство по программированию на C#).

Конструкторы

Определение тела выражения для конструктора обычно состоит из одного выражения присваивания или вызова метода, который обрабатывает аргументы конструктора или инициализирует состояние экземпляра.

В следующем примере определяется класс `Location`, конструктор которого имеет один строковый параметр *name*. Определение тела выражения присваивает аргумент свойству `Name`.

```
public class Location
{
   private string locationName;

   public Location(string name) => Name = name;

   public string Name
   {
      get => locationName;
      set => locationName = value;
   }
}
```

Дополнительные сведения см. в разделе Конструкторы (руководство по программированию на C#).

Методы завершения

Определение тела выражения для метода завершения обычно содержит операторы очистки, например операторы, высвобождающие неуправляемые ресурсы.

В следующем примере определяется метод завершения, который использует определение тела выражения для указания того, что был вызван метод завершения.

```
using System;

public class Destroyer
{
   public override string ToString() => GetType().Name;

   ~Destroyer() => Console.WriteLine($"The {ToString()} finalizer is executing.");
}
```

Дополнительные сведения см. в разделе Методы завершения (руководство по программированию на C#).

Индексаторы

Как при использовании свойств, методы доступа `get` и `set` индексатора состоят из определений тела выражений, если метод доступа `get` состоит из одного выражения, которое возвращает значение, или метод доступа `set` выполняет простое присваивание.

В следующем примере определяется класс с именем `Sports`, включающий внутренний массив String, который содержит названия нескольких видов спорта. Методы доступа `get` и `set` индексатора реализуются как определения тела выражений.

```
using System;
using System.Collections.Generic;

public class Sports
{
   private string[] types = { "Baseball", "Basketball", "Football",
                              "Hockey", "Soccer", "Tennis",
                              "Volleyball" };

   public string this[int i]
   {
      get => types[i];
      set => types[i] = value;
   }
}
```

Дополнительные сведения см. в разделе Индексаторы (руководство по программированию на C#).

Проверка равенства (руководство по программированию на C#)

18.12.2021 • 3 minutes to read

Иногда возникает необходимость проверить равенство двух значений. В некоторых случаях проверяется *равенство значений*, которое также называют их *эквивалентностью*, что означает, что содержащиеся в двух переменных значения равны. В других случаях требуется определить, ссылаются ли две переменные на один и тот же объект в памяти. Такой тип равенства называется *равенством ссылок* или *идентичностью*. В этом разделе описаны эти два типа равенства и приведены ссылки на разделы, содержащие дополнительные сведения.

Равенство ссылок

Равенство ссылок означает, что два объекта ссылаются на один и тот же объект. Это может произойти при простом присваивании, как показано в следующем примере.

```
using System;
class Test
{
    public int Num { get; set; }
    public string Str { get; set; }

    static void Main()
    {
        Test a = new Test() { Num = 1, Str = "Hi" };
        Test b = new Test() { Num = 1, Str = "Hi" };

        bool areEqual = System.Object.ReferenceEquals(a, b);
        // False:
        System.Console.WriteLine("ReferenceEquals(a, b) = {0}", areEqual);

        // Assign b to a.
        b = a;

        // Repeat calls with different results.
        areEqual = System.Object.ReferenceEquals(a, b);
        // True:
        System.Console.WriteLine("ReferenceEquals(a, b) = {0}", areEqual);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

В этом фрагменте кода создается два объекта, но после инструкции присваивания они оба ссылаются на один и тот же объект. Поэтому между ними устанавливается равенство ссылок. Метод ReferenceEquals позволяет определить, ссылаются ли две ссылки на один и тот же объект.

Концепция равенства ссылок применима только к ссылочным типам. Объекты типа значения не могут иметь равенство ссылок, поскольку переменной присваивается экземпляр типа значения, создается копия этого значения. Поэтому две распакованные структуры никогда не могут ссылаться на одно и то же расположение в памяти. Более того, для сравнения двух типов значений ReferenceEquals всегда будет возвращаться результат `false`, даже если содержащиеся в объектах значения идентичны. Это связано с

тем, что каждая переменная запакована в отдельный экземпляр объекта. См. сведения в руководстве по тестированию на равенство (идентичность) ссылок.

Равенство значений

Равенство значений означает, что два объекта содержат одинаковое значение или одинаковые значения. Для примитивных типов значений, таких как int или bool, проверка на равенство значений представляет собой очевидную задачу. Можно использовать оператор ==, как показано в следующем примере.

```
int a = GetOriginalValue();
int b = GetCurrentValue();

// Test for value equality.
if (b == a)
{
    // The two integers are equal.
}
```

Для большинства других типов проверка равенства значений является более сложной, поскольку при этом необходимо понимать, как равенство определено в типе. Для всех классов и структур, имеющих несколько полей или свойств, равенство значений часто определяется как равенство значений всех полей и свойств. Например, два объекта `Point` могут быть определены как равные, если значение pointA.X равняется pointB.X, а значение pointA.Y равняется pointB.Y. Для записей равенство значений означает, что две переменные типа "запись" равны, если совпадают типы и все значения свойств и полей.

Однако общего требования о том, что равенство должно определяться на основе всех полей в типе, нет. Оно может определяться и на основе части полей. При сравнении значений типов сторонних разработчиков необходимо понимать, как именно реализовано равенство для таких типов. См. сведения об определении равенства значений в собственных классах и структурах в руководстве по определению равенства значений для типа.

Равенство значений с плавающей запятой

Проверка равенства значений с плавающей запятой (double и float) представляет проблему из-за неточности арифметических операций с плавающей запятой на двоичных компьютерах. Дополнительные сведения можно найти в заметках в разделе System.Double.

См. также

ЗАГОЛОВОК	ОПИСАНИЕ
Проверка на равенство ссылок (идентичность)	Описание процедуры проверки равенства ссылок двух переменных.
Определение равенства значений для типа	Описание предоставления пользовательского определения равенства значений для типа.
Руководство по программированию на C#	Ссылки на подробные сведения о важных элементах языка C#, а также об элементах, доступных в этом языке благодаря .NET.
Типы	Сведения о системе типов C# и ссылки на дополнительные сведения.
Записи	Предоставляет сведения о типах записей, которые по умолчанию проверяют равенство значений.

См. также раздел

- Руководство по программированию на C#

Как определить равенство значений для класса или структуры (руководство по программированию на C#)

18.12.2021 • 8 minutes to read

Записи автоматически реализуют равенство значений. Попробуйте определить `record` вместо `class`, когда ваш тип моделирует данные и должен реализовать равенство значений.

При определении класса или структуры необходимо решить, имеет ли смысл создавать пользовательское определение равенства значений (или эквивалентности) для этого типа. Обычно равенство значений реализуется, если объекты этого типа будут добавляться в коллекции или если они предназначены в первую очередь для хранения набора полей или свойств. В основу определения равенства значений можно положить сравнение всех полей и свойств в типе или только их части.

В любом случае, реализация как для классов, так и для структур, должна соответствовать следующим пяти принципам обеспечения эквивалентности (в следующих правилах предполагается, что `x`, `y` и `z` не равны NULL):

1. Рефлексивное свойство: `x.Equals(x)` возвращает `true`.
2. Симметричное свойство: `x.Equals(y)` возвращает то же значение, что и `y.Equals(x)`.
3. Транзитивное свойство: если `(x.Equals(y) && y.Equals(z))` возвращает `true`, `x.Equals(z)` возвращает `true`.
4. Последовательные вызовы `x.Equals(y)` возвращают одно и то же значение до тех пор, пока не будут изменены объекты, на которые ссылаются x и y.
5. Любое значение, отличающееся от NULL, не равно NULL. Поэтому `x.Equals(y)` вызывает исключение, если `x` имеет значение NULL. Это нарушает правила 1 или 2 в зависимости от аргумента для `Equals`.

Любая определяемая вами структура имеет заданную по умолчанию реализацию равенства значений, которая наследуется от переопределения System.ValueType метода Object.Equals(Object). Эта реализация использует отражение для проверки всех полей и свойств в типе. Хотя эта реализация возвращает верный результат, она отличается невысокой скоростью по сравнению с пользовательской реализацией, которую можно написать специально для конкретного типа.

Детали реализации равенства значений для классов и структур различаются. Однако для реализации равенства как для классов, так и для структур, необходимо выполнить одни и те же базовые действия.

1. Переопределите виртуальный метод Object.Equals(Object). В большинстве случаев пользовательская реализация `bool Equals( object obj )` должна вызывать относящийся к конкретному типу метод `Equals`, который является реализацией интерфейса System.IEquatable<T>. (См. шаг 2.)
2. Реализуйте интерфейс System.IEquatable<T>, предоставив метод `Equals` для конкретного типа. Именно на этом этапе происходит фактическое сравнение значений. Например, функцию равенства можно определить путем сравнения только одного из двух полей в типе. Не создавайте исключения в `Equals`. Для классов, которые связаны наследованием, должно соблюдаться следующее.

- этот метод должен проверять только те поля, которые объявлены в классе. Он должен вызывать метод `base.Equals` для проверки полей в базовом классе. (Не вызывайте `base.Equals`, если тип наследует напрямую от Object, так как реализация Object для Object.Equals(Object) выполняет проверку равенства ссылок.)
- Две переменные должны считаться равными, только если совпадают их типы времени выполнения. Также убедитесь, что реализация `IEquatable` метода `Equals` для типов времени выполнения используется, если типы времени выполнения и времени компиляции переменной отличаются. Одна из стратегий обеспечения правильного сравнения типов времени выполнения — реализация `IEquatable` только в классах `sealed`. Дополнительные сведения см. в примере класса ниже в этой статье.

3. Необязательно, но рекомендуется: перегрузите операторы == и !=.
4. Переопределите Object.GetHashCode таким образом, чтобы два объекта с равными значениями создавали одинаковый хэш-код.
5. Необязательные: Для поддержки определений "больше" и "меньше" реализуйте для типа интерфейс IComparable<T>, а также перегрузите операторы <= и >=.

> **NOTE**
> Начиная с версии C# 9.0 можно использовать записи для получения семантики равенства значений без ненужного стандартного кода.

Пример класса

В следующем примере показана реализация равенства значений в классе (ссылочный тип).

```
using System;

namespace ValueEqualityClass
{
    class TwoDPoint : IEquatable<TwoDPoint>
    {
        public int X { get; private set; }
        public int Y { get; private set; }

        public TwoDPoint(int x, int y)
        {
            if (x is (< 1 or > 2000) || y is (< 1 or > 2000))
            {
                throw new ArgumentException("Point must be in range 1 - 2000");
            }
            this.X = x;
            this.Y = y;
        }

        public override bool Equals(object obj) => this.Equals(obj as TwoDPoint);

        public bool Equals(TwoDPoint p)
        {
            if (p is null)
            {
                return false;
            }

            // Optimization for a common success case.
            if (Object.ReferenceEquals(this, p))
            {
                return true;
```

```
            }

            // If run-time types are not exactly the same, return false.
            if (this.GetType() != p.GetType())
            {
                return false;
            }

            // Return true if the fields match.
            // Note that the base class is not invoked because it is
            // System.Object, which defines Equals as reference equality.
            return (X == p.X) && (Y == p.Y);
        }

        public override int GetHashCode() => (X, Y).GetHashCode();

        public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs)
        {
            if (lhs is null)
            {
                if (rhs is null)
                {
                    return true;
                }

                // Only the left side is null.
                return false;
            }
            // Equals handles case of null on right side.
            return lhs.Equals(rhs);
        }

        public static bool operator !=(TwoDPoint lhs, TwoDPoint rhs) => !(lhs == rhs);
    }

    // For the sake of simplicity, assume a ThreeDPoint IS a TwoDPoint.
    class ThreeDPoint : TwoDPoint, IEquatable<ThreeDPoint>
    {
        public int Z { get; private set; }

        public ThreeDPoint(int x, int y, int z)
            : base(x, y)
        {
            if ((z < 1) || (z > 2000))
            {
                throw new ArgumentException("Point must be in range 1 - 2000");
            }
            this.Z = z;
        }

        public override bool Equals(object obj) => this.Equals(obj as ThreeDPoint);

        public bool Equals(ThreeDPoint p)
        {
            if (p is null)
            {
                return false;
            }

            // Optimization for a common success case.
            if (Object.ReferenceEquals(this, p))
            {
                return true;
            }

            // Check properties that this class declares.
            if (Z == p.Z)
            {
                // Let base class check its own fields
```

```
            // and do the run-time type comparison.
            return base.Equals((TwoDPoint)p);
        }
        else
        {
            return false;
        }
    }

    public override int GetHashCode() => (X, Y, Z).GetHashCode();

    public static bool operator ==(ThreeDPoint lhs, ThreeDPoint rhs)
    {
        if (lhs is null)
        {
            if (rhs is null)
            {
                // null == null = true.
                return true;
            }

            // Only the left side is null.
            return false;
        }
        // Equals handles the case of null on right side.
        return lhs.Equals(rhs);
    }

    public static bool operator !=(ThreeDPoint lhs, ThreeDPoint rhs) => !(lhs == rhs);
}

class Program
{
    static void Main(string[] args)
    {
        ThreeDPoint pointA = new ThreeDPoint(3, 4, 5);
        ThreeDPoint pointB = new ThreeDPoint(3, 4, 5);
        ThreeDPoint pointC = null;
        int i = 5;

        Console.WriteLine("pointA.Equals(pointB) = {0}", pointA.Equals(pointB));
        Console.WriteLine("pointA == pointB = {0}", pointA == pointB);
        Console.WriteLine("null comparison = {0}", pointA.Equals(pointC));
        Console.WriteLine("Compare to some other type = {0}", pointA.Equals(i));

        TwoDPoint pointD = null;
        TwoDPoint pointE = null;

        Console.WriteLine("Two null TwoDPoints are equal: {0}", pointD == pointE);

        pointE = new TwoDPoint(3, 4);
        Console.WriteLine("(pointE == pointA) = {0}", pointE == pointA);
        Console.WriteLine("(pointA == pointE) = {0}", pointA == pointE);
        Console.WriteLine("(pointA != pointE) = {0}", pointA != pointE);

        System.Collections.ArrayList list = new System.Collections.ArrayList();
        list.Add(new ThreeDPoint(3, 4, 5));
        Console.WriteLine("pointE.Equals(list[0]): {0}", pointE.Equals(list[0]));

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
    pointA.Equals(pointB) = True
    pointA == pointB = True
    null comparison = False
```

```
        Compare to some other type = False
        Two null TwoDPoints are equal: True
        (pointE == pointA) = False
        (pointA == pointE) = False
        (pointA != pointE) = True
        pointE.Equals(list[0]): False
    */
}
```

В классах (ссылочных типах) реализация по умолчанию обоих методов Object.Equals(Object) выполняет проверку равенства ссылок, а не значений. Когда разработчик переопределяет виртуальный метод, его задача заключается в том, чтобы реализовать семантику равенства значений.

К объектам класса можно применять операторы `==` и `!=`, даже если они не были перегружены в классе. Однако по умолчанию они служат для проверки равенства ссылок. При перегрузке в классе метода `Equals` необходимо перегрузить операторы `==` и `!=`, но это необязательно.

IMPORTANT

Приведенный выше код может не поддерживать все сценарии наследования, как ожидается. Рассмотрим следующий код.

```
TwoDPoint p1 = new ThreeDPoint(1, 2, 3);
TwoDPoint p2 = new ThreeDPoint(1, 2, 4);
Console.WriteLine(p1.Equals(p2)); // output: True
```

Этот код сообщает, что `p1` и `p2` равны несмотря на разницу в значениях `z`. Эта разница игнорируется, так как компилятор выбирает реализацию `TwoDPoint` для `IEquatable` на основе типа времени компиляции.

Встроенный механизм обеспечения равенства значений типов `record` поддерживает такие сценарии. Если бы `TwoDPoint` и `ThreeDPoint` имели тип `record`, результатом `p1.Equals(p2)` было бы значение `False`. См. дополнительные сведения о равенстве в иерархиях наследования типа `record`.

Пример структуры

В следующем примере показана реализация равенства значений в структуре (тип значения).

```
using System;

namespace ValueEqualityStruct
{
    struct TwoDPoint : IEquatable<TwoDPoint>
    {
        public int X { get; private set; }
        public int Y { get; private set; }

        public TwoDPoint(int x, int y)
            : this()
        {
            if (x is (< 1 or > 2000) || y is (< 1 or > 2000))
            {
                throw new ArgumentException("Point must be in range 1 - 2000");
            }
            X = x;
            Y = y;
        }

        public override bool Equals(object obj) => obj is TwoDPoint other && this.Equals(other);

        public bool Equals(TwoDPoint p) => X == p.X && Y == p.Y;
```

```
    public override int GetHashCode() => (X, Y).GetHashCode();

    public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs) => lhs.Equals(rhs);

    public static bool operator !=(TwoDPoint lhs, TwoDPoint rhs) => !(lhs == rhs);
}

class Program
{
    static void Main(string[] args)
    {
        TwoDPoint pointA = new TwoDPoint(3, 4);
        TwoDPoint pointB = new TwoDPoint(3, 4);
        int i = 5;

        // True:
        Console.WriteLine("pointA.Equals(pointB) = {0}", pointA.Equals(pointB));
        // True:
        Console.WriteLine("pointA == pointB = {0}", pointA == pointB);
        // True:
        Console.WriteLine("object.Equals(pointA, pointB) = {0}", object.Equals(pointA, pointB));
        // False:
        Console.WriteLine("pointA.Equals(null) = {0}", pointA.Equals(null));
        // False:
        Console.WriteLine("(pointA == null) = {0}", pointA == null);
        // True:
        Console.WriteLine("(pointA != null) = {0}", pointA != null);
        // False:
        Console.WriteLine("pointA.Equals(i) = {0}", pointA.Equals(i));
        // CS0019:
        // Console.WriteLine("pointA == i = {0}", pointA == i);

        // Compare unboxed to boxed.
        System.Collections.ArrayList list = new System.Collections.ArrayList();
        list.Add(new TwoDPoint(3, 4));
        // True:
        Console.WriteLine("pointA.Equals(list[0]): {0}", pointA.Equals(list[0]));

        // Compare nullable to nullable and to non-nullable.
        TwoDPoint? pointC = null;
        TwoDPoint? pointD = null;
        // False:
        Console.WriteLine("pointA == (pointC = null) = {0}", pointA == pointC);
        // True:
        Console.WriteLine("pointC == pointD = {0}", pointC == pointD);

        TwoDPoint temp = new TwoDPoint(3, 4);
        pointC = temp;
        // True:
        Console.WriteLine("pointA == (pointC = 3,4) = {0}", pointA == pointC);

        pointD = temp;
        // True:
        Console.WriteLine("pointD == (pointC = 3,4) = {0}", pointD == pointC);

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
    pointA.Equals(pointB) = True
    pointA == pointB = True
    Object.Equals(pointA, pointB) = True
    pointA.Equals(null) = False
    (pointA == null) = False
    (pointA != null) = True
    pointA.Equals(i) = False
```

```
        pointE.Equals(list[0]): True
        pointA == (pointC = null) = False
        pointC == pointD = True
        pointA == (pointC = 3,4) = True
        pointD == (pointC = 3,4) = True
    */
}
```

Для структур реализация по умолчанию Object.Equals(Object) (представляет собой переопределенную версию в System.ValueType) выполняет проверку равенства значений посредством отражения, сравнивая значения каждого поля в типе. Когда разработчик переопределяет виртуальный метод `Equals` в структуре, его задача состоит в том, чтобы найти более эффективный способ проверки равенства значений и, если это возможно, реализовать сравнение только на основании части полей или свойств структуры.

Операторы == и != нельзя применять к структурам, если только они явным образом не перегружены для структуры.

См. также

- Сравнения на равенство
- Руководство по программированию на C#

Руководство по программированию на C#. Тестирование на равенство (идентичность) ссылок

18.12.2021 • 3 minutes to read

Вам не требуется реализовывать настраиваемую логику, чтобы обеспечить поддержку проверки ссылок на равенство в типах. Эту возможность для всех типов реализует метод Object.ReferenceEquals.

В следующем примере показано, как проверить две переменные на *равенство ссылок*, то есть как определить, ссылаются ли они на один и тот же объект в памяти.

В этом примере также показано, почему Object.ReferenceEquals всегда возвращает `false` для типов значений и почему не следует использовать ReferenceEquals для проверки строк на равенство.

Пример

```
using System;
using System.Text;

namespace TestReferenceEquality
{
    struct TestStruct
    {
        public int Num { get; private set; }
        public string Name { get; private set; }

        public TestStruct(int i, string s) : this()
        {
            Num = i;
            Name = s;
        }
    }

    class TestClass
    {
        public int Num { get; set; }
        public string Name { get; set; }
    }

    class Program
    {
        static void Main()
        {
            // Demonstrate reference equality with reference types.
            #region ReferenceTypes

            // Create two reference type instances that have identical values.
            TestClass tcA = new TestClass() { Num = 1, Name = "New TestClass" };
            TestClass tcB = new TestClass() { Num = 1, Name = "New TestClass" };

            Console.WriteLine("ReferenceEquals(tcA, tcB) = {0}",
                              Object.ReferenceEquals(tcA, tcB)); // false

            // After assignment, tcB and tcA refer to the same object.
            // They now have reference equality.
            tcB = tcA;
            Console.WriteLine("After assignment: ReferenceEquals(tcA, tcB) = {0}",
```

```
                                Object.ReferenceEquals(tcA, tcB)); // true

            // Changes made to tcA are reflected in tcB. Therefore, objects
            // that have reference equality also have value equality.
            tcA.Num = 42;
            tcA.Name = "TestClass 42";
            Console.WriteLine("tcB.Name = {0} tcB.Num: {1}", tcB.Name, tcB.Num);
            #endregion

            // Demonstrate that two value type instances never have reference equality.
            #region ValueTypes

            TestStruct tsC = new TestStruct( 1, "TestStruct 1");

            // Value types are copied on assignment. tsD and tsC have
            // the same values but are not the same object.
            TestStruct tsD = tsC;
            Console.WriteLine("After assignment: ReferenceEquals(tsC, tsD) = {0}",
                              Object.ReferenceEquals(tsC, tsD)); // false
            #endregion

            #region stringRefEquality
            // Constant strings within the same assembly are always interned by the runtime.
            // This means they are stored in the same location in memory. Therefore,
            // the two strings have reference equality although no assignment takes place.
            string strA = "Hello world!";
            string strB = "Hello world!";
            Console.WriteLine("ReferenceEquals(strA, strB) = {0}",
                               Object.ReferenceEquals(strA, strB)); // true

            // After a new string is assigned to strA, strA and strB
            // are no longer interned and no longer have reference equality.
            strA = "Goodbye world!";
            Console.WriteLine("strA = \"{0}\" strB = \"{1}\"", strA, strB);

            Console.WriteLine("After strA changes, ReferenceEquals(strA, strB) = {0}",
                              Object.ReferenceEquals(strA, strB)); // false

            // A string that is created at runtime cannot be interned.
            StringBuilder sb = new StringBuilder("Hello world!");
            string stringC = sb.ToString();
            // False:
            Console.WriteLine("ReferenceEquals(stringC, strB) = {0}",
                              Object.ReferenceEquals(stringC, strB));

            // The string class overloads the == operator to perform an equality comparison.
            Console.WriteLine("stringC == strB = {0}", stringC == strB); // true

            #endregion

            // Keep the console open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }
}

/* Output:
    ReferenceEquals(tcA, tcB) = False
    After assignment: ReferenceEquals(tcA, tcB) = True
    tcB.Name = TestClass 42 tcB.Num: 42
    After assignment: ReferenceEquals(tsC, tsD) = False
    ReferenceEquals(strA, strB) = True
    strA = "Goodbye world!" strB = "Hello world!"
    After strA changes, ReferenceEquals(strA, strB) = False
    ReferenceEquals(stringC, strB) = False
    stringC == strB = True
*/
```

Реализация `Equals` в универсальном базовом классе System.Object также выполняет проверку на равенство ссылок, однако использовать такой подход не рекомендуется, поскольку в случае переопределения метода в классе результат может отличаться от ожидаемого. Это справедливо также для операторов `==` и `!=`. При работе со ссылочными типами операторы `==` и `!=` по умолчанию выполняют проверку на равенство ссылок. Тем не менее производные классы могут перегружать оператор для проверки на равенство значений. Чтобы свести к минимуму вероятность появления ошибки, рекомендуется всегда использовать ReferenceEquals в тех случаях, когда требуется проверить два объекта на равенство ссылок.

Константные строки в рамках одной сборки всегда интернируются во время выполнения. Таким образом, всегда присутствует только один экземпляр каждого уникального строкового литерала. Тем не менее во время выполнения не гарантируется интернирование строк, созданных во время выполнения, а также интернирование двух равных константных строк из разных сборок.

См. также

- Сравнения на равенство

Руководство по программированию на C#. Приведение и преобразование типов

18.12.2021 • 4 minutes to read

Поскольку код C# является статически типизированным во время компиляции, после объявления переменной ее нельзя объявить повторно или назначить ей значения другого типа, если этот тип невозможно неявно преобразовать в тип переменной. Например, `string` невозможно неявно преобразовать в `int`. Поэтому после объявления `i` как `int` нельзя назначить ей строку "Hello", как показано в следующем коде:

```
int i;

// error CS0029: Cannot implicitly convert type 'string' to 'int'
i = "Hello";
```

Тем не менее иногда может потребоваться скопировать значение в переменную или параметр метода другого типа. Например, может потребоваться передать целочисленную переменную в метод, параметр которого имеет тип `double`. Или может понадобиться присвоить переменную класса переменной типа интерфейса. Такого рода операции называются *преобразованиями типа*. В C# можно выполнять следующие виды преобразований.

- **Неявные преобразования**. Специальный синтаксис не требуется, так как преобразование всегда завершается успешно и данные не будут потеряны. Примеры включают преобразования из меньших в большие целочисленные типы и преобразования из производных классов в базовые классы.
- **Явные преобразования (приведения)** . Для явных преобразований требуется выражение приведения. Приведение требуется, если в ходе преобразования данные могут быть утрачены или преобразование может завершиться сбоем по другим причинам. Типичными примерами являются числовое преобразование в тип с меньшей точностью или меньшим диапазоном и преобразование экземпляра базового класса в производный класс.
- **Пользовательские преобразования**. Такие преобразования выполняются специальными методами, которые можно определить для включения явных и неявных преобразований между пользовательскими типами без связи "базовый класс — производный класс". Дополнительные сведения см. в разделе Операторы пользовательского преобразования.
- **Преобразования с использованием вспомогательных классов**. Чтобы выполнить преобразование между несовместимыми типами, например целыми числами и объектами System.DateTime или шестнадцатеричными строками и массивами байтов, можно использовать классы System.BitConverter и System.Convert, а также методы `Parse` встроенных числовых типов, такие как Int32.Parse. Дополнительные сведения см. в руководствах по преобразованию массива байтов в значение типа int, преобразованию строки в число и преобразованию из шестнадцатеричных строк в числовые типы.

Неявные преобразования

Для встроенных числовых типов неявное преобразование можно выполнить, если сохраняемое значение может уместиться в переменной без усечения или округления. При использовании целочисленных типов это означает, что диапазон исходного типа является надлежащим подмножеством диапазона для

целевого типа. Например, переменная типа long (64-разрядное целое число) может хранить любое значение, которое может хранить переменная int (32-разрядное целое число). В следующем примере компилятор неявно преобразует значение `num` справа в тип `long` перед назначением `bigNum`.

```
// Implicit conversion. A long can
// hold any value an int can hold, and more!
int num = 2147483647;
long bigNum = num;
```

Полный список всех неявных числовых преобразований см. в разделе Таблица неявных числовых преобразований в статье Встроенные числовые преобразования.

Для ссылочных типов неявное преобразование всегда предусмотрено из класса в любой из его прямых или косвенных базовых классов или интерфейсов. Никакой специальный синтаксис не требуется, поскольку производный класс всегда содержит все члены базового класса.

```
Derived d = new Derived();

// Always OK.
Base b = d;
```

Явные преобразования

Тем не менее если преобразование нельзя выполнить без риска потери данных, компилятор требует выполнения явного преобразования, которое называется *приведением*. Приведение — это способ явно указать компилятору, что необходимо выполнить преобразование и что вам известно, что может произойти потеря данных или приведение может завершиться сбоем во время выполнения. Чтобы выполнить приведение, укажите тип, в который производится приведение, в круглых скобках перед преобразуемым значением или переменной. В следующей программе выполняется приведение типа double в int. Программа не будет компилироваться без приведения.

```
class Test
{
    static void Main()
    {
        double x = 1234.7;
        int a;
        // Cast double to int.
        a = (int)x;
        System.Console.WriteLine(a);
    }
}
// Output: 1234
```

Полный список всех поддерживаемых явных числовых преобразований см. в разделе Таблица явных числовых преобразований в статье Встроенные числовые преобразования.

Для ссылочных типов явное приведение является обязательным, если необходимо преобразовать базовый тип в производный тип:

```
// Create a new derived type.
Giraffe g = new Giraffe();

// Implicit conversion to base type is safe.
Animal a = g;

// Explicit conversion is required to cast back
// to derived type. Note: This will compile but will
// throw an exception at run time if the right-side
// object is not in fact a Giraffe.
Giraffe g2 = (Giraffe)a;
```

Операция приведения между ссылочными типами не меняет тип времени выполнения базового объекта; изменяется только тип значения, который используется в качестве ссылки на этот объект. Дополнительные сведения см. в разделе Полиморфизм.

Исключения преобразования типов во время выполнения

В некоторых преобразованиях ссылочных типов компилятор не может определить, будет ли приведение допустимым. Есть вероятность, что правильно скомпилированная операция приведения завершится сбоем во время выполнения. Как показано в следующем примере, приведение типа, завершившееся сбоем во время выполнения, вызывает исключение InvalidCastException.

```
class Animal
{
    public void Eat() => System.Console.WriteLine("Eating.");

    public override string ToString() => "I am an animal.";
}

class Reptile : Animal { }
class Mammal : Animal { }

class UnSafeCast
{
    static void Main()
    {
        Test(new Mammal());

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }

    static void Test(Animal a)
    {
        // System.InvalidCastException at run time
        // Unable to cast object of type 'Mammal' to type 'Reptile'
        Reptile r = (Reptile)a;
    }
}
```

Метод `Test` имеет параметр `Animal`, поэтому явное приведение `a` аргумента к `Reptile` формирует опасное допущение. Более безопасно будет не делать допущений, а проверить тип. C# предоставляет оператор is, чтобы можно было проверить совместимость перед фактическим выполнением приведения. Дополнительные сведения см. в статье Практическое руководство. Безопасное приведение с помощью сопоставления шаблонов, а также операторов is и as.

Спецификация языка C#

Дополнительные сведения см. в разделе Преобразованияспецификация языка C#.

См. также

- Руководство по программированию на C#
- Типы
- Выражение приведения
- Операторы пользовательского преобразования
- Обобщенное преобразование типов
- Практическое руководство. Преобразование строки в число

Упаковка-преобразование и распаковка-преобразование (Руководство по программированию на C#)

18.12.2021 • 5 minutes to read

Упаковка представляет собой процесс преобразования типа значения в тип `object` или в любой другой тип интерфейса, реализуемый этим типом значения. Когда тип значения упаковывается общеязыковой средой выполнения (CLR), он инкапсулирует значение внутри экземпляра System.Object и сохраняет его в управляемой куче. Операция распаковки извлекает тип значения из объекта. Упаковка является неявной; распаковка является явной. Понятия упаковки и распаковки лежат в основе единой системы типов C#, в которой значение любого типа можно рассматривать как объект.

В следующем примере выполнена операция `i` упаковки *целочисленной переменной*, которая присвоена объекту `o`.

```
int i = 123;
// The following line boxes i.
object o = i;
```

Затем можно выполнить операцию распаковки объекта `o` и присвоить его целочисленной переменной `i`:

```
o = 123;
i = (int)o;  // unboxing
```

Следующий пример иллюстрирует использование упаковки в C#.

```
// String.Concat example.
// String.Concat has many versions. Rest the mouse pointer on
// Concat in the following statement to verify that the version
// that is used here takes three object arguments. Both 42 and
// true must be boxed.
Console.WriteLine(String.Concat("Answer", 42, true));

// List example.
// Create a list of objects to hold a heterogeneous collection
// of elements.
List<object> mixedList = new List<object>();

// Add a string element to the list.
mixedList.Add("First Group:");

// Add some integers to the list.
for (int j = 1; j < 5; j++)
{
    // Rest the mouse pointer over j to verify that you are adding
    // an int to a list of objects. Each element j is boxed when
    // you add j to mixedList.
    mixedList.Add(j);
}

// Add another string and more integers.
mixedList.Add("Second Group:");
```

```
for (int j = 5; j < 10; j++)
{
    mixedList.Add(j);
}

// Display the elements in the list. Declare the loop variable by
// using var, so that the compiler assigns its type.
foreach (var item in mixedList)
{
    // Rest the mouse pointer over item to verify that the elements
    // of mixedList are objects.
    Console.WriteLine(item);
}

// The following loop sums the squares of the first group of boxed
// integers in mixedList. The list elements are objects, and cannot
// be multiplied or added to the sum until they are unboxed. The
// unboxing must be done explicitly.
var sum = 0;
for (var j = 1; j < 5; j++)
{
    // The following statement causes a compiler error: Operator
    // '*' cannot be applied to operands of type 'object' and
    // 'object'.
    //sum += mixedList[j] * mixedList[j]);

    // After the list elements are unboxed, the computation does
    // not cause a compiler error.
    sum += (int)mixedList[j] * (int)mixedList[j];
}

// The sum displayed is 30, the sum of 1 + 4 + 9 + 16.
Console.WriteLine("Sum: " + sum);

// Output:
// Answer42True
// First Group:
// 1
// 2
// 3
// 4
// Second Group:
// 5
// 6
// 7
// 8
// 9
// Sum: 30
```

Производительность

По сравнению с простыми операциями присваивания операции упаковки и распаковки являются весьма затратными процессами с точки зрения вычислений. При выполнении упаковки типа значения необходимо создать и разместить новый объект. Объем вычислений при выполнении операции распаковки, хотя и в меньшей степени, но тоже весьма значителен. Дополнительные сведения см. в разделе Производительность.

Упаковка

Упаковка используется для хранения типов значений в куче со сбором мусора. Упаковка представляет собой неявное преобразование типа значения в тип `object` или в любой другой тип интерфейса, реализуемый этим типом значения. При упаковке типа значения в куче выделяется экземпляр объекта и выполняется копирование значения в этот новый объект.

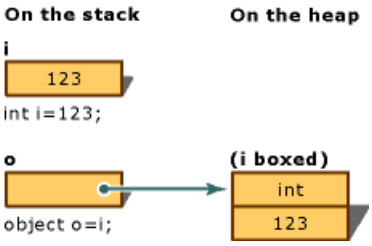
Рассмотрим следующее объявление переменной типа значения.

```
int i = 123;
```

Следующий оператор неявно применяет операцию упаковки к переменной `i`.

```
// Boxing copies the value of i into object o.
object o = i;
```

Результат этого оператора создает ссылку на объект `o` в стеке, которая ссылается на значение типа `int` в куче. Это значение является копией значения типа значения, присвоенного переменной `i`. Разница между этими двумя переменными, `i` и `o`, показана на рисунке упаковки-преобразования ниже:



Можно также выполнять упаковку явным образом, как в следующем примере, однако явная упаковка не является обязательной.

```
int i = 123;
object o = (object)i;  // explicit boxing
```

Пример

В этом примере целочисленная переменная `i` преобразуется в объект `o` при помощи упаковки. Затем значение, хранимое переменной `i`, меняется с `123` на `456`. В примере показано, что исходный тип значения и упакованный объект используют отдельные ячейки памяти, а значит могут хранить разные значения.

```
class TestBoxing
{
    static void Main()
    {
        int i = 123;

        // Boxing copies the value of i into object o.
        object o = i;

        // Change the value of i.
        i = 456;

        // The change in i doesn't affect the value stored in o.
        System.Console.WriteLine("The value-type value = {0}", i);
        System.Console.WriteLine("The object-type value = {0}", o);
    }
}
/* Output:
    The value-type value = 456
    The object-type value = 123
*/
```

Распаковка

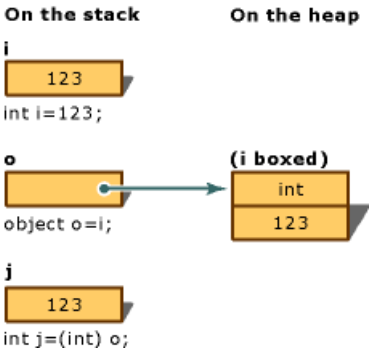
Распаковка является явным преобразованием из типа `object` в тип значения или из типа интерфейса в тип значения, реализующего этот интерфейс. Операция распаковки состоит из следующих действий:

- проверка экземпляра объекта на то, что он является упакованным значением заданного типа значения;
- копирование значения из экземпляра в переменную типа значения.

В следующем коде показаны операции по упаковке и распаковке.

```
int i = 123;      // a value type
object o = i;     // boxing
int j = (int)o;   // unboxing
```

На рисунке ниже представлен результат выполнения этого кода.



Для успешной распаковки типов значений во время выполнения необходимо, чтобы экземпляр, который распаковывается, был ссылкой на объект, предварительно созданный с помощью упаковки экземпляра этого типа значения. Попытка распаковать `null` создает исключение NullReferenceException. Попытка распаковать ссылку на несовместимый тип значения создает исключение InvalidCastException.

Пример

В следующем примере показан случай недопустимой распаковки, в результате чего создается исключение `InvalidCastException`. В случае использования `try` и `catch` при возникновении ошибки выводится сообщение.

```
class TestUnboxing
{
    static void Main()
    {
        int i = 123;
        object o = i;  // implicit boxing

        try
        {
            int j = (short)o;  // attempt to unbox

            System.Console.WriteLine("Unboxing OK.");
        }
        catch (System.InvalidCastException e)
        {
            System.Console.WriteLine("{0} Error: Incorrect unboxing.", e.Message);
        }
    }
}
```

При выполнении этой программы выводится следующий результат:

```
Specified cast is not valid. Error: Incorrect unboxing.
```

При изменении оператора:

```
int j = (short)o;
```

на:

```
int j = (int)o;
```

будет выполнено преобразование со следующим результатом:

```
Unboxing OK.
```

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Руководство по программированию на C#
- Ссылочные типы
- Типы значений

Руководство по программированию на C#. Преобразование массива байтов в значение типа int

18.12.2021 • 2 minutes to read

В этом примере демонстрируется использование класса BitConverter для преобразования массива байтов в значение типа int и обратно в массив байтов. Например, может потребоваться преобразование из байтов во встроенный тип данных после чтения байтов из сети. В дополнение к методу ToInt32(Byte[], Int32), показанному в примере, для преобразования байтов (из массива байтов) в другие встроенные типы данных можно использовать и другие методы класса BitConverter, представленные в таблице ниже.

ВОЗВРАЩАЕМЫЙ ТИП	МЕТОД
`bool`	ToBoolean(Byte[], Int32)
`char`	ToChar(Byte[], Int32)
`double`	ToDouble(Byte[], Int32)
`short`	ToInt16(Byte[], Int32)
`int`	ToInt32(Byte[], Int32)
`long`	ToInt64(Byte[], Int32)
`float`	ToSingle(Byte[], Int32)
`ushort`	ToUInt16(Byte[], Int32)
`uint`	ToUInt32(Byte[], Int32)
`ulong`	ToUInt64(Byte[], Int32)

Примеры

Этот пример инициализирует массив байтов, обращает порядок расположения элементов в массиве, если в архитектуре компьютера используется прямой порядок байтов (т. е. первым сохраняется наименее значащий байт), и затем вызывает метод ToInt32(Byte[], Int32) для преобразования четырех байтов массива в значение типа `int`. Второй аргумент ToInt32(Byte[], Int32) указывает начальный индекс массива байтов.

> **NOTE**
> Результат зависит от порядка следования байтов в архитектуре компьютера.

```
byte[] bytes = { 0, 0, 0, 25 };

// If the system architecture is little-endian (that is, little end first),
// reverse the byte array.
if (BitConverter.IsLittleEndian)
    Array.Reverse(bytes);

int i = BitConverter.ToInt32(bytes, 0);
Console.WriteLine("int: {0}", i);
// Output: int: 25
```

В этом примере вызывается метод GetBytes(Int32) класса BitConverter для преобразования значения `int` в массив байтов.

> **NOTE**
> Результат зависит от порядка следования байтов в архитектуре компьютера.

```
byte[] bytes = BitConverter.GetBytes(201805978);
Console.WriteLine("byte array: " + BitConverter.ToString(bytes));
// Output: byte array: 9A-50-07-0C
```

См. также

- BitConverter
- IsLittleEndian
- Типы

Руководство по программированию на C#. Преобразование строки в число

18.12.2021 • 4 minutes to read

Для преобразования `string` в число используется вызов метода `Parse` или `TryParse`, который можно найти в числовых типах (`int`, `long`, `double` и т. д.), или используются методы в классе System.Convert.

Немного эффективнее и проще вызвать метод `TryParse` (например, `int.TryParse("11", out number)`) или метод `Parse` (например, `var number = int.Parse("11")`). Использование метода Convert более удобно для общих объектов, реализующих IConvertible.

Можно использовать методы `Parse` или `TryParse` в числовом типе, который предположительно содержит строка, таком как тип System.Int32. Метод Convert.ToInt32 использует Parse внутри себя. Метод `Parse` возвращает преобразованное число; метод `TryParse` возвращает логическое значение, которое указывает, успешно ли выполнено преобразование, и возвращает преобразованное число в параметр `out`. Если строка имеет недопустимый формат, `Parse` создает исключение, а `TryParse` возвращает значение `false`. В случае сбоя операции синтаксического анализа при вызове метода `Parse` вы всегда должны использовать обработку исключений, чтобы перехватить FormatException.

Вызов метода Parse или TryParse

Методы `Parse` и `TryParse` игнорируют пробелы в начале и в конце строки, но все остальные символы должны быть символами, которые образуют соответствующий числовой тип (`int`, `long`, `ulong`, `float`, `decimal` и т. д.). Любые пробелы в строке, образующие число, приводят к ошибке. Например, можно использовать `decimal.TryParse` для анализа "10", "10.3" или " 10 ", но этот метод нельзя использовать для анализа 10 из "10X", "1 0" (обратите внимание на внедренный пробел), "10 .3" (обратите внимание на внедренный пробел), "10e1" (здесь работает `float.TryParse`) и т. д. Строку со значением `null` или String.Empty невозможно успешно проанализировать. Вы можете проверить наличие NULL или пустой строки, прежде чем пытаться ее проанализировать, вызвав метод String.IsNullOrEmpty.

В указанном ниже примере демонстрируются успешные и неуспешные вызовы методов `Parse` и `TryParse`.

```
using System;

public static class StringConversion
{
    public static void Main()
    {
        string input = String.Empty;
        try
        {
            int result = Int32.Parse(input);
            Console.WriteLine(result);
        }
        catch (FormatException)
        {
            Console.WriteLine($"Unable to parse '{input}'");
        }
        // Output: Unable to parse ''

        try
        {
            int numVal = Int32.Parse("-105");
            Console.WriteLine(numVal);
        }
        catch (FormatException e)
        {
            Console.WriteLine(e.Message);
        }
        // Output: -105

        if (Int32.TryParse("-105", out int j))
        {
            Console.WriteLine(j);
        }
        else
        {
            Console.WriteLine("String could not be parsed.");
        }
        // Output: -105

        try
        {
            int m = Int32.Parse("abc");
        }
        catch (FormatException e)
        {
            Console.WriteLine(e.Message);
        }
        // Output: Input string was not in a correct format.

        const string inputString = "abc";
        if (Int32.TryParse(inputString, out int numValue))
        {
            Console.WriteLine(numValue);
        }
        else
        {
            Console.WriteLine($"Int32.TryParse could not parse '{inputString}' to an int.");
        }
        // Output: Int32.TryParse could not parse 'abc' to an int.
    }
}
```

В следующем примере показан один из подходов к анализу строки, которая, как ожидается, будет включать начальные числовые символы (включая шестнадцатеричные символы) и конечные нечисловые символы. Он назначает допустимые символы в начале новой строки перед вызовом метода TryParse.

Поскольку анализируемые строки содержат небольшое количество символов, в примере вызывается метод String.Concat для назначения допустимых символов новой строке. Для большей строки можете использовать класс StringBuilder.

```
using System;

public static class StringConversion
{
    public static void Main()
    {
        var str = "  10FFxxx";
        string numericString = string.Empty;
        foreach (var c in str)
        {
            // Check for numeric characters (hex in this case) or leading or trailing spaces.
            if ((c >= '0' && c <= '9') || (char.ToUpperInvariant(c) >= 'A' && char.ToUpperInvariant(c) <=
'F') || c == ' ')
            {
                numericString = string.Concat(numericString, c.ToString());
            }
            else
            {
                break;
            }
        }

        if (int.TryParse(numericString, System.Globalization.NumberStyles.HexNumber, null, out int i))
        {
            Console.WriteLine($"'{str}' --> '{numericString}' --> {i}");
        }
        // Output: '  10FFxxx' --> '  10FF' --> 4351

        str = "   -10FFXXX";
        numericString = "";
        foreach (char c in str)
        {
            // Check for numeric characters (0-9), a negative sign, or leading or trailing spaces.
            if ((c >= '0' && c <= '9') || c == ' ' || c == '-')
            {
                numericString = string.Concat(numericString, c);
            }
            else
            {
                break;
            }
        }

        if (int.TryParse(numericString, out int j))
        {
            Console.WriteLine($"'{str}' --> '{numericString}' --> {j}");
        }
        // Output: '   -10FFXXX' --> '   -10' --> -10
    }
}
```

Вызов методов класса Convert

В следующей таблице перечислены некоторые методы класса Convert, которые можно использовать для преобразования строки в число.

ЧИСЛОВОЙ ТИП	МЕТОД
`decimal`	ToDecimal(String)

ЧИСЛОВОЙ ТИП	МЕТОД
`float`	ToSingle(String)
`double`	ToDouble(String)
`short`	ToInt16(String)
`int`	ToInt32(String)
`long`	ToInt64(String)
`ushort`	ToUInt16(String)
`uint`	ToUInt32(String)
`ulong`	ToUInt64(String)

В данном примере вызывается метод Convert.ToInt32(String) для преобразования входных данных string в значение типа int. Пример перехватывает два наиболее распространенных исключения, которые могут создаваться этим методом, — FormatException и OverflowException. Если итоговое число можно увеличить, не превышая Int32.MaxValue, пример добавляет 1 к результату и отображает вывод.

```
using System;

public class ConvertStringExample1
{
    static void Main(string[] args)
    {
        int numVal = -1;
        bool repeat = true;

        while (repeat)
        {
            Console.Write("Enter a number between –2,147,483,648 and +2,147,483,647 (inclusive): ");

            string input = Console.ReadLine();

            // ToInt32 can throw FormatException or OverflowException.
            try
            {
                numVal = Convert.ToInt32(input);
                if (numVal < Int32.MaxValue)
                {
                    Console.WriteLine("The new value is {0}", ++numVal);
                }
                else
                {
                    Console.WriteLine("numVal cannot be incremented beyond its current value");
                }
           }
            catch (FormatException)
            {
                Console.WriteLine("Input string is not a sequence of digits.");
            }
            catch (OverflowException)
            {
                Console.WriteLine("The number cannot fit in an Int32.");
            }

            Console.Write("Go again? Y/N: ");
            string go = Console.ReadLine();
            if (go.ToUpper() != "Y")
            {
                repeat = false;
            }
        }
    }
}
// Sample Output:
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive): 473
//   The new value is 474
//   Go again? Y/N: y
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive): 2147483647
//   numVal cannot be incremented beyond its current value
//   Go again? Y/N: y
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive): -1000
//   The new value is -999
//   Go again? Y/N: n
```

Руководство по программированию на C#. Преобразование шестнадцатеричных значений из строкового типа в числовой тип

18.12.2021 • 3 minutes to read

В следующих примерах кода показано выполнение указанных ниже задач.

- Получение шестнадцатеричного значения каждого символа в string.
- Получение char, соответствующего каждому значению в шестнадцатеричной строке.
- Преобразование шестнадцатеричного значения `string` в int.
- Преобразование шестнадцатеричного значения `string` в float.
- Преобразование массива byte в шестнадцатеричное значение `string`.

Примеры

Результатом следующего примера является шестнадцатеричное значение каждого символа в `string`. Сначала выполняется разбор `string` до массива символов. Затем, чтобы получить числовое значение каждого символа, для каждого из них вызывается метод ToInt32(Char). В конце формат числа меняется на шестнадцатеричный в `string`.

```
string input = "Hello World!";
char[] values = input.ToCharArray();
foreach (char letter in values)
{
    // Get the integral value of the character.
    int value = Convert.ToInt32(letter);
    // Convert the integer value to a hexadecimal value in string form.
    Console.WriteLine($"Hexadecimal value of {letter} is {value:X}");
}
/* Output:
    Hexadecimal value of H is 48
    Hexadecimal value of e is 65
    Hexadecimal value of l is 6C
    Hexadecimal value of l is 6C
    Hexadecimal value of o is 6F
    Hexadecimal value of   is 20
    Hexadecimal value of W is 57
    Hexadecimal value of o is 6F
    Hexadecimal value of r is 72
    Hexadecimal value of l is 6C
    Hexadecimal value of d is 64
    Hexadecimal value of ! is 21
 */
```

В этом примере анализируется `string` шестнадцатеричных значений и выводится символ, соответствующий каждому шестнадцатеричному значению. Сначала вызывается метод Split(Char[]) для получения каждого шестнадцатеричного значения как отдельной `string` в массиве. Затем вызывается метод ToInt32(String, Int32), который преобразует шестнадцатеричное значение в десятичное, представленное в целое число. В примере показано два разных способа получения символа, соответствующего этому коду символа. В первом случае используется ConvertFromUtf32(Int32), который

возвращает символ, соответствующий целочисленному аргументу, в виде `string`. По второму способу выполняется явное приведение `int` к char.

```
string hexValues = "48 65 6C 6C 6F 20 57 6F 72 6C 64 21";
string[] hexValuesSplit = hexValues.Split(' ');
foreach (string hex in hexValuesSplit)
{
    // Convert the number expressed in base-16 to an integer.
    int value = Convert.ToInt32(hex, 16);
    // Get the character corresponding to the integral value.
    string stringValue = Char.ConvertFromUtf32(value);
    char charValue = (char)value;
    Console.WriteLine("hexadecimal value = {0}, int value = {1}, char value = {2} or {3}",
                      hex, value, stringValue, charValue);
}
/* Output:
    hexadecimal value = 48, int value = 72, char value = H or H
    hexadecimal value = 65, int value = 101, char value = e or e
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 6F, int value = 111, char value = o or o
    hexadecimal value = 20, int value = 32, char value =   or
    hexadecimal value = 57, int value = 87, char value = W or W
    hexadecimal value = 6F, int value = 111, char value = o or o
    hexadecimal value = 72, int value = 114, char value = r or r
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 64, int value = 100, char value = d or d
    hexadecimal value = 21, int value = 33, char value = ! or !
*/
```

В этом примере показан еще один способ преобразования шестнадцатеричного `string` в целое число — с помощью метода Parse(String, NumberStyles) .

```
string hexString = "8E2";
int num = Int32.Parse(hexString, System.Globalization.NumberStyles.HexNumber);
Console.WriteLine(num);
//Output: 2274
```

В следующем примере показано преобразование шестнадцатеричного `string` в число с плавающей запятой с помощью класса System.BitConverter и метод UInt32.Parse.

```
string hexString = "43480170";
uint num = uint.Parse(hexString, System.Globalization.NumberStyles.AllowHexSpecifier);

byte[] floatVals = BitConverter.GetBytes(num);
float f = BitConverter.ToSingle(floatVals, 0);
Console.WriteLine("float convert = {0}", f);

// Output: 200.0056
```

В следующем примере показано преобразование массива байтов в шестнадцатеричную строку с помощью класса System.BitConverter.

```
byte[] vals = { 0x01, 0xAA, 0xB1, 0xDC, 0x10, 0xDD };

string str = BitConverter.ToString(vals);
Console.WriteLine(str);

str = BitConverter.ToString(vals).Replace("-", "");
Console.WriteLine(str);

/*Output:
  01-AA-B1-DC-10-DD
  01AAB1DC10DD
 */
```

В следующем примере показано преобразование массива байтов в шестнадцатеричную строку посредством вызова метода Convert.ToHexString, который был представлен в .NET 5.0.

```
byte[] array = { 0x64, 0x6f, 0x74, 0x63, 0x65, 0x74 };

string hexValue = Convert.ToHexString(array);
Console.WriteLine(hexValue);

/*Output:
  646F74636574
 */
```

См. также

- Строки стандартных числовых форматов
- Типы
- Определение представления числового значения в строке

Использование типа dynamic (руководство по программированию на C#)

18.12.2021 • 4 minutes to read

C# 4 добавляет новый тип `dynamic`. Этот тип является статическим, но объект типа `dynamic` обходит проверку статического типа. В большинстве случаев он работает как тип `object`. Во время компиляции предполагается, что элемент, типизированный как `dynamic`, поддерживает любые операции. Это значит, что вам не придется задумываться о том, получает ли объект значение из API COM, из динамического языка, такого как IronPython, из модели DOM HTML, из отражения или из другой части программы. При этом если код недопустимый, ошибки перехватываются во время выполнения.

Например, если метод экземпляра `exampleMethod1` в следующем коде имеет только один параметр, компилятор определяет, что первый вызов метода `ec.exampleMethod1(10, 4)` недопустим, поскольку содержит два аргумента. Этот вызов приводит к ошибке компилятора. Второй вызов метода `dynamic_ec.exampleMethod1(10, 4)` компилятором не проверяется, поскольку `dynamic_ec` имеет тип `dynamic`. В результате ошибка компилятора не возникает. Тем не менее ошибка не может оставаться незамеченной неограниченное время. Она перехватывается во время выполнения и вызывает исключение времени выполнения.

```
static void Main(string[] args)
{
    ExampleClass ec = new ExampleClass();
    // The following call to exampleMethod1 causes a compiler error
    // if exampleMethod1 has only one parameter. Uncomment the line
    // to see the error.
    //ec.exampleMethod1(10, 4);

    dynamic dynamic_ec = new ExampleClass();
    // The following line is not identified as an error by the
    // compiler, but it causes a run-time exception.
    dynamic_ec.exampleMethod1(10, 4);

    // The following calls also do not cause compiler errors, whether
    // appropriate methods exist or not.
    dynamic_ec.someMethod("some argument", 7, null);
    dynamic_ec.nonexistentMethod();
}
```

```
class ExampleClass
{
    public ExampleClass() { }
    public ExampleClass(int v) { }

    public void exampleMethod1(int i) { }

    public void exampleMethod2(string str) { }
}
```

Роль компилятора в этих примерах заключается в том, чтобы объединить информацию о том, какие действия каждый оператор будет выполнять с объектом или выражением, типизированным как `dynamic`. Во время выполнения сохраненная информация проверяется, а все недействительные операторы вызывают исключение времени выполнения.

Результатом большинства динамических операций является сам `dynamic`. Например, при наведении указателя мыши на `testSum` в следующем примере IntelliSense отображает тип (**локальная переменная**) dynamic testSum.

```
dynamic d = 1;
var testSum = d + 3;
// Rest the mouse pointer over testSum in the following statement.
System.Console.WriteLine(testSum);
```

Операции, в которых результатом не является `dynamic`:

- Преобразования из `dynamic` в другой тип.
- Вызовы конструктора, которые включают аргументы типа `dynamic`.

Например, `testInstance` в следующих объявлениях имеет тип `ExampleClass`, а не `dynamic`:

```
var testInstance = new ExampleClass(d);
```

Примеры преобразований приводятся в следующем разделе — "Преобразования".

Преобразования

Преобразования динамических объектов в другие типы выполнять очень просто. Это позволяет разработчику переключаться между динамическим и нединамическим поведением.

Любой объект можно преобразовать в динамический тип неявно, как показано в следующих примерах.

```
dynamic d1 = 7;
dynamic d2 = "a string";
dynamic d3 = System.DateTime.Today;
dynamic d4 = System.Diagnostics.Process.GetProcesses();
```

И наоборот, явное преобразование можно динамически применить к любому выражению типа `dynamic`.

```
int i = d1;
string str = d2;
DateTime dt = d3;
System.Diagnostics.Process[] procs = d4;
```

Разрешение перегрузки с аргументами типа dynamic

Разрешение перегрузки возникают во время выполнения, а не компиляции, если один или несколько аргументов в вызове метода имеют тип `dynamic` либо получатель вызова метода имеет тип `dynamic`. В следующем примере, если единственный доступный метод `exampleMethod2` определен как принимающий аргумент строки, отправка `d1` как аргумента не вызывает ошибку компилятора, но создает исключение времени выполнения. Разрешение перегрузки завершается ошибкой во время выполнения, поскольку `d1` имеет тип времени выполнения `int`, а `exampleMethod2` требуется строка.

```
// Valid.
ec.exampleMethod2("a string");

// The following statement does not cause a compiler error, even though ec is not
// dynamic. A run-time exception is raised because the run-time type of d1 is int.
ec.exampleMethod2(d1);
// The following statement does cause a compiler error.
//ec.exampleMethod2(7);
```

Среда DLR

Среда DLR (Dynamic Language Runtime) — это новый API в .NET Framework 4. Она предоставляет инфраструктуру, которая поддерживает тип `dynamic` в C#, а также реализацию динамических языков программирования, таких как IronPython и IronRuby. Дополнительные сведения о DLR см. в разделе Общие сведения о среде DLR.

взаимодействие COM

C# 4 включает несколько функций, улучшающих взаимодействие с интерфейсами API COM, такими как API автоматизации Office. К таким усовершенствованиям относятся применение типа `dynamic` и именованных и необязательных аргументов.

Многие методы COM допускают вариативность в типах аргументов и возвращают тип, назначая типы как `object`. В связи с этим для согласования со строго типизированными переменными в C# необходимо явно приводить значения. При компиляции с использованием параметра **EmbedInteropTypes** (параметры компилятора C#) введение типа `dynamic` позволяет обрабатывать вхождения `object` в сигнатурах COM, как если бы они имели тип `dynamic`, и таким образом избежать большей части приведения. Например, следующие инструкции показывают, как получить доступ к ячейке в электронной таблице Microsoft Office Excel с типом `dynamic` и без типа `dynamic`.

```
// Before the introduction of dynamic.
((Excel.Range)excelApp.Cells[1, 1]).Value2 = "Name";
Excel.Range range2008 = (Excel.Range)excelApp.Cells[1, 1];
```

```
// After the introduction of dynamic, the access to the Value property and
// the conversion to Excel.Range are handled by the run-time COM binder.
excelApp.Cells[1, 1].Value = "Name";
Excel.Range range2010 = excelApp.Cells[1, 1];
```

См. также

ЗАГОЛОВОК	ОПИСАНИЕ
dynamic	Описывает применение ключевого слова `dynamic`.
Общие сведения о среде DLR	Содержит общие сведения о среде DLR — среде выполнения, которая добавляет в общую среду языковой среды исполнения (CLR) набор служб для динамических языков.
Пошаговое руководство: Создание и использование динамических объектов	Содержит пошаговые инструкции по созданию пользовательских динамических объектов и проекта, который обращается к библиотеке `IronPython`.

ЗАГОЛОВОК	ОПИСАНИЕ
Практическое руководство. Доступ к объектам взаимодействия Office с помощью функций языка C#	Содержит сведения о создании проекта с использованием именованных и необязательных аргументов, типа `dynamic` и других усовершенствований, которые упрощают доступ к объектам Office API.

Пошаговое руководство. Создание и использование динамических объектов (C# и Visual Basic)

18.12.2021 • 10 minutes to read

Динамические объекты предоставляют такие элементы, как свойства и методы, во время выполнения, а не во время компиляции. Это позволяет создавать объекты для работы со структурами, не соответствующими статическому типу или формату. Например, можно использовать динамический объект для ссылки на модель DOM HTML, которая может содержать любую комбинацию допустимых элементов и атрибутов разметки HTML. Поскольку каждый документ HTML является уникальным, элементы для конкретного документа HTML определяются во время выполнения. Наиболее распространенный способ ссылки на атрибут элемента HTML заключается в передаче имени этого атрибута в метод `GetProperty` элемента. Для ссылки на атрибут `id` элемента HTML `<div id="Div1">` следует сначала получить ссылку на элемент `<div>`, а затем использовать `divElement.GetProperty("id")`. При использовании динамического объекта можно сослаться на атрибут `id` в виде `divElement.id`.

Динамические объекты обеспечивают удобный доступ к динамическим языкам, таким как IronPython и IronRuby. С помощью динамического объекта можно ссылаться на динамический скрипт, интерпретируемый во время выполнения.

Ссылка на динамический объект выполняется с помощью позднего связывания. В C# тип объектов с поздним связыванием указывается как `dynamic`. В Visual Basic тип объектов с поздним связыванием указывается как `Object`. Дополнительные сведения см. в разделах dynamic и Раннее и позднее связывание.

Вы можете создавать настраиваемые динамические объекты, используя классы из пространства имен System.Dynamic. Например, можно создать объект ExpandoObject и задать члены этого объекта во время выполнения. Также можно создать собственный тип, наследующий класс DynamicObject. Затем для обеспечения динамических функциональных возможностей во время выполнения можно переопределить члены класса DynamicObject.

Эта статья содержит два независимых пошаговых руководства.

- Создание пользовательского объекта, который динамически предоставляет содержимое текстового файла в виде свойств объекта.
- Создание проекта, использующего библиотеку `IronPython`.

Вы можете выполнить одно из них или оба, в последнем случае порядок не имеет значения.

Предварительные требования

- Visual Studio 2019 версии 16.9 или более поздней с установленной рабочей нагрузкой **Разработка классических приложений** .NET. Пакет SDK для .NET 5 устанавливается автоматически при выборе этой рабочей нагрузки.

> **NOTE**
> Отображаемые на компьютере имена или расположения некоторых элементов пользовательского интерфейса Visual Studio могут отличаться от указанных в следующих инструкциях. Это зависит от имеющегося выпуска Visual Studio и используемых параметров. Дополнительные сведения см. в разделе Персонализация среды IDE.

- Для второго пошагового руководства установите IronPython для .NET. Перейдите на страницу загрузки для получения последней версии.

Создание пользовательского динамического объекта

В первом пошаговом руководстве определяется пользовательский динамический объект, выполняющий поиск по содержимому текстового файла. Динамическое свойство указывает искомый текст. Например, если в вызывающем коде указано `dynamicFile.Sample`, динамический класс возвращает общий список строк, содержащий все строки из файла, которые начинаются со слова "Sample". При поиске не учитывается регистр. Динамический класс также поддерживает два дополнительных аргумента. Первый аргумент — это значение перечисления параметра поиска, задающее, где динамический класс должен искать соответствия: в начале строки, в конце строки или в любом месте строки. Второй аргумент задает, что динамический класс должен перед поиском отсекать начальные и конечные пробелы в каждой строке. Например, если в вызывающем коде указано `dynamicFile.Sample(StringSearchOption.Contains)`, динамический класс выполняет поиск слова "Sample" в любом месте строки. Если в вызывающем коде указано `dynamicFile.Sample(StringSearchOption.StartsWith, false)`, динамический класс выполняет поиск слова "Sample" в начале каждой строки и не удаляет начальные и конечные пробелы в строках. По умолчанию динамический класс выполняет поиск соответствия в начале каждой строки, предварительно удаляя начальные и конечные пробелы.

Создание пользовательского динамического класса

1. Запустите Visual Studio.
2. Выберите **Создать новый проект**.
3. В диалоговом окне **Создание нового проекта** выберите C# или Visual Basic, выберите **Консольное приложение**, а затем нажмите кнопку **Далее**.
4. В диалоговом окне **Настройка нового проекта** введите значение `DynamicSample` для параметра **Имя проекта** и нажмите кнопку **Далее**.
5. В диалоговом окне **Дополнительные сведения** выберите значение .NET 5.0 (**текущая**) для параметра **Целевая платформа**, а затем нажмите кнопку **Создать**.

   Создается новый проект.
6. В **обозревателе решений** щелкните проект DynamicSample правой кнопкой мыши и выберите **Добавить** > **Класс**. В поле **Имя** введите `ReadOnlyFile`, а затем нажмите кнопку **Добавить**.

   Будет добавлен новый файл, содержащий класс ReadOnlyFile.
7. В верхней части файла *ReadOnlyFile.cs* или *ReadOnlyFile.vb* добавьте следующий код для импорта пространств имен System.IO и System.Dynamic.

```
using System.IO;
using System.Dynamic;
```

```
Imports System.IO
Imports System.Dynamic
```

8. Пользовательский динамический объект использует перечисление для определения условия поиска. Перед оператором класса добавьте следующее определение перечисления.

```
public enum StringSearchOption
{
    StartsWith,
    Contains,
    EndsWith
}
```

```
Public Enum StringSearchOption
    StartsWith
    Contains
    EndsWith
End Enum
```

9. Обновите оператор класса, чтобы он наследовал класс `DynamicObject`, как показано в следующем примере кода.

```
class ReadOnlyFile : DynamicObject
```

```
Public Class ReadOnlyFile
    Inherits DynamicObject
```

10. Добавьте в класс `ReadOnlyFile` следующий код, чтобы задать закрытое поле для пути к файлу и конструктор для класса `ReadOnlyFile`.

```
// Store the path to the file and the initial line count value.
private string p_filePath;

// Public constructor. Verify that file exists and store the path in
// the private variable.
public ReadOnlyFile(string filePath)
{
    if (!File.Exists(filePath))
    {
        throw new Exception("File path does not exist.");
    }

    p_filePath = filePath;
}
```

```
' Store the path to the file and the initial line count value.
Private p_filePath As String

' Public constructor. Verify that file exists and store the path in
' the private variable.
Public Sub New(ByVal filePath As String)
    If Not File.Exists(filePath) Then
        Throw New Exception("File path does not exist.")
    End If

    p_filePath = filePath
End Sub
```

11. Добавьте следующий метод `GetPropertyValue` в класс `ReadOnlyFile`. Метод `GetPropertyValue` принимает в качестве входных данных условие поиска и возвращает строки текстового файла, соответствующие этому условию. Динамический метод, предоставленный классом `ReadOnlyFile`, вызывает метод `GetPropertyValue` для извлечения соответствующих результатов.

```
public List<string> GetPropertyValue(string propertyName,
                                      StringSearchOption StringSearchOption =
StringSearchOption.StartsWith,
                                      bool trimSpaces = true)
{
    StreamReader sr = null;
    List<string> results = new List<string>();
    string line = "";
    string testLine = "";

    try
    {
        sr = new StreamReader(p_filePath);

        while (!sr.EndOfStream)
        {
            line = sr.ReadLine();

            // Perform a case-insensitive search by using the specified search options.
            testLine = line.ToUpper();
            if (trimSpaces) { testLine = testLine.Trim(); }

            switch (StringSearchOption)
            {
                case StringSearchOption.StartsWith:
                    if (testLine.StartsWith(propertyName.ToUpper())) { results.Add(line); }
                    break;
                case StringSearchOption.Contains:
                    if (testLine.Contains(propertyName.ToUpper())) { results.Add(line); }
                    break;
                case StringSearchOption.EndsWith:
                    if (testLine.EndsWith(propertyName.ToUpper())) { results.Add(line); }
                    break;
            }
        }
    }
    catch
    {
        // Trap any exception that occurs in reading the file and return null.
        results = null;
    }
    finally
    {
        if (sr != null) {sr.Close();}
    }

    return results;
}
```

```
Public Function GetPropertyValue(ByVal propertyName As String,
                                 Optional ByVal StringSearchOption As StringSearchOption =
StringSearchOption.StartsWith,
                                 Optional ByVal trimSpaces As Boolean = True) As List(Of String)

    Dim sr As StreamReader = Nothing
    Dim results As New List(Of String)
    Dim line = ""
    Dim testLine = ""

    Try
        sr = New StreamReader(p_filePath)

        While Not sr.EndOfStream
            line = sr.ReadLine()

            ' Perform a case-insensitive search by using the specified search options.
            testLine = UCase(line)
            If trimSpaces Then testLine = Trim(testLine)

            Select Case StringSearchOption
                Case StringSearchOption.StartsWith
                    If testLine.StartsWith(UCase(propertyName)) Then results.Add(line)
                Case StringSearchOption.Contains
                    If testLine.Contains(UCase(propertyName)) Then results.Add(line)
                Case StringSearchOption.EndsWith
                    If testLine.EndsWith(UCase(propertyName)) Then results.Add(line)
            End Select
        End While
    Catch
        ' Trap any exception that occurs in reading the file and return Nothing.
        results = Nothing
    Finally
        If sr IsNot Nothing Then sr.Close()
    End Try

    Return results
End Function
```

12. После метода `GetPropertyValue` добавьте следующий код, чтобы переопределить метод TryGetMember класса DynamicObject. Метод TryGetMember вызывается при запросе члена динамического класса без указания аргументов. Аргумент `binder` содержит сведения об элементе, на который дается ссылка, а аргумент `result` ссылается на результат, возвращенный для указанного элемента. Метод TryGetMember возвращает логическое значение `true`, если запрошенный элемент существует. В противном случае возвращается `false`.

```
// Implement the TryGetMember method of the DynamicObject class for dynamic member calls.
public override bool TryGetMember(GetMemberBinder binder,
                                  out object result)
{
    result = GetPropertyValue(binder.Name);
    return result == null ? false : true;
}
```

```
' Implement the TryGetMember method of the DynamicObject class for dynamic member calls.
Public Overrides Function TryGetMember(ByVal binder As GetMemberBinder,
                                       ByRef result As Object) As Boolean
    result = GetPropertyValue(binder.Name)
    Return If(result Is Nothing, False, True)
End Function
```

13. После метода `TryGetMember` добавьте следующий код, чтобы переопределить метод TryInvokeMember класса DynamicObject. Метод TryInvokeMember вызывается при запросе члена динамического класса с аргументами. Аргумент `binder` содержит сведения об элементе, на который дается ссылка, а аргумент `result` ссылается на результат, возвращенный для указанного элемента. Аргумент `args` содержит массив аргументов, передаваемых в элемент. Метод TryInvokeMember возвращает логическое значение `true`, если запрошенный элемент существует. В противном случае возвращается `false`.

    Пользовательская версия метода `TryInvokeMember` ожидает, что первый аргумент будет значением из перечисления `StringSearchOption`, заданного на предыдущем шаге. Метод `TryInvokeMember` ожидает, что второй аргумент будет логическим значением. Если один или оба элемента имеют допустимые значения, они передаются в метод `GetPropertyValue` для получения результатов.

```
// Implement the TryInvokeMember method of the DynamicObject class for
// dynamic member calls that have arguments.
public override bool TryInvokeMember(InvokeMemberBinder binder,
                                     object[] args,
                                     out object result)
{
    StringSearchOption StringSearchOption = StringSearchOption.StartsWith;
    bool trimSpaces = true;

    try
    {
        if (args.Length > 0) { StringSearchOption = (StringSearchOption)args[0]; }
    }
    catch
    {
        throw new ArgumentException("StringSearchOption argument must be a StringSearchOption enum
value.");
    }

    try
    {
        if (args.Length > 1) { trimSpaces = (bool)args[1]; }
    }
    catch
    {
        throw new ArgumentException("trimSpaces argument must be a Boolean value.");
    }

    result = GetPropertyValue(binder.Name, StringSearchOption, trimSpaces);

    return result == null ? false : true;
}
```

```
' Implement the TryInvokeMember method of the DynamicObject class for
' dynamic member calls that have arguments.
Public Overrides Function TryInvokeMember(ByVal binder As InvokeMemberBinder,
                                          ByVal args() As Object,
                                          ByRef result As Object) As Boolean

    Dim StringSearchOption As StringSearchOption = StringSearchOption.StartsWith
    Dim trimSpaces = True

    Try
        If args.Length > 0 Then StringSearchOption = CType(args(0), StringSearchOption)
    Catch
        Throw New ArgumentException("StringSearchOption argument must be a StringSearchOption enum
value.")
    End Try

    Try
        If args.Length > 1 Then trimSpaces = CType(args(1), Boolean)
    Catch
        Throw New ArgumentException("trimSpaces argument must be a Boolean value.")
    End Try

    result = GetPropertyValue(binder.Name, StringSearchOption, trimSpaces)

    Return If(result Is Nothing, False, True)
End Function
```

14. Сохраните и закройте файл.

Создание примера текстового файла

1. В **обозревателе решений** щелкните проект DynamicSample правой кнопкой мыши и выберите **Добавить** > **Новый элемент**. На панели **Установленные шаблоны** выберите **Общие**, а затем шаблон **Текстовый файл**. В поле **Имя** оставьте имя по умолчанию *TextFile1.txt* и нажмите кнопку **Добавить**. В проект добавится новый текстовый файл.

2. Скопируйте в файл *TextFile1.txt* следующий текст.

```
List of customers and suppliers

Supplier: Lucerne Publishing (https://www.lucernepublishing.com/)
Customer: Preston, Chris
Customer: Hines, Patrick
Customer: Cameron, Maria
Supplier: Graphic Design Institute (https://www.graphicdesigninstitute.com/)
Supplier: Fabrikam, Inc. (https://www.fabrikam.com/)
Customer: Seubert, Roxanne
Supplier: Proseware, Inc. (http://www.proseware.com/)
Customer: Adolphi, Stephan
Customer: Koch, Paul
```

3. Сохраните и закройте файл.

Создание примера приложения, в котором применяется пользовательский динамический объект

1. В **обозревателе решений** дважды щелкните файл *Program.vb*, если используется Visual Basic, или файл *Program.cs*, если используется Visual C#.

2. Добавьте следующий код в процедуру `Main`, чтобы создать экземпляр класса `ReadOnlyFile` для файла *TextFile1.txt*. В этом коде используется позднее связывание для вызова динамических элементов и извлечения строк текста, которые содержат строку "Customer".

```
dynamic rFile = new ReadOnlyFile(@"..\..\..\TextFile1.txt");
foreach (string line in rFile.Customer)
{
    Console.WriteLine(line);
}
Console.WriteLine("----------------------------");
foreach (string line in rFile.Customer(StringSearchOption.Contains, true))
{
    Console.WriteLine(line);
}
```

```
Dim rFile As Object = New ReadOnlyFile("..\..\..\TextFile1.txt")
For Each line In rFile.Customer
    Console.WriteLine(line)
Next
Console.WriteLine("----------------------------")
For Each line In rFile.Customer(StringSearchOption.Contains, True)
    Console.WriteLine(line)
Next
```

3. Сохраните файл и нажмите клавиши `CTRL+F5 для сборки и запуска приложения.`

Вызов библиотеки динамического языка

В следующем пошаговом руководстве создается проект, который осуществляет доступ к библиотеке, написанной на динамическом языке IronPython.

Создание пользовательского динамического класса

1. В Visual Studio выберите **Файл** > **Создать** > **Проект**.
2. В диалоговом окне **Создание нового проекта** выберите C# или Visual Basic, выберите **Консольное приложение**, а затем нажмите кнопку **Далее**.
3. В диалоговом окне **Настройка нового проекта** введите значение `DynamicIronPythonSample` для параметра **Имя проекта** и нажмите кнопку **Далее**.
4. В диалоговом окне **Дополнительные сведения** выберите значение .NET 5.0 (**текущая**) для параметра **Целевая платформа**, а затем нажмите кнопку **Создать**.

   Создается новый проект.
5. Установите пакет NuGet IronPython.
6. Если вы используете Visual Basic, отредактируйте файл *Program.vb*. Если вы используете Visual C#, отредактируйте файл *Program.cs*.
7. В верхней части файла добавьте следующий код для импорта пространств имен `Microsoft.Scripting.Hosting` и `IronPython.Hosting` из библиотек IronPython и пространства имен `System.Linq`.

```
using System.Linq;
using Microsoft.Scripting.Hosting;
using IronPython.Hosting;
```

```
Imports Microsoft.Scripting.Hosting
Imports IronPython.Hosting
Imports System.Linq
```

8. В методе Main добавьте следующий код, чтобы создать объект `Microsoft.Scripting.Hosting.ScriptRuntime`, в котором будут размещены библиотеки IronPython. Объект `ScriptRuntime` загружает модуль библиотеки IronPython random.py.

```
// Set the current directory to the IronPython libraries.
System.IO.Directory.SetCurrentDirectory(
   Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles) +
   @"\IronPython 2.7\Lib");

// Create an instance of the random.py IronPython library.
Console.WriteLine("Loading random.py");
ScriptRuntime py = Python.CreateRuntime();
dynamic random = py.UseFile("random.py");
Console.WriteLine("random.py loaded.");
```

```
' Set the current directory to the IronPython libraries.
System.IO.Directory.SetCurrentDirectory(
    Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles) &
       "\IronPython 2.7\Lib")

' Create an instance of the random.py IronPython library.
Console.WriteLine("Loading random.py")
Dim py = Python.CreateRuntime()
Dim random As Object = py.UseFile("random.py")
Console.WriteLine("random.py loaded.")
```

9. После указания в коде необходимости загрузки модуля random.py добавьте следующий код, чтобы создать массив целых чисел. Массив передается методу `shuffle` модуля random.py, который произвольно сортирует значения в массиве.

```
// Initialize an enumerable set of integers.
int[] items = Enumerable.Range(1, 7).ToArray();

// Randomly shuffle the array of integers by using IronPython.
for (int i = 0; i < 5; i++)
{
    random.shuffle(items);
    foreach (int item in items)
    {
        Console.WriteLine(item);
    }
    Console.WriteLine("--------------------");
}
```

```
' Initialize an enumerable set of integers.
Dim items = Enumerable.Range(1, 7).ToArray()

' Randomly shuffle the array of integers by using IronPython.
For i = 0 To 4
    random.shuffle(items)
    For Each item In items
        Console.WriteLine(item)
    Next
    Console.WriteLine("--------------------")
Next
```

10. Сохраните файл и нажмите клавиши `CTRL+F5 для сборки и запуска приложения.`

См. также

- System.Dynamic
- System.Dynamic.DynamicObject
- Использование типа dynamic
- Раннее и позднее связывание
- dynamic
- Реализация динамических интерфейсов (документ в формате PDF на сайте Microsoft TechNet)

Практическое руководство. Управление версиями с помощью ключевых слов "Override" и "New" (Руководство по программированию в C#)

18.12.2021 • 4 minutes to read

Язык C# построен таким образом, что управление версиями базовых и производных классов в различных библиотеках может включать сохранение и расширение обратной совместимости. Это означает, например, то, что C# полностью поддерживает введение в базовый класс нового члена с таким же именем, как у члена производного класса, и никакое непредвиденное поведение при этом не возникает. Кроме того, это значит, что класс должен прямо указывать, в каких случаях метод будет перезаписывать унаследованный метод, а в каких он будет становиться новым методом, скрывающим одноименный унаследованный метод.

В C# производные классы могут содержать методы с такими же именами, как у методов базового класса.

- Если методу в производном классе не предшествуют ключевые слова new или override, компилятор выдаст предупреждение, а метод будет вести себя так, как если бы имелось ключевое слово `new`.
- Если методу в производном классе предшествует ключевое слово `new`, он определяется как независимый от метода в базовом классе.
- Если методу в производном классе предшествует ключевое слово `override`, объекты производного класса вызывают этот метод вместо метода базового класса.
- Чтобы применить ключевое слово `override` к методу в производном классе, необходимо фактически определить метод базового класса.
- Метод базового класса можно вызывать из производного класса с помощью ключевого слова `base`.
- Ключевые слова `override`, `virtual` и `new` можно также применять к свойствам, индексаторам и событиям.

По умолчанию методы C# не являются виртуальными. Если метод объявлен как виртуальный, любой наследующий его класс может реализовать свою собственную версию. Чтобы сделать метод виртуальным, в объявление метода базового класса добавляется модификатор `virtual`. После этого производный класс может переопределить базовый виртуальный метод с помощью ключевого слова `override` или скрыть виртуальный метод в базовом классе с помощью ключевого слова `new`. Если ключевое слово `override` или `new` не указано, компилятор выдает предупреждение, а метод в производном классе скрывает метод в базовом классе.

Чтобы продемонстрировать это на практике, предположим, что компания A создала класс с именем `GraphicsClass`, который использует ваша программа. Ниже показан `GraphicsClass`:

```
class GraphicsClass
{
    public virtual void DrawLine() { }
    public virtual void DrawPoint() { }
}
```

Ваша компания использует этот класс, на основе которого вы создаете собственный производный класс,

добавляя новый метод:

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public void DrawRectangle() { }
}
```

Ваше приложение работает, пока компания A не выпускает новую версию `GraphicsClass`, напоминающую следующий код:

```
class GraphicsClass
{
    public virtual void DrawLine() { }
    public virtual void DrawPoint() { }
    public virtual void DrawRectangle() { }
}
```

Новая версия `GraphicsClass` содержит метод с именем `DrawRectangle`. Сначала ничего не происходит. Новая версия по-прежнему совместима со старой версией на уровне двоичного кода. Любая развернутая вами программа будет работать как раньше, даже если в системе соответствующего компьютера установлен новый класс. Существующие вызовы метода `DrawRectangle` будут и дальше ссылаться на вашу версию в производном классе.

Однако, как только вы выполните повторную компиляцию приложения с новой версией `GraphicsClass`, компилятор выдаст предупреждение CS0108. В предупреждении будет сказано, что вам необходимо указать, каким образом метод `DrawRectangle` будет вести себя в приложении.

Если метод должен переопределять новый метод базового класса, используйте ключевое слово `override` :

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public override void DrawRectangle() { }
}
```

Ключевое слово `override` гарантирует, что все объекты, производные от `YourDerivedGraphicsClass`, будут использовать версию производного класса `DrawRectangle`. Объекты, производные от `YourDerivedGraphicsClass`, сохраняют доступ к версии базового класса `DrawRectangle` за счет ключевого слова base:

```
base.DrawRectangle();
```

Если вы не хотите, чтобы ваш метод переопределял новый метод базового класса, воспользуйтесь следующими рекомендациями. Чтобы избежать путаницы между двумя методами, вы можете переименовать свой метод. Эта работа требует времени и подвержена ошибкам, а в некоторых случаях непрактична. В то же время, если проект относительно небольшой, метод можно переименовать, используя параметры рефакторинга в Visual Studio. Дополнительные сведения см. в разделе Рефакторинг классов и типов (конструктор классов).

Кроме того, предупреждения можно избежать с помощью ключевого слова `new` в определении производного класса:

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public new void DrawRectangle() { }
}
```

Ключевое слово `new` сообщает компилятору о том, что ваше определение скрывает определение, которое содержится в базовом классе. Это поведение установлено по умолчанию.

Переопределение и выбор метода

Если метод в классе именован, с вызовом совместимы сразу несколько методов (например, если существуют два метода с одинаковыми именами), а параметры совместимы с переданным параметром, компилятор C# выбирает метод, наиболее подходящий для вызова. Пример совместимых методов:

```
public class Derived : Base
{
    public override void DoWork(int param) { }
    public void DoWork(double param) { }
}
```

При вызове `DoWork` к экземпляру `Derived` компилятор C#, в первую очередь, пытается сделать вызов совместимым с версиями `DoWork`, изначально объявленными в `Derived`. Методы переопределения не считаются объявленными в классе, они представляют собой новые реализации метода, объявленного в базовом классе. Только в том случае, если компилятор C# не сможет сопоставить вызов метода с исходным методом в `Derived`, он попытается сопоставить вызов с переопределенным методом, имеющим то же имя и совместимые параметры. Пример:

```
int val = 5;
Derived d = new Derived();
d.DoWork(val);  // Calls DoWork(double).
```

Поскольку переменная `val` может быть преобразована в значение double неявно, компилятор C# вызывает `DoWork(double)`, а не `DoWork(int)`. Избежать этого можно двумя способами. Во-первых, избегайте объявления новых методов, имена которых совпадают с виртуальными методами. Во-вторых, можно настроить компилятор C# на вызов виртуального метода, заставив его выполнить поиск метода базового класса путем приведения экземпляра `Derived` к `Base`. Поскольку метод виртуальный, будет вызвана реализация `DoWork(int)` в `Derived`. Пример:

```
((Base)d).DoWork(val);  // Calls DoWork(int) on Derived.
```

Дополнительные примеры `new` и `override` см. в разделе Использование ключевых слов Override и New.

См. также

- Руководство по программированию на C#
- Система типов C#
- Методы
- Наследование

Использование ключевых слов "Override" и "New" (Руководство по программированию в C#)

18.12.2021 • 9 minutes to read

В C# метод в производном классе может иметь то же имя, что и метод в базовом классе. Можно задать способ взаимодействия методов, воспользовавшись ключевыми словами new и override. Модификатор `override` *расширяет* метод `virtual` базового класса, а модификатор `new` *скрывает* доступный метод базового класса. Эта разница показана в примере в этой статье.

В консольном приложении объявите два класса — `BaseClass` и `DerivedClass`. Тип `DerivedClass` наследуется от типа `BaseClass`.

```
class BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base - Method1");
    }
}

class DerivedClass : BaseClass
{
    public void Method2()
    {
        Console.WriteLine("Derived - Method2");
    }
}
```

В методе `Main` объявите переменные `bc`, `dc` и `bcdc`.

- Параметр `bc` имеет тип `BaseClass` и его значение — тип `BaseClass`.
- Параметр `dc` имеет тип `DerivedClass` и его значение — тип `DerivedClass`.
- Параметр `bcdc` имеет тип `BaseClass` и его значение — тип `DerivedClass`. Это переменная, на которую следует обратить внимание.

Поскольку `bc` и `bcdc` имеют тип `BaseClass`, они могут только напрямую обращаться к методу `Method1` (если не используется приведение). Переменная `dc` может обращаться к `Method1` и `Method2`. Эти связи показаны в следующем коде.

```
class Program
{
    static void Main(string[] args)
    {
        BaseClass bc = new BaseClass();
        DerivedClass dc = new DerivedClass();
        BaseClass bcdc = new DerivedClass();

        bc.Method1();
        dc.Method1();
        dc.Method2();
        bcdc.Method1();
    }
    // Output:
    // Base - Method1
    // Base - Method1
    // Derived - Method2
    // Base - Method1
}
```

Далее добавьте следующий метод `Method2` в класс `BaseClass`. Сигнатура этого метода соответствует сигнатуре метода `Method2` в `DerivedClass`.

```
public void Method2()
{
    Console.WriteLine("Base - Method2");
}
```

Поскольку `BaseClass` теперь имеет метод `Method2`, можно добавить второй оператор вызова для переменных `bc` и `bcdc` класса `BaseClass`, как показано в следующем коде.

```
bc.Method1();
bc.Method2();
dc.Method1();
dc.Method2();
bcdc.Method1();
bcdc.Method2();
```

При построении проекта видно, что добавление метода `Method2` в `BaseClass` вызывает предупреждение. Предупреждение говорит, что метод `Method2` в `DerivedClass` скрывает метод `Method2` в `BaseClass`. Рекомендуется использовать ключевое слово `new` в определении `Method2`, если требуется получить такой результат. Другой вариант — переименовать один из методов `Method2` для разрешения предупреждения, но это не всегда целесообразно.

Прежде чем добавлять `new`, запустите программу, чтобы увидеть выходные данные дополнительных операторов вызова. Отобразятся следующие результаты.

```
// Output:
// Base - Method1
// Base - Method2
// Base - Method1
// Derived - Method2
// Base - Method1
// Base - Method2
```

Ключевое слово `new` сохраняет связи, дающие этот результат, однако подавляет предупреждение. Переменные, имеющие тип `BaseClass`, продолжают обращаться к элементам класса `BaseClass`, а

переменная, имеющая тип `DerivedClass`, продолжает обращаться к элементам класса `DerivedClass` в первую очередь, а затем к элементам, унаследованным от `BaseClass`.

Чтобы подавить предупреждение, добавьте модификатор `new` в определение `Method2` в классе `DerivedClass`, как показано в следующем коде. Модификатор можно добавить перед или после `public`.

```
public new void Method2()
{
    Console.WriteLine("Derived - Method2");
}
```

Запустите программу еще раз, чтобы убедиться, что выходные данные не изменились. Также убедитесь, что предупреждение больше не появляется. Используя ключевое слово `new`, вы утверждаете, что вам известно, что модифицируемый им член скрывается членом, который наследуется от базового класса. Дополнительные сведения о скрытии имен через наследование см. в разделе Модификатор new.

Чтобы противопоставить это поведение эффекту использования `override`, добавьте в `DerivedClass` следующий метод. Модификатор `override` можно добавить перед или после `public`.

```
public override void Method1()
{
    Console.WriteLine("Derived - Method1");
}
```

Добавьте модификатор `virtual` в определение `Method1` в `BaseClass`. Модификатор `virtual` можно добавить перед или после `public`.

```
public virtual void Method1()
{
    Console.WriteLine("Base - Method1");
}
```

Снова запустите проект. Обратите особое внимание на последние две строки следующих выходных данных.

```
// Output:
// Base - Method1
// Base - Method2
// Derived - Method1
// Derived - Method2
// Derived - Method1
// Base - Method2
```

Использование модификатора `override` позволяет `bcdc` осуществлять доступ к методу `Method1`, который определен в `DerivedClass`. Как правило, это требуемое поведение в иерархиях наследования. Требуется, чтобы объекты со значениями, созданными из производного класса, использовали определенные в производном классе методы. Это поведение достигается с использованием `override` для расширения метода базового класса.

Следующий код содержит полный пример.

```
using System;
using System.Text;

namespace OverrideAndNew
{
    class Program
    {
        static void Main(string[] args)
        {
            BaseClass bc = new BaseClass();
            DerivedClass dc = new DerivedClass();
            BaseClass bcdc = new DerivedClass();

            // The following two calls do what you would expect. They call
            // the methods that are defined in BaseClass.
            bc.Method1();
            bc.Method2();
            // Output:
            // Base - Method1
            // Base - Method2

            // The following two calls do what you would expect. They call
            // the methods that are defined in DerivedClass.
            dc.Method1();
            dc.Method2();
            // Output:
            // Derived - Method1
            // Derived - Method2

            // The following two calls produce different results, depending
            // on whether override (Method1) or new (Method2) is used.
            bcdc.Method1();
            bcdc.Method2();
            // Output:
            // Derived - Method1
            // Base - Method2
        }
    }

    class BaseClass
    {
        public virtual void Method1()
        {
            Console.WriteLine("Base - Method1");
        }

        public virtual void Method2()
        {
            Console.WriteLine("Base - Method2");
        }
    }

    class DerivedClass : BaseClass
    {
        public override void Method1()
        {
            Console.WriteLine("Derived - Method1");
        }

        public new void Method2()
        {
            Console.WriteLine("Derived - Method2");
        }
    }
}
```

В следующем примере показано подобное поведение в другом контексте. В примере определяются три класса: базовый класс `Car` и два класса, производных от него: `ConvertibleCar` и `Minivan`. Базовый класс содержит метод `DescribeCar`. Этот метод отображает общее описание автомобиля, а затем вызывает `ShowDetails` для предоставления дополнительных сведений. Каждый из трех классов определяет метод `ShowDetails`. Для определения метода `ShowDetails` в классе `ConvertibleCar` используется модификатор `new`. Для определения метода `ShowDetails` в классе `Minivan` используется модификатор `override`.

```
// Define the base class, Car. The class defines two methods,
// DescribeCar and ShowDetails. DescribeCar calls ShowDetails, and each derived
// class also defines a ShowDetails method. The example tests which version of
// ShowDetails is selected, the base class method or the derived class method.
class Car
{
    public void DescribeCar()
    {
        System.Console.WriteLine("Four wheels and an engine.");
        ShowDetails();
    }

    public virtual void ShowDetails()
    {
        System.Console.WriteLine("Standard transportation.");
    }
}

// Define the derived classes.

// Class ConvertibleCar uses the new modifier to acknowledge that ShowDetails
// hides the base class method.
class ConvertibleCar : Car
{
    public new void ShowDetails()
    {
        System.Console.WriteLine("A roof that opens up.");
    }
}

// Class Minivan uses the override modifier to specify that ShowDetails
// extends the base class method.
class Minivan : Car
{
    public override void ShowDetails()
    {
        System.Console.WriteLine("Carries seven people.");
    }
}
```

В примере проверяется версия вызванного `ShowDetails`. Следующий метод, `TestCars1`, объявляет экземпляр каждого класса, а затем вызывает `DescribeCar` на каждом экземпляре.

```
public static void TestCars1()
{
    System.Console.WriteLine("\nTestCars1");
    System.Console.WriteLine("----------");

    Car car1 = new Car();
    car1.DescribeCar();
    System.Console.WriteLine("----------");

    // Notice the output from this test case. The new modifier is
    // used in the definition of ShowDetails in the ConvertibleCar
    // class.

    ConvertibleCar car2 = new ConvertibleCar();
    car2.DescribeCar();
    System.Console.WriteLine("----------");

    Minivan car3 = new Minivan();
    car3.DescribeCar();
    System.Console.WriteLine("----------");
}
```

`TestCars1` формирует следующие выходные данные. Обратите особое внимание на результаты для `car2`, который, вероятно, не соответствует ожидаемым. Тип объекта — `ConvertibleCar`, но `DescribeCar` не получает доступа к версии `ShowDetails`, определенной в классе `ConvertibleCar`, так как этот метод объявлен с модификатором `new`, а не `override`. В результате объект `ConvertibleCar` отображает то же описание, что и объект `Car`. Сравните результаты для `car3`, который является объектом `Minivan`. В этом случае метод `ShowDetails`, объявленный в классе `Minivan`, переопределяет метод `ShowDetails`, объявленный в классе `Car`, и отображается описание микроавтобуса.

```
// TestCars1
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Carries seven people.
// ----------
```

`TestCars2` создает список объектов, имеющих тип `Car`. Значения объектов создаются из классов `Car`, `ConvertibleCar` и `Minivan`. `DescribeCar` вызывается для каждого элемента списка. В следующем коде показано определение `TestCars2`.

```
public static void TestCars2()
{
    System.Console.WriteLine("\nTestCars2");
    System.Console.WriteLine("----------");

    var cars = new List<Car> { new Car(), new ConvertibleCar(),
        new Minivan() };

    foreach (var car in cars)
    {
        car.DescribeCar();
        System.Console.WriteLine("----------");
    }
}
```

Выводится следующий результат. Обратите внимание, что он совпадает с выходными данными, отображаемыми `TestCars1`. Метод `ShowDetails` класса `ConvertibleCar` не вызывается, независимо от того, является ли тип объекта `ConvertibleCar`, как в `TestCars1`, или `Car`, как в `TestCars2`. И наоборот, `car3` вызывает метод `ShowDetails` класса `Minivan` в обоих случаях, независимо от того, какого они типа — `Minivan` или `Car`.

```
// TestCars2
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Carries seven people.
// ----------
```

Методы `TestCars3` и `TestCars4` завершают пример. Эти методы вызывают `ShowDetails` напрямую: сначала из объектов, объявленных с типом `ConvertibleCar` и `Minivan` (`TestCars3`), а затем из объектов, объявленных с типом `Car` (`TestCars4`). Следующий код определяет эти два метода.

```
public static void TestCars3()
{
    System.Console.WriteLine("\nTestCars3");
    System.Console.WriteLine("----------");
    ConvertibleCar car2 = new ConvertibleCar();
    Minivan car3 = new Minivan();
    car2.ShowDetails();
    car3.ShowDetails();
}

public static void TestCars4()
{
    System.Console.WriteLine("\nTestCars4");
    System.Console.WriteLine("----------");
    Car car2 = new ConvertibleCar();
    Car car3 = new Minivan();
    car2.ShowDetails();
    car3.ShowDetails();
}
```

Методы производят следующий результат, который соответствует результатам из первого примера в этой теме.

```
// TestCars3
// ----------
// A roof that opens up.
// Carries seven people.

// TestCars4
// ----------
// Standard transportation.
// Carries seven people.
```

В следующем коде приведен полный проект и его результат.

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Text;

namespace OverrideAndNew2
{
    class Program
    {
        static void Main(string[] args)
        {
            // Declare objects of the derived classes and test which version
            // of ShowDetails is run, base or derived.
            TestCars1();

            // Declare objects of the base class, instantiated with the
            // derived classes, and repeat the tests.
            TestCars2();

            // Declare objects of the derived classes and call ShowDetails
            // directly.
            TestCars3();

            // Declare objects of the base class, instantiated with the
            // derived classes, and repeat the tests.
            TestCars4();
        }

        public static void TestCars1()
        {
            System.Console.WriteLine("\nTestCars1");
            System.Console.WriteLine("----------");

            Car car1 = new Car();
            car1.DescribeCar();
            System.Console.WriteLine("----------");

            // Notice the output from this test case. The new modifier is
            // used in the definition of ShowDetails in the ConvertibleCar
            // class.
            ConvertibleCar car2 = new ConvertibleCar();
            car2.DescribeCar();
            System.Console.WriteLine("----------");

            Minivan car3 = new Minivan();
            car3.DescribeCar();
            System.Console.WriteLine("----------");
        }
        // Output:
        // TestCars1
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Carries seven people.
        // ----------

        public static void TestCars2()
        {
            System.Console.WriteLine("\nTestCars2");
            System.Console.WriteLine("----------");

            var cars = new List<Car> { new Car(), new ConvertibleCar(),
                new Minivan() };

            foreach (var car in cars)
            {
                car.DescribeCar();
```

```
            car.DescribeCar();
            System.Console.WriteLine("----------");
        }
    }
    // Output:
    // TestCars2
    // ----------
    // Four wheels and an engine.
    // Standard transportation.
    // ----------
    // Four wheels and an engine.
    // Standard transportation.
    // ----------
    // Four wheels and an engine.
    // Carries seven people.
    // ----------

    public static void TestCars3()
    {
        System.Console.WriteLine("\nTestCars3");
        System.Console.WriteLine("----------");
        ConvertibleCar car2 = new ConvertibleCar();
        Minivan car3 = new Minivan();
        car2.ShowDetails();
        car3.ShowDetails();
    }
    // Output:
    // TestCars3
    // ----------
    // A roof that opens up.
    // Carries seven people.

    public static void TestCars4()
    {
        System.Console.WriteLine("\nTestCars4");
        System.Console.WriteLine("----------");
        Car car2 = new ConvertibleCar();
        Car car3 = new Minivan();
        car2.ShowDetails();
        car3.ShowDetails();
    }
    // Output:
    // TestCars4
    // ----------
    // Standard transportation.
    // Carries seven people.
}

// Define the base class, Car. The class defines two virtual methods,
// DescribeCar and ShowDetails. DescribeCar calls ShowDetails, and each derived
// class also defines a ShowDetails method. The example tests which version of
// ShowDetails is used, the base class method or the derived class method.
class Car
{
    public virtual void DescribeCar()
    {
        System.Console.WriteLine("Four wheels and an engine.");
        ShowDetails();
    }

    public virtual void ShowDetails()
    {
        System.Console.WriteLine("Standard transportation.");
    }
}

// Define the derived classes.

// Class ConvertibleCar uses the new modifier to acknowledge that ShowDetails
// hides the base class method.
```

```
    // hides the base class method.
    class ConvertibleCar : Car
    {
        public new void ShowDetails()
        {
            System.Console.WriteLine("A roof that opens up.");
        }
    }

    // Class Minivan uses the override modifier to specify that ShowDetails
    // extends the base class method.
    class Minivan : Car
    {
        public override void ShowDetails()
        {
            System.Console.WriteLine("Carries seven people.");
        }
    }

}
```

См. также

- Руководство по программированию на C#
- Система типов C#
- Управление версиями с помощью ключевых слов Override и New
- base
- abstract

Практическое руководство. Переопределение метода ToString (руководство по программированию на C#)

18.12.2021 • 2 minutes to read

Каждый класс или структура в языке C# неявно наследует класс Object. Поэтому каждый объект в языке C# получает метод ToString, который возвращает строковое представление данного объекта. Например, все переменные типа `int` имеют метод `ToString`, который позволяет им возвращать их содержимое в виде строки:

```
int x = 42;
string strx = x.ToString();
Console.WriteLine(strx);
// Output:
// 42
```

При создании пользовательского класса или структуры необходимо переопределить метод ToString, чтобы передать информацию о типе клиентскому коду.

Дополнительные сведения об использовании строк форматирования и других типов пользовательского форматирования с методом `ToString` см. в разделе Типы форматирования.

> **IMPORTANT**
> При принятии решения относительно того, какая информация должна будет предоставляться посредством этого метода, подумайте, будет ли создаваемый класс или структура когда-либо использоваться ненадежным кодом. Постарайтесь не предоставлять информацию, которая может быть использована вредоносным кодом.

Чтобы переопределить метод `ToString` в классе или структуре, выполните указанные ниже действия.

1. Объявите метод `ToString` со следующими модификаторами и типом возвращаемого значения:

   ```
   public override string ToString(){}
   ```

2. Реализуйте этот метод таким образом, чтобы он возвращал строку.

   В приведенном ниже примере возвращается не только имя класса, но и специфические данные для конкретного экземпляра класса.

   ```
   class Person
   {
       public string Name { get; set; }
       public int Age { get; set; }

       public override string ToString()
       {
           return "Person: " + Name + " " + Age;
       }
   }
   ```

Метод `ToString` можно проверить с помощью показанного ниже кода.

```
Person person = new Person { Name = "John", Age = 12 };
Console.WriteLine(person);
// Output:
// Person: John 12
```

См. также

- IFormattable
- Руководство по программированию на C#
- Система типов C#
- Строки
- string
- override
- virtual
- Типы форматирования

Члены (Руководство по программированию на C#)

18.12.2021 • 2 minutes to read

В классах и структурах есть члены, представляющие их данные и поведение. Члены класса включают все члены, объявленные в этом классе, а также все члены (кроме конструкторов и методов завершения), объявленные во всех классах в иерархии наследования данного класса. Закрытые члены в базовых классах наследуются, но недоступны из производных классов.

В следующей таблице перечислены виды членов, которые содержатся в классе или в структуре.

ЧЛЕН	ОПИСАНИЕ
Поля	Поля являются переменными, объявленными в области класса. Поле может иметь встроенный числовой тип или быть экземпляром другого класса. Например, в классе календаря может быть поле, содержащее текущую дату.
Константы	Константы — это поля, значения которых устанавливаются во время компиляции и не изменяются.
Свойства	Свойства — это методы класса. Доступ к ним осуществляется так же, как если бы они были полями этого класса. Свойство может защитить поле класса от изменений (независимо от объекта).
Методы	Методы определяют действия, которые может выполнить класс. Методы могут принимать параметры, предоставляющие входные данные, и возвращать выходные данные посредством параметров. Методы могут также возвращать значения напрямую, без использования параметров.
События	События предоставляют другим объектам уведомления о различных случаях, таких как нажатие кнопки или успешное выполнение метода. События определяются и переключаются с помощью делегатов.
Инструкции	Перегруженные операторы считаются членами типа. При перегрузке оператора его следует определять как открытый статический метод в типе. Для получения дополнительной информации см. раздел Перегрузка операторов.
Индексаторы	Индексаторы позволяют индексировать объекты аналогично массивам.
Конструкторы	Конструкторы — это методы, которые вызываются при создании объекта. Зачастую они используются для инициализации данных объекта.

ЧЛЕН	ОПИСАНИЕ
Методы завершения	Методы завершения очень редко используются в C#. Они являются методами, вызываемыми средой выполнения, когда объект нужно удалить из памяти. Они обычно применяются для правильной обработки ресурсов, которые должны быть высвобождены.
Вложенные типы	Вложенными типами являются типы, объявленные в другом типе. Вложенные типы часто применяются для описания объектов, использующихся только типами, в которых эти объекты находятся.

См. также

- Руководство по программированию на C#
- Классы

Абстрактные и запечатанные классы и члены классов (Руководство по программированию на C#)

18.12.2021 • 2 minutes to read

Ключевое слово abstract позволяет создавать классы и члены классов, которые являются неполными и должны быть реализованы в производном классе.

Ключевое слово sealed позволяет предотвратить наследование класса или определенных членов класса, помеченных ранее как virtual.

Абстрактные классы и члены классов

Классы могут быть объявлены абстрактными путем помещения ключевого слова `abstract` перед определением класса. Пример:

```
public abstract class A
{
    // Class members here.
}
```

Создавать экземпляры абстрактного класса нельзя. Назначение абстрактного класса заключается в предоставлении общего определения для базового класса, которое могут совместно использовать несколько производных классов. Например, в библиотеке классов может быть определен абстрактный класс, используемый в качестве параметра для многих из ее функций, поэтому программисты, использующие эту библиотеку, должны задать свою реализацию этого класса, создав производный класс.

Абстрактные классы могут определять абстрактные методы. Для этого перед типом возвращаемого значения метода необходимо поместить ключевое слово `abstract`. Пример:

```
public abstract class A
{
    public abstract void DoWork(int i);
}
```

Абстрактные методы не имеют реализации, поэтому определение такого метода заканчивается точкой с запятой вместо обычного блока метода. Классы, производные от абстрактного класса, должны реализовывать все абстрактные методы. Если абстрактный класс наследует виртуальный метод из базового класса, абстрактный класс может переопределить виртуальный метод с помощью абстрактного метода. Пример:

```
// compile with: -target:library
public class D
{
    public virtual void DoWork(int i)
    {
        // Original implementation.
    }
}

public abstract class E : D
{
    public abstract override void DoWork(int i);
}

public class F : E
{
    public override void DoWork(int i)
    {
        // New implementation.
    }
}
```

Если метод `virtual` объявляется как `abstract`, он все равно считается виртуальным по отношению к любому классу, наследующему от абстрактного класса. Класс, наследующий абстрактный метод, не может получать доступ к исходной реализации метода (так, в предыдущем примере `DoWork` для класса F не может вызывать `DoWork` для класса D). Таким образом, абстрактный класс может принудительно задавать необходимость предоставлять новые реализации виртуальных методов в производных классах.

Запечатанные классы и члены классов

Классы могут быть объявлены как запечатанные путем помещения ключевого слова `sealed` перед определением класса. Пример:

```
public sealed class D
{
    // Class members here.
}
```

Запечатанный класс не может использоваться в качестве базового класса. Поэтому он также не может быть абстрактным классом. Запечатанные классы предотвращают наследование. Поскольку их нельзя использовать в качестве базовых классов, определенная оптимизация во время выполнения позволяет несколько ускорить вызов членов запечатанных классов.

Метод, индексатор, свойство или событие для производного класса, переопределяющего виртуальный член базового класса, может объявлять этот член как запечатанный. Это делает бесполезным виртуальный аспект члена для каждого последующего производного класса. Для этого в объявлении члена класса необходимо перед ключевым словом override поместить ключевое слово `sealed`. Пример:

```
public class D : C
{
    public sealed override void DoWork() { }
}
```

См. также

- Руководство по программированию на C#

- Система типов C#
- Наследование
- Методы
- Поля
- Определение абстрактных свойств

Статические классы и члены статических классов (Руководство по программированию в C#)

18.12.2021 • 4 minutes to read

Статический класс в основном такой же, как и нестатический класс, но имеется одно отличие: нельзя создавать экземпляры статического класса. Другими словами, нельзя использовать оператор new для создания переменной типа класса. Поскольку нет переменной экземпляра, доступ к членам статического класса осуществляется с использованием самого имени класса. Например, если есть статический класс, называемый `UtilityClass`, имеющий открытый статический метод с именем `MethodA`, вызов метода выполняется, как показано в следующем примере:

```
UtilityClass.MethodA();
```

Статический класс может использоваться как обычный контейнер для наборов методов, работающих на входных параметрах, и не должен возвращать или устанавливать каких-либо внутренних полей экземпляра. Например, в библиотеке классов .NET статический класс System.Math содержит методы, выполняющие математические операции, без требования сохранять или извлекать данные, уникальные для конкретного экземпляра класса Math. Это значит, что члены класса применяются путем задания имени класса и имени метода, как показано в следующем примере.

```
double dub = -3.14;
Console.WriteLine(Math.Abs(dub));
Console.WriteLine(Math.Floor(dub));
Console.WriteLine(Math.Round(Math.Abs(dub)));

// Output:
// 3.14
// -4
// 3
```

Как и в случае с типами всех классов, сведения о типе для статического класса загружаются средой выполнения .NET, когда загружается программа, которая ссылается на класс. Программа не может точно указать, когда загружается класс. Однако гарантируется загрузка этого класса, инициализация его полей и вызов статического конструктора перед первым обращением к классу в программе. Статический конструктор вызывается только один раз, и статический класс остается в памяти на время существования домена приложения, в котором находится программа.

> **NOTE**
> Создание нестатического класса, который допускает создание только одного экземпляра самого себя, см. в документе Реализация Singleton в C#.

Ниже приведены основные возможности статического класса.

- Содержит только статические члены.
- Создавать его экземпляры нельзя.
- Является запечатанным.

- Не может содержать конструкторы экземпляров.

По сути, создание статического класса аналогично созданию класса, содержащего только статические члены и закрытый конструктор. Закрытый конструктор не допускает создания экземпляров класса. Преимущество применения статических классов заключается в том, что компилятор может проверить отсутствие случайно добавленных членов экземпляров. Таким образом, компилятор гарантирует невозможность создания экземпляров таких классов.

Статические классы запечатаны, поэтому их нельзя наследовать. Они не могут наследовать ни от каких классов, кроме Object. Статические классы не могут содержать конструктор экземпляров. Однако они могут содержать статический конструктор. Нестатические классы также должен определять статический конструктор, если класс содержит статические члены, для которых нужна нетривиальная инициализация. Дополнительные сведения см. в разделе Статические конструкторы.

Пример

Ниже приведен пример статического класса, содержащего два метода, преобразующих температуру по Цельсию в температуру по Фаренгейту и наоборот.

```
public static class TemperatureConverter
{
    public static double CelsiusToFahrenheit(string temperatureCelsius)
    {
        // Convert argument to double for calculations.
        double celsius = Double.Parse(temperatureCelsius);

        // Convert Celsius to Fahrenheit.
        double fahrenheit = (celsius * 9 / 5) + 32;

        return fahrenheit;
    }

    public static double FahrenheitToCelsius(string temperatureFahrenheit)
    {
        // Convert argument to double for calculations.
        double fahrenheit = Double.Parse(temperatureFahrenheit);

        // Convert Fahrenheit to Celsius.
        double celsius = (fahrenheit - 32) * 5 / 9;

        return celsius;
    }
}

class TestTemperatureConverter
{
    static void Main()
    {
        Console.WriteLine("Please select the convertor direction");
        Console.WriteLine("1. From Celsius to Fahrenheit.");
        Console.WriteLine("2. From Fahrenheit to Celsius.");
        Console.Write(":");

        string selection = Console.ReadLine();
        double F, C = 0;

        switch (selection)
        {
            case "1":
                Console.Write("Please enter the Celsius temperature: ");
                F = TemperatureConverter.CelsiusToFahrenheit(Console.ReadLine());
                Console.WriteLine("Temperature in Fahrenheit: {0:F2}", F);
                break;
```

```
            case "2":
                Console.Write("Please enter the Fahrenheit temperature: ");
                C = TemperatureConverter.FahrenheitToCelsius(Console.ReadLine());
                Console.WriteLine("Temperature in Celsius: {0:F2}", C);
                break;

            default:
                Console.WriteLine("Please select a convertor.");
                break;
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Example Output:
    Please select the convertor direction
    1. From Celsius to Fahrenheit.
    2. From Fahrenheit to Celsius.
    :2
    Please enter the Fahrenheit temperature: 20
    Temperature in Celsius: -6.67
    Press any key to exit.
 */
```

Статический члены

Нестатический класс может содержать статические методы, поля, свойства или события. Статический член вызывается для класса даже в том случае, если не создан экземпляр класса. Доступ к статическому члены всегда выполняется по имени класса, а не экземпляра. Существует только одна копия статического члена, независимо от того, сколько создано экземпляров класса. Статические методы и свойства не могут обращаться к нестатическим полям и событиям в их содержащем типе, и они не могут обращаться к переменной экземпляра объекта, если он не передается явно в параметре метода.

Более привычно объявление нестатического класса с несколькими статическими членами, чем объявление всего класса как статического. Статические поля обычно используются для следующих двух целей: хранение счетчика числа созданных объектов или хранение значения, которое должно совместно использоваться всеми экземплярами.

Статические методы могут быть перегружены, но не переопределены, поскольку они относятся к классу, а не к экземпляру класса.

Несмотря на то, что поле не может быть объявлено как `static const`, поле const по своему поведению является статическим. Он относится к типу, а не к экземплярам типа. Поэтому к полям `const` можно обращаться с использованием той же нотации `ClassName.MemberName`, что и для статических полей. Экземпляр объекта не требуется.

C# не поддерживает статические локальные переменные (то есть переменные, объявленные в области действия метода).

Для объявления статических методов класса используется ключевое слово `static` перед возвращаемым типом члена, как показано в следующем примере:

```
public class Automobile
{
    public static int NumberOfWheels = 4;

    public static int SizeOfGasTank
    {
        get
        {
            return 15;
        }
    }

    public static void Drive() { }

    public static event EventType RunOutOfGas;

    // Other non-static fields and properties...
}
```

Статические члены инициализируются перед первым доступом к статическому члену и перед вызовом статического конструктора, если таковой имеется. Для доступа к члену статического класса следует использовать имя класса, а не имя переменной, указывая расположение члена, как показано в следующем примере:

```
Automobile.Drive();
int i = Automobile.NumberOfWheels;
```

Если класс содержит статические поля, должен быть указан статический конструктор, который инициализирует эти поля при загрузке класса.

Вызов статического метода генерирует инструкцию вызова в промежуточном языке Microsoft (MSIL), в то время как вызов метода экземпляра генерирует инструкцию `callvirt`, которая также проверяет наличие ссылок на пустые объекты. Однако в большинстве случаев разница в производительности двух видов вызовов несущественна.

Спецификация языка C#

Дополнительные сведения см. в разделах Статические классы и Члены экземпляра и статические члены в документации Предварительная спецификация C# 6.0. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Руководство по программированию на C#
- static
- Классы
- class
- Статические конструкторы
- Конструкторы экземпляров

Модификаторы доступа (Руководство по программированию в C#)

18.12.2021 • 4 minutes to read

Все типы и члены типов имеют уровень доступности. Он определяет возможность их использования из другого кода в вашей или в других сборках. Следующие модификаторы доступа позволяют указать доступность типа или члена при объявлении:

- public: Доступ к типу или члену возможен из любого другого кода в той же сборке или другой сборке, ссылающейся на него. Уровень доступности общедоступных элементов типа определяется уровнем доступности самого типа.
- private: доступ к типу или члену возможен только из кода в том же объекте `class` или `struct`.
- protected: доступ к типу или члену возможен только из кода в том же объекте `class` либо в `class`, производном от этого `class`.
- internal: Доступ к типу или члену возможен из любого кода в той же сборке, но не из другой сборки.
- protected internal: доступ к типу или члену возможен из любого кода в той сборке, где он был объявлен, или из производного `class` в другой сборке.
- private protected: доступ к типу или элементу возможен из типов, производных от объекта `class` и объявляемых в сборке, содержащей этот объект.

Сводная таблица

РАСПОЛОЖЕНИЕ ВЫЗЫВАЮЩЕГО ОБЪЕКТА	`PUBLIC`	`PROTECTED INTERNAL`	`PROTECTED`	`INTERNAL`	`PRIVATE PROTECTED`	`PRIVATE`
В классе	✔	✔	✔	✔	✔	✔
Производный класс (та же сборка)	✔	✔	✔	✔	✔	✘
Непроизводный класс (та же сборка)	✔	✔	✘	✔	✘	✘
Производный класс (другая сборка)	✔	✔	✔	✘	✘	✘
Непроизводный класс (другая сборка)	✔	✘	✘	✘	✘	✘

В следующих примерах показано, как изменить модификаторы доступа для типа или члена типа:

```
public class Bicycle
{
    public void Pedal() { }
}
```

Не все модификаторы доступа подходят для всех типов или членов во всех контекстах. В некоторых случаях доступность члена типа ограничивается доступностью содержащего его типа.

Доступность классов, записей и структур

Классы, записи и структуры, объявленные непосредственно в пространстве имен (другими словами, не вложенные в другие классы или структуры), могут быть `public` или `internal`. Если модификатор доступа не указан, по умолчанию используется `internal`.

Члены структуры, включая вложенные классы и структуры, могут быть объявлены как `public`, `internal` или `private`. Члены класса, включая вложенные классы и структуры, могут быть `public`, `protected internal`, `protected`, `internal`, `private protected` или `private`. Члены класса и структуры, включая вложенные классы и структуры, по умолчанию имеют доступ `private`. Закрытые вложенные типы недоступны за пределами типа, в котором содержатся.

Производные классы записи не могут быть более доступными, чем соответствующие базовые типы. Невозможно объявить открытый класс `B`, производный от внутреннего класса `A`. Если бы это было возможно, класс `A` стал бы открытым, так как все члены `protected` или `internal` класса `A` были бы доступны из производного класса.

Доступ к внутренним типам можно предоставить некоторым другим сборкам с помощью `InternalsVisibleToAttribute`. Дополнительные сведения см. в разделе Дружественные сборки.

Доступность членов классов, записей и структур

Члены классов и записей (включая вложенные классы, записи и структуры) можно объявлять с любым из шести типов доступа. Члены структуры не могут быть объявлены как `protected`, `protected internal` или `private protected`, так как структуры не поддерживают наследование.

Как правило, уровень доступности члена не может быть выше уровня доступности типа, в который он входит. При этом член `public` внутреннего класса может быть доступен за пределами сборки, если он реализует методы интерфейса или переопределяет виртуальные методы, определенные в открытом базовом классе.

Тип любого поля, свойства или события члена должен иметь по меньшей мере такой же уровень доступности, как у самого члена. Точно так же тип возвращаемого значения и типы параметров любого метода, индексатора или делегата должен иметь по меньшей мере такой же уровень доступности, как у самого члена. Например, невозможно иметь метод `public M`, возвращающий класс `C`, если только `C` также не является `public`. Аналогичным образом, невозможно иметь свойство `protected` типа `A`, если `A` объявлен как `private`.

Пользовательские операторы всегда должны объявляться как `public` и `static`. Для получения дополнительной информации см. раздел Перегрузка операторов.

Методы завершения не могут иметь модификаторы доступа.

Чтобы настроить уровень доступа для члена `class`, `record` или `struct`, добавьте в объявление этого члена соответствующее ключевое слово, как показано в следующем примере.

```
// public class:
public class Tricycle
{
    // protected method:
    protected void Pedal() { }

    // private field:
    private int wheels = 3;

    // protected internal property:
    protected internal int Wheels
    {
        get { return wheels; }
    }
}
```

Другие типы

Интерфейсы, объявляемые непосредственно в пространстве имен, могут быть `public` или `internal`. Как и в случае с классами и структурами, для интерфейсов по умолчанию задается доступ `internal`. Члены интерфейса по умолчанию являются `public`, так как интерфейс как раз и создан для того, чтобы обеспечить другим типам доступ к классу или структуре. Объявления членов интерфейса могут включать любой модификатор доступа. Это удобнее всего использовать для статических методов, чтобы предоставить общие реализации, необходимые всем объектам, реализующим класс.

Члены перечисления всегда являются `public`, и модификаторы доступа к ним не применяются.

Делегаты ведут себя как классы и структуры. По умолчанию они имеют доступ `internal`, если объявляются непосредственно в пространстве имен, и доступ `private`, если являются вложенными.

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Руководство по программированию на C#
- Система типов C#
- Интерфейсы
- private
- public
- internal
- protected
- protected internal
- private protected
- class
- struct
- interface

Поля (Руководство по программированию в C#)

18.12.2021 • 3 minutes to read

Поле является переменной любого типа, которая объявлена непосредственно в классе или структуре. Поля являются *членами* содержащих их типов.

Класс или структура может иметь поля экземпляра и (или) статические поля. Поля экземпляра относятся только к экземпляру типа. Если имеется класс T с полем экземпляра F, можно создать два объекта типа T и изменить значение F в каждом объекте, не влияя на значение в другом объекте. Напротив, статическое поле принадлежит самому типу и является общим для всех экземпляров этого типа. Доступ к статическому полю можно получить только с помощью имени типа. Если вы обращаетесь к статическому полю по имени экземпляра, вы получите ошибки времени компиляции CS0176.

Как правило, следует использовать поля только для переменных, являющихся закрытыми или защищенными. Данные, которые тип представляет клиентскому коду, должны предоставляться через методы, свойства и индексаторы. Используя эти конструкции для косвенного доступа к внутренним полям, можно предотвратить использование недопустимых входных значений. Закрытое поле, которое хранит данные, представленные открытым свойством, называется *резервным хранилищем* или *резервным полем*.

Как правило, поля хранят данные, которые должны быть доступны нескольким методам типа и храниться дольше, чем время существования любого отдельного метода. Например, тип, представляющий календарную дату, может иметь три целочисленных поля: одно для месяца, одно для числа и одно для года. Переменные, не используемые вне области одного метода, должны быть объявлены как *локальные переменные* в самом теле метода.

Поля объявляются в блоке класса или структуры путем указания уровня доступа поля, за которым следует тип поля, а затем имя поля. Пример:

```
public class CalendarEntry
{

    // private field (Located near wrapping "Date" property).
    private DateTime _date;

    // Public property exposes _date field safely.
    public DateTime Date
    {
        get
        {
            return _date;
        }
        set
        {
            // Set some reasonable boundaries for likely birth dates.
            if (value.Year > 1900 && value.Year <= DateTime.Today.Year)
            {
                _date = value;
            }
            else
            {
                throw new ArgumentOutOfRangeException();
            }
        }
    }

    // public field (Generally not recommended).
    public string Day;

    // Public method also exposes _date field safely.
    // Example call: birthday.SetDate("1975, 6, 30");
    public void SetDate(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        // Set some reasonable boundaries for likely birth dates.
        if (dt.Year > 1900 && dt.Year <= DateTime.Today.Year)
        {
            _date = dt;
        }
        else
        {
            throw new ArgumentOutOfRangeException();
        }
    }

    public TimeSpan GetTimeSpan(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        if (dt.Ticks < _date.Ticks)
        {
            return _date - dt;
        }
        else
        {
            throw new ArgumentOutOfRangeException();
        }
    }
}
```

Для доступа к полю в экземпляре добавьте точку после имени экземпляра, за которой следует имя поля, как в `instancename._fieldName`. Пример:

```
CalendarEntry birthday = new CalendarEntry();
birthday.Day = "Saturday";
```

Полю можно присвоить начальное значение с помощью оператора присваивания при объявлении поля. Чтобы автоматически назначить поле `Day`, например полю `"Monday"`, нужно объявить `Day`, как в следующем примере:

```
public class CalendarDateWithInitialization
{
    public string Day = "Monday";
    //...
}
```

Поля инициализируются непосредственно перед вызовом конструктора для экземпляра объекта. Если конструктор присваивает значение поля, он заменит значения, присвоенные при объявлении поля. Дополнительные сведения см. в разделе Использование конструкторов.

> **NOTE**
> Инициализатор поля не может ссылаться на другие поля экземпляров.

Поля могут иметь пометку public, private, protected, internal, protected internal или private protected. Эти модификаторы доступа определяют, каким образом пользователи типа смогут получать доступ к полю. Дополнительные сведения см. в статье Модификаторы доступа.

При необходимости можно объявить поле статическим (static). Это делает поле доступным для вызывающих объектов в любое время, даже если экземпляр типа не существует. Дополнительные сведения см. в статье Статические классы и члены статических классов.

Поле может быть объявлено доступным только для чтения (readonly). Полю только для чтения можно присвоить значение только во время инициализации или в конструкторе. Поле `static readonly` очень похоже на константу, за исключением того, что компилятор C# не имеет доступа к значению статического поля только для чтения во время компиляции, но только во время выполнения. Дополнительные сведения см. в разделе Константы.

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Руководство по программированию на C#
- Система типов C#
- Использование конструкторов
- Наследование
- Модификаторы доступа
- Абстрактные и запечатанные классы и члены классов

Константы (Руководство по программированию на C#)

18.12.2021 • 2 minutes to read

Константы — это постоянные значения, которые известны во время компиляции и не изменяются во время выполнения программы. Константы должны объявляться с модификатором const. Только встроенные типы C# (за исключением System.Object) могут быть объявлены как `const`. Пользовательские типы, включая классы, структуры и массивы, не могут объявляться как `const`. Модификатор readonly позволяет создать класс, структуру или массив, которые инициализируются один раз (например, в конструкторе), и впоследствии изменить их нельзя.

C# не поддерживает методы, свойства или события `const`.

Тип перечисления позволяет определять именованные константы для целочисленных встроенных типов (например `int`, `uint`, `long` и т. д.). Дополнительные сведения см. в разделе Перечисление.

Константы должны инициализироваться сразу после объявления. Пример:

```
class Calendar1
{
    public const int Months = 12;
}
```

В этом примере константа `Months` всегда имеет значение 12, и его не может изменить даже сам класс. На самом деле в случае, если компилятор встречает идентификатор константы в исходном коде C# (например, `Months`), он подставляет значение литерала непосредственно в создаваемый им промежуточный язык (IL). Поскольку с константой в среде выполнения не связан адрес ни одной переменной, поля `const` не могут передаваться по ссылке и отображаться в выражении как левостороннее значение.

> **NOTE**
>
> Будьте внимательны, ссылаясь на постоянные значения, определенные в другом коде, например, в DLL. Если в новой версии DLL для константы определяется новое значение, старое значение литерала хранится в вашей программе вплоть до повторной компиляции для новой версии.

Несколько констант одного типа можно объявить одновременно, например:

```
class Calendar2
{
    public const int Months = 12, Weeks = 52, Days = 365;
}
```

Выражение, которое используется для инициализации константы, может ссылаться на другую константу, если не создает циклическую ссылку. Пример:

```
class Calendar3
{
    public const int Months = 12;
    public const int Weeks = 52;
    public const int Days = 365;

    public const double DaysPerWeek = (double) Days / (double) Weeks;
    public const double DaysPerMonth = (double) Days / (double) Months;
}
```

Константы могут иметь пометку public, private, protected, internal, protected internal или private protected. Эти модификаторы доступа определяют, каким образом пользователи класса смогут получать доступ к константе. Дополнительные сведения см. в разделе Модификаторы доступа.

Доступ к константам осуществляется так, как если бы они были статическими полями, поскольку значение константы одинаково для всех экземпляров типа. Для их объявления используйте ключевое слово `static`. Выражения, которые не относятся к классу, определяющему константу, должны включать имя класса, период и имя константы для доступа к этой константе. Пример:

```
int birthstones = Calendar.Months;
```

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Руководство по программированию на C#
- Свойства
- Типы
- readonly
- Immutability in C# Part One: Kinds of Immutability (Неизменность в C#, часть 1. Виды неизменности)

Практическое руководство. Определение абстрактных свойств (руководство по программированию на C#)

18.12.2021 • 2 minutes to read

В следующем примере показано, как определять абстрактные свойства. В объявлении абстрактного свойства не предоставляется реализация методов доступа к свойству. В нем объявляется, что класс поддерживает свойства, однако реализация методов доступа к ним передается в производные классы. В следующем примере показано, как реализовать абстрактные свойства, наследуемые от базового класса.

Этот пример включает три файла, каждый из которых компилируется отдельно в сборку, на которую задаются ссылки при последующей компиляции:

- abstractshape.cs: класс `Shape`, который содержит абстрактное свойство `Area`.
- shapes.cs: подклассы класса `Shape`.
- shapetest.cs: тестовая программа для отображения областей некоторых объектов, производных от `Shape`.

Чтобы скомпилировать этот пример, используйте следующую команду:

```
csc abstractshape.cs shapes.cs shapetest.cs
```

При этом будет создан исполняемый файл shapetest.exe.

Примеры

В этом файле объявляется класс `Shape`, который содержит свойство `Area` типа `double`.

```
// compile with: csc -target:library abstractshape.cs
public abstract class Shape
{
    private string name;

    public Shape(string s)
    {
        // calling the set accessor of the Id property.
        Id = s;
    }

    public string Id
    {
        get
        {
            return name;
        }

        set
        {
            name = value;
        }
    }

    // Area is a read-only property - only a get accessor is needed:
    public abstract double Area
    {
        get;
    }

    public override string ToString()
    {
        return $"{Id} Area = {Area:F2}";
    }
}
```

- Модификаторы свойства помещаются в само объявление свойства. Пример:

  ```
  public abstract double Area
  ```

- При объявлении абстрактного свойства, такого как `Area` в этом примере, вы просто указываете используемые методы доступа, но не реализуете их. В этом примере используется только метод доступа get, поэтому свойство будет доступно только для чтения.

В следующем коде представлены три подкласса класса `Shape` и демонстрируется, как они переопределяют свойство `Area` для получения его собственной реализации.

```
// compile with: csc -target:library -reference:abstractshape.dll shapes.cs
public class Square : Shape
{
    private int side;

    public Square(int side, string id)
        : base(id)
    {
        this.side = side;
    }

    public override double Area
    {
        get
        {
            // Given the side, return the area of a square:
            return side * side;
        }
    }
}

public class Circle : Shape
{
    private int radius;

    public Circle(int radius, string id)
        : base(id)
    {
        this.radius = radius;
    }

    public override double Area
    {
        get
        {
            // Given the radius, return the area of a circle:
            return radius * radius * System.Math.PI;
        }
    }
}

public class Rectangle : Shape
{
    private int width;
    private int height;

    public Rectangle(int width, int height, string id)
        : base(id)
    {
        this.width = width;
        this.height = height;
    }

    public override double Area
    {
        get
        {
            // Given the width and height, return the area of a rectangle:
            return width * height;
        }
    }
}
```

В следующем коде показана тестовая программа, которая создает несколько производных от `Shape` объектов и печатает их области.

```
// compile with: csc -reference:abstractshape.dll;shapes.dll shapetest.cs
class TestClass
{
    static void Main()
    {
        Shape[] shapes =
        {
            new Square(5, "Square #1"),
            new Circle(3, "Circle #1"),
            new Rectangle( 4, 5, "Rectangle #1")
        };

        System.Console.WriteLine("Shapes Collection");
        foreach (Shape s in shapes)
        {
            System.Console.WriteLine(s);
        }
    }
}
/* Output:
    Shapes Collection
    Square #1 Area = 25.00
    Circle #1 Area = 28.27
    Rectangle #1 Area = 20.00
*/
```

См. также

- Руководство по программированию на C#
- Система типов C#
- Абстрактные и запечатанные классы и члены классов
- Свойства

Определение констант в C#

18.12.2021 • 2 minutes to read

Константы — это поля, значения которых устанавливаются во время компиляции и не изменяются. С помощью констант можно присвоить особым значениям значащие имена вместо числовых литералов.

> **NOTE**
> В C# с помощью директивы препроцессора #define нельзя определять константы так, как это было реализовано в C и C++.

Чтобы определить значения константы целочисленного типа (`int` , `byte` и т. д.), используйте перечисляемый тип. Дополнительные сведения см. в разделе Перечисление.

Чтобы определить нецелочисленные константы, можно сгруппировать их в статический класс с именем `Constants` . В этом случае перед любыми ссылками на константы будет необходимо указывать имя класса, как показано в следующем примере.

Пример

```
using System;

static class Constants
{
    public const double Pi = 3.14159;
    public const int SpeedOfLight = 300000; // km per sec.
}

class Program
{
    static void Main()
    {
        double radius = 5.3;
        double area = Constants.Pi * (radius * radius);
        int secsFromSun = 149476000 / Constants.SpeedOfLight; // in km
        Console.WriteLine(secsFromSun);
    }
}
```

Используя квалификатор имени класса, вы гарантируете, что вы сами и другие разработчики будете понимать, что имеете дело с константой, которую нельзя изменить.

См. также

- Система типов C#

Свойства (Руководство по программированию в C#)

18.12.2021 • 4 minutes to read

Свойство — это член, предоставляющий гибкий механизм для чтения, записи или вычисления значения частного поля. Свойства можно использовать, как если бы они были членами общих данных, но фактически они представляют собой специальные методы, называемые *методами доступа*. Это позволяет легко получать доступ к данным и помогает повысить безопасность и гибкость методов.

Общие сведения о свойствах

- Свойства позволяют классу предоставлять общий способ получения и задания значений, скрывая при этом код реализации или проверки.
- Метод доступа get используется для возврата значения свойства, а метод доступа set — для присвоения нового значения. В C# 9 и более поздних версий метод доступа к свойству init используется для назначения нового значения только при создании объекта. Эти методы доступа могут иметь различные уровни доступа. Дополнительные сведения см. в разделе Доступность методов доступа.
- Ключевое слово value используется для определения значения, присваиваемого методом доступа `set` или `init`.
- Свойства могут быть доступны для *чтения и записи* (они имеют оба метода доступа — `get` и `set`), *только для чтения* (они имеют метод доступа `get`, но не имеют метода доступа `set`) или *только для записи* (они имеют метод доступа `set`, но не имеют метода доступа `get`). Свойства только для записи встречаются редко и чаще всего используются для ограничения доступа к конфиденциальным данным.
- Простые свойства, не требующие пользовательского кода метода доступа, можно реализовать как определения текста выражений или как автоматически реализуемые свойства.

Свойства с резервными полями

Одной из базовых схем реализации свойств является использование закрытого резервного поля для установки и извлечения значения свойства. Метод доступа `get` возвращает значение закрытого поля, а метод доступа `set` может выполнять определенные проверки данных до присвоения значению закрытого поля. Оба метода доступа также могут выполнять некоторые преобразования или вычисления данных до их сохранения или возвращения.

Это показано в следующем примере. В этом примере класс `TimePeriod` представляет интервал времени. На внутреннем уровне класс сохраняет интервал времени в секундах в закрытом поле с именем `_seconds`. Свойство чтения и записи с именем `Hours` позволяет клиенту указывать временной интервал в часах. Методы доступа `get` и `set` выполняют необходимое преобразование между часами и секундами. Кроме того, метод доступа `set` проверяет данные и создает ArgumentOutOfRangeException, если указано недопустимое количество часов.

```
using System;

class TimePeriod
{
   private double _seconds;

   public double Hours
   {
       get { return _seconds / 3600; }
       set {
          if (value < 0 || value > 24)
             throw new ArgumentOutOfRangeException(
                   $"{nameof(value)} must be between 0 and 24.");

          _seconds = value * 3600;
       }
   }
}

class Program
{
   static void Main()
   {
       TimePeriod t = new TimePeriod();
       // The property assignment causes the 'set' accessor to be called.
       t.Hours = 24;

       // Retrieving the property causes the 'get' accessor to be called.
       Console.WriteLine($"Time in hours: {t.Hours}");
   }
}
// The example displays the following output:
//    Time in hours: 24
```

Определения текста выражений

Как правило, методы доступа к свойствам состоят из однострочных операторов, которые просто назначают или возвращают результат выражения. Эти свойства можно реализовать как члены, воплощающие выражение. Определения текста выражений состоят из символа `=>`, за которым идет выражение, назначаемое свойству или извлекаемое из него.

Начиная с версии C# 6 свойства только для чтения могут реализовывать метод доступа `get` как член, воплощающий выражение. В этом случае не используется ни ключевое слово метода доступа `get`, ни ключевое слово `return`. В следующем примере показана реализация свойства только для чтения `Name` в виде члена, воплощающего выражение.

```
using System;

public class Person
{
   private string _firstName;
   private string _lastName;

   public Person(string first, string last)
   {
      _firstName = first;
      _lastName = last;
   }

   public string Name => $"{_firstName} {_lastName}";
}

public class Example
{
   public static void Main()
   {
      var person = new Person("Magnus", "Hedlund");
      Console.WriteLine(person.Name);
   }
}
// The example displays the following output:
//      Magnus Hedlund
```

Начиная с C# 7.0 методы доступа `get` и `set` можно реализовывать в виде членов, воплощающих выражения. В этом случае необходимо указывать ключевые слова `get` и `set`. В следующем примере показано использование определений текста выражений для обоих методов доступа. Обратите внимание, что ключевое слово `return` не используется с методом доступа `get`.

```
using System;

public class SaleItem
{
   string _name;
   decimal _cost;

   public SaleItem(string name, decimal cost)
   {
      _name = name;
      _cost = cost;
   }

   public string Name
   {
      get => _name;
      set => _name = value;
   }

   public decimal Price
   {
      get => _cost;
      set => _cost = value;
   }
}

class Program
{
   static void Main(string[] args)
   {
      var item = new SaleItem("Shoes", 19.95m);
      Console.WriteLine($"{item.Name}: sells for {item.Price:C2}");
   }
}
// The example displays output like the following:
//       Shoes: sells for $19.95
```

Автоматически реализуемые свойства

В некоторых случаях свойство `get` и методы доступа `set` просто присваивают значение резервному полю или извлекают значение из него без включения дополнительной логики. С помощью автоматически реализуемых свойств можно упростить код, в то время как компилятор C# будет прозрачно предоставлять вам резервное поле.

Если у свойства есть методы доступа `get` и `set` (или `get` и `init`), они оба должны быть автоматически реализованы. Автоматически реализуемое свойство определяется с помощью ключевых слов `get` и `set` без указания какой-либо реализации. Следующий пример аналогичен предыдущему, за исключением того, что `Name` и `Price` являются автоматически реализуемыми свойствами. В этом примере также удаляется параметризованный конструктор, что позволяет инициализировать объекты `SaleItem`, вызывая конструктор без параметров и инициализатор объекта.

```
using System;

public class SaleItem
{
   public string Name
   { get; set; }

   public decimal Price
   { get; set; }
}

class Program
{
   static void Main(string[] args)
   {
      var item = new SaleItem{ Name = "Shoes", Price = 19.95m };
      Console.WriteLine($"{item.Name}: sells for {item.Price:C2}");
   }
}
// The example displays output like the following:
//       Shoes: sells for $19.95
```

Связанные разделы

- Использование свойств
- Свойства интерфейса
- Сравнение свойств и индексаторов
- Ограничение доступности методов доступа
- Автоматически реализуемые свойства

Спецификация языка C#

Дополнительные сведения см. в разделе Свойства в Спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Руководство по программированию на C#
- Использование свойств
- Индексаторы
- Ключевое слово get
- Ключевое слово set

Использование свойств (Руководство по программированию в C#)

18.12.2021 • 7 minutes to read

Свойства сочетают в себе возможности полей и методов. Пользователю объекта свойство представляется как поле, и для доступа к нему применяется тот же синтаксис. При реализации класса свойство представляется в виде одного или двух блоков кода для методов доступа get и (или) set. Блок кода для метода доступа `get` выполняется только при считывании свойства, а для метода `set` — при присвоении свойству нового значения. Свойство без метода доступа `set` доступно только для чтения. Свойство без метода доступа `get` доступно только для записи. Свойство, для которого определены оба этих метода, доступно для чтения и записи. В C# 9 и более поздних версий можно использовать метод доступа `init` вместо `set`, чтобы сделать свойство доступным только для чтения.

В отличие от полей, свойства не классифицируются как переменные. Соответственно, нельзя передать свойство в качестве параметра ref или out.

Свойства нашли широкое применение в программировании. Их можно использовать для проверки данных перед подтверждением изменения; для прозрачного предоставления доступа к данным класса, которые фактически извлекаются из других источников, например из базы данных; для выполнения действия при изменении данных (например, в этом случае может создаваться событие или изменяться значение других полей).

При объявлении свойств в блоке класса указывается уровень доступа поля, затем тип и имя свойства, а после этого блок кода, в котором объявляются методы доступа `get` и (или) `set`. Пример:

```
public class Date
{
    private int _month = 7;  // Backing store

    public int Month
    {
        get => _month;
        set
        {
            if ((value > 0) && (value < 13))
            {
                _month = value;
            }
        }
    }
}
```

В этом примере `Month` объявляется как свойство, а метод доступа `set` обеспечивает установку значения `Month` в диапазоне от 1 до 12. Для отслеживания фактического значения свойство `Month` использует частное поле. Фактическое местоположение данных свойства часто называется "резервным хранилищем". Как правило, в качестве резервного хранилища свойства используют частные поля. Поле помечается как частное для того, чтобы гарантировать возможность его изменения только посредством вызова свойства. Дополнительные сведения об ограничениях открытого и закрытого доступа см. в разделе Модификаторы доступа.

Автоматически реализуемые свойства поддерживают упрощенный синтаксис для простых объявлений свойств. Дополнительные сведения см. в разделе Автоматически реализуемые свойства.

Метод доступа get

Тело метода доступа `get` похоже на тело метода. Оно должно возвращать значение заданного типа свойства. Выполнение метода доступа `get` эквивалентно считыванию значения поля. Например, если включена оптимизация и метод доступа `get` возвращает частную переменную, вызов метода доступа `get` определяется компилятором как встроенный, что позволяет исключить затраты ресурсов на вызов метода. Тем не менее виртуальный метод доступа `get` не может определяться как встроенный, поскольку во время компиляции компилятору не известно, как метод может быть фактически вызван во время выполнения. Ниже показан метод доступа `get`, возвращающий значение частного поля `_name`:

```
class Person
{
    private string _name;  // the name field
    public string Name => _name;     // the Name property
}
```

При ссылке на свойство (кроме случаев, когда свойство является целью присваивания) вызывается метод доступа `get`, который считывает значение свойства. Пример:

```
Person person = new Person();
//...

System.Console.Write(person.Name);  // the get accessor is invoked here
```

Метод доступа `get` должен завершаться инструкцией return или throw, при этом управление не может передаваться из тела метода доступа.

Изменение состояния объекта с помощью метода доступа `get` считается ошибочным стилем программирования. Например, побочным эффектом следующего метода доступа является изменение состояния объекта каждый раз при доступе к полю `_number`.

```
private int _number;
public int Number => _number++; // Don't do this
```

Метод доступа `get` можно использовать для возврата значения поля напрямую или после вычисления. Пример:

```
class Employee
{
    private string _name;
    public string Name => _name != null ? _name : "NA";
}
```

Если в предыдущем фрагменте кода свойству `Name` не присвоено значение, будет возвращено значение `NA`.

Метод доступа set

Метод доступа `set` похож на метод с типом возвращаемого значения void. В нем используется неявный параметр `value`, тип которого соответствует типу свойства. В следующем примере метод доступа `set` добавляется к свойству `Name`:

```
class Person
{
    private string _name;  // the name field
    public string Name    // the Name property
    {
        get => _name;
        set => _name = value;
    }
}
```

При присвоении значения свойству вызывается метод доступа `set` с аргументом, содержащим новое значение. Пример:

```
Person person = new Person();
person.Name = "Joe";  // the set accessor is invoked here

System.Console.Write(person.Name);  // the get accessor is invoked here
```

Использование имени явного параметра (`value`) для объявления локальной переменной в методе доступа `set` является ошибкой.

Метод доступа init

Код для создания метода доступа `init` аналогичен коду для создания метода доступа `set` , за исключением того, что используется ключевое слово `init` вместо `set` . Различие заключается в том, что метод доступа `init` можно использовать только в конструкторе или с помощью инициализатора объекта.

Комментарии

Свойства могут быть помечены как `public` , `private` , `protected` , `internal` , `protected internal` или `private protected` . Эти модификаторы доступа определяют, каким образом пользователи класса смогут получать доступ к свойству. Методы доступа `get` и `set` для одного свойства могут иметь разные модификаторы доступа. Например, метод доступа `get` может иметь модификатор `public` , разрешающий доступ из-за пределов типа только для чтения, а метод доступа `set` — модификатор `private` или `protected` . Дополнительные сведения см. в статье Модификаторы доступа.

Свойство может быть объявлено как статическое с помощью ключевого слова `static` . Это делает свойство доступным для вызывающих объектов в любое время, даже если экземпляр класса не существует. Дополнительные сведения см. в статье Статические классы и члены статических классов.

Свойство может быть помечено как виртуальное с помощью ключевого слова virtual. Это позволяет производным классам переопределять поведение свойства с помощью ключевого слова override. Дополнительные сведения об этих параметрах см. в разделе Наследование.

Свойство, переопределяющее виртуальное свойство, также может быть запечатанным (sealed). Это указывает, что для производных классов оно больше не является виртуальным. Наконец, свойство можно объявить абстрактным (abstract). Это означает, что в классе не будет определена реализация такого свойства, и в производных классах должны использоваться собственные реализации. Дополнительные сведения об этих параметрах см. в разделе Абстрактные и запечатанные классы и члены классов.

NOTE

Использование модификаторов virtual, abstract или override в методе доступа статического (static) свойства является ошибкой.

Примеры

В этом примере демонстрируются свойства экземпляра, а также статические и доступные только для чтения свойства. Этот метод принимает введенное с клавиатуры имя сотрудника, увеличивает значение `NumberOfEmployees` на 1, после чего отображает имя и номер сотрудника.

```
public class Employee
{
    public static int NumberOfEmployees;
    private static int _counter;
    private string _name;

    // A read-write instance property:
    public string Name
    {
        get => _name;
        set => _name = value;
    }

    // A read-only static property:
    public static int Counter => _counter;

    // A Constructor:
    public Employee() => _counter = ++NumberOfEmployees; // Calculate the employee's number:
}

class TestEmployee
{
    static void Main()
    {
        Employee.NumberOfEmployees = 107;
        Employee e1 = new Employee();
        e1.Name = "Claude Vige";

        System.Console.WriteLine("Employee number: {0}", Employee.Counter);
        System.Console.WriteLine("Employee name: {0}", e1.Name);
    }
}
/* Output:
    Employee number: 108
    Employee name: Claude Vige
*/
```

Пример скрытого свойства

В этом примере демонстрируется доступ к свойству базового класса, которое скрыто в производном классе другим свойством с таким же именем:

```
public class Employee
{
    private string _name;
    public string Name
    {
        get => _name;
        set => _name = value;
    }
}

public class Manager : Employee
{
    private string _name;

    // Notice the use of the new modifier:
    public new string Name
    {
        get => _name;
        set => _name = value + ", Manager";
    }
}

class TestHiding
{
    static void Main()
    {
        Manager m1 = new Manager();

        // Derived class property.
        m1.Name = "John";

        // Base class property.
        ((Employee)m1).Name = "Mary";

        System.Console.WriteLine("Name in the derived class is: {0}", m1.Name);
        System.Console.WriteLine("Name in the base class is: {0}", ((Employee)m1).Name);
    }
}
/* Output:
    Name in the derived class is: John, Manager
    Name in the base class is: Mary
*/
```

На что следует обратить внимание в предыдущем примере:

- Свойство `Name` в производном классе скрывает свойство `Name` базового класса. В таком случае в объявлении свойства в производном классе используется модификатор `new`:

  ```
  public new string Name
  ```

- Для доступа к скрытому свойству в базовом классе используется приведение `(Employee)`:

  ```
  ((Employee)m1).Name = "Mary";
  ```

  Дополнительные сведения о скрытии элементов см. в разделе Модификатор new.

Пример переопределения свойства

В этом примере два класса (`Cube` и `Square`) реализуют абстрактный класс `Shape` и переопределяют его абстрактное свойство `Area` . Обратите внимание на использование модификатора override в свойствах. Программа принимает введенную длину стороны, на основании которой рассчитывает площади квадрата

и куба. Также принимается введенное значение площади, на основании которой рассчитываются длины сторон квадрата и куба.

```
abstract class Shape
{
    public abstract double Area
    {
        get;
        set;
    }
}

class Square : Shape
{
    public double side;

    //constructor
    public Square(double s) => side = s;

    public override double Area
    {
        get => side * side;
        set => side = System.Math.Sqrt(value);
    }
}

class Cube : Shape
{
    public double side;

    //constructor
    public Cube(double s) => side = s;

    public override double Area
    {
        get => 6 * side * side;
        set => side = System.Math.Sqrt(value / 6);
    }
}

class TestShapes
{
    static void Main()
    {
        // Input the side:
        System.Console.Write("Enter the side: ");
        double side = double.Parse(System.Console.ReadLine());

        // Compute the areas:
        Square s = new Square(side);
        Cube c = new Cube(side);

        // Display the results:
        System.Console.WriteLine("Area of the square = {0:F2}", s.Area);
        System.Console.WriteLine("Area of the cube = {0:F2}", c.Area);
        System.Console.WriteLine();

        // Input the area:
        System.Console.Write("Enter the area: ");
        double area = double.Parse(System.Console.ReadLine());

        // Compute the sides:
        s.Area = area;
        c.Area = area;

        // Display the results:
        System.Console.WriteLine("Side of the square = {0:F2}", s.side);
        System.Console.WriteLine("Side of the cube = {0:F2}", c.side);
```

```
            System.Console.WriteLine("Side of the cube = {0:F2}", c.side);
        }
    }
    /* Example Output:
        Enter the side: 4
        Area of the square = 16.00
        Area of the cube = 96.00

        Enter the area: 24
        Side of the square = 4.90
        Side of the cube = 2.00
    */
```

См. также

- Руководство по программированию на C#
- Свойства
- Свойства интерфейса
- Автоматически реализуемые свойства

Свойства интерфейса (Руководство по программированию на C#)

18.12.2021 • 2 minutes to read

Свойства можно объявлять для интерфейса. Следующий пример объявляет метод доступа к свойству интерфейса:

```
public interface ISampleInterface
{
    // Property declaration:
    string Name
    {
        get;
        set;
    }
}
```

Свойства интерфейса обычно не имеют тела. Методы доступа указывают, доступно ли свойство для чтения и записи, только для чтения или только для записи. В отличие от классов и структур, объявление методов доступа без тела не приводит к объявлению автоматически реализуемого свойства. Начиная с C# 8.0 интерфейс может определять реализацию по умолчанию для членов, включая свойства. Определение реализации по умолчанию для свойства в интерфейсе используется редко, так как интерфейсы могут не определять поля данных экземпляра.

Пример

В этом примере интерфейс `IEmployee` содержит доступное для чтения и записи свойство `Name`, а также свойство `Counter`, предназначенное только для чтения. Класс `Employee` реализует интерфейс `IEmployee` и использует эти два свойства. Программа считывает имя нового сотрудника и текущее число сотрудников, после чего отображает имя сотрудника и его рассчитанный номер.

Вы можете использовать полное имя свойства, которое ссылается на интерфейс, где был объявлен член. Пример:

```
string IEmployee.Name
{
    get { return "Employee Name"; }
    set { }
}
```

В предыдущем примере показана явная реализация интерфейса. Например, если класс `Employee` реализует два интерфейса (`ICitizen` и `IEmployee`), оба из которых содержат свойство `Name`, потребуется явная реализация члена интерфейса. Это значит, что потребуется следующее объявление свойства:

```
string IEmployee.Name
{
    get { return "Employee Name"; }
    set { }
}
```

реализует свойство `Name` в интерфейсе `IEmployee`, тогда как следующее объявление:

```
string ICitizen.Name
{
    get { return "Citizen Name"; }
    set { }
}
```

реализует свойство `Name` в интерфейсе `ICitizen`.

```
interface IEmployee
{
    string Name
    {
        get;
        set;
    }

    int Counter
    {
        get;
    }
}

public class Employee : IEmployee
{
    public static int numberOfEmployees;

    private string _name;
    public string Name  // read-write instance property
    {
        get => _name;
        set => _name = value;
    }

    private int _counter;
    public int Counter  // read-only instance property
    {
        get => _counter;
    }

    // constructor
    public Employee() => _counter = ++numberOfEmployees;
}
```

```
System.Console.Write("Enter number of employees: ");
Employee.numberOfEmployees = int.Parse(System.Console.ReadLine());

Employee e1 = new Employee();
System.Console.Write("Enter the name of the new employee: ");
e1.Name = System.Console.ReadLine();

System.Console.WriteLine("The employee information:");
System.Console.WriteLine("Employee number: {0}", e1.Counter);
System.Console.WriteLine("Employee name: {0}", e1.Name);
```

Пример полученных результатов

```
Enter number of employees: 210
Enter the name of the new employee: Hazem Abolrous
The employee information:
Employee number: 211
Employee name: Hazem Abolrous
```

См. также

- Руководство по программированию на C#
- Свойства
- Использование свойств
- Сравнение свойств и индексаторов
- Индексаторы
- Интерфейсы

Ограничение доступности методов доступа (Руководство по программированию на C#)

18.12.2021 • 4 minutes to read

Выражения get и set свойства или индексатора называются *методами доступа*. По умолчанию они имеют такие же уровни видимости или доступа, что и свойство или индексатор, которым они принадлежат. Дополнительные сведения см. в разделе Уровни доступа. Тем не менее в некоторых случаях рекомендуется ограничить уровни доступа для этих методов. Как правило, в этом случае ограничивается уровень доступа для метода `set`, тогда как метод `get` остается общедоступным. Пример:

```
private string _name = "Hello";

public string Name
{
    get
    {
        return _name;
    }
    protected set
    {
        _name = value;
    }
}
```

В этом примере свойство `Name` определяет методы доступа `get` и `set`. Метод доступа `get` получает уровень доступа самого свойства (в этом случае `public`), а к методу `set` явным образом применяется модификатор доступа protected.

Ограничения модификаторов доступа для методов доступа

При использовании модификаторов доступа для свойств или индексаторов необходимо соблюдать следующие ограничения:

- Модификаторы доступа нельзя использовать в интерфейсе или явной реализации элемента interface.
- Модификаторы доступа можно использовать только в том случае, если для свойства или индексатора определены одновременно методы доступа `set` и `get`. В этом случае модификатор может применяться только к одному из двух методов доступа.
- Если свойству или индексатору назначен модификатор override, модификатор доступа должен соответствовать методу доступа для переопределенного метода доступа, если такой существует.
- Уровень доступности для метода доступа должен быть более строгим по сравнению с уровнем доступа самого свойства или идентификатора.

Модификаторы доступа при переопределении методов доступа

При переопределении свойства или индексатора переопределенные методы доступа должны быть доступны коду переопределения. Кроме того, уровни доступа свойства или индексатора и их методов доступа должны совпадать с уровнями переопределенных свойства или индексатора и их методов доступа. Пример:

```
public class Parent
{
    public virtual int TestProperty
    {
        // Notice the accessor accessibility level.
        protected set { }

        // No access modifier is used here.
        get { return 0; }
    }
}
public class Kid : Parent
{
    public override int TestProperty
    {
        // Use the same accessibility level as in the overridden accessor.
        protected set { }

        // Cannot use access modifier here.
        get { return 0; }
    }
}
```

Реализация интерфейсов

Методы доступа, которые используются для реализации интерфейса, не могут иметь модификаторы доступа. Тем не менее при реализации интерфейса с использованием одного метода доступа (например, `get`) другой метод доступа может иметь модификатор доступа, как показано в следующем примере:

```
public interface ISomeInterface
{
    int TestProperty
    {
        // No access modifier allowed here
        // because this is an interface.
        get;
    }
}

public class TestClass : ISomeInterface
{
    public int TestProperty
    {
        // Cannot use access modifier here because
        // this is an interface implementation.
        get { return 10; }

        // Interface property does not have set accessor,
        // so access modifier is allowed.
        protected set { }
    }
}
```

Домен доступности для метода доступа

При использовании модификатора доступа для метода доступа этот модификатор определяет домен доступности для такого метода.

Если к методу доступа не применяется модификатор доступа, домен доступности этого метода определяется уровнем доступности свойства или индексатора.

Пример

В следующем примере определяются три класса: `BaseClass`, `DerivedClass` и `MainClass`. В классе `BaseClass` для обоих классов определены свойства `Name` и `Id`. В этом примере демонстрируется, как свойство `Id` класса `DerivedClass` может быть скрыто свойством `Id` класса `BaseClass` при использовании ограничивающего модификатора доступа, такого как protected или private. Таким образом, при присвоении значений этому свойству вместо него вызывается свойство класса `BaseClass`. Чтобы сделать это свойство доступным, необходимо заменить модификатор доступа на public.

В этом примере также демонстрируется, что ограничивающий модификатор доступа (`private` или `protected`) для метода доступа `set` свойства `Name` в классе `DerivedClass` запрещает доступ к этому методу и при попытке присвоить ему значение возвращает ошибку.

```
public class BaseClass
{
    private string _name = "Name-BaseClass";
    private string _id = "ID-BaseClass";

    public string Name
    {
        get { return _name; }
        set { }
    }

    public string Id
    {
        get { return _id; }
        set { }
    }
}

public class DerivedClass : BaseClass
{
    private string _name = "Name-DerivedClass";
    private string _id = "ID-DerivedClass";

    new public string Name
    {
        get
        {
            return _name;
        }

        // Using "protected" would make the set accessor not accessible.
        set
        {
            _name = value;
        }
    }

    // Using private on the following property hides it in the Main Class.
    // Any assignment to the property will use Id in BaseClass.
    new private string Id
    {
        get
        {
            return _id;
        }
        set
        {
            _id = value;
        }
    }
}
```

```
class MainClass
{
    static void Main()
    {
        BaseClass b1 = new BaseClass();
        DerivedClass d1 = new DerivedClass();

        b1.Name = "Mary";
        d1.Name = "John";

        b1.Id = "Mary123";
        d1.Id = "John123";  // The BaseClass.Id property is called.

        System.Console.WriteLine("Base: {0}, {1}", b1.Name, b1.Id);
        System.Console.WriteLine("Derived: {0}, {1}", d1.Name, d1.Id);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    Base: Name-BaseClass, ID-BaseClass
    Derived: John, ID-BaseClass
*/
```

Комментарии

Обратите внимание, что если заменить объявление `new private string Id` на `new public string Id`, будут получены следующие выходные данные:

```
Name and ID in the base class: Name-BaseClass, ID-BaseClass
```

```
Name and ID in the derived class: John, John123
```

См. также

- Руководство по программированию на C#
- Свойства
- Индексаторы
- Модификаторы доступа

Практическое руководство. Объявление и использование свойств чтения и записи (руководство по программированию на C#)

18.12.2021 • 2 minutes to read

Свойства имеют все преимущества открытых членов данных, но не связаны с рисками незащищенного, неконтролируемого и несанкционированного доступа к данным объекта. Это достигается благодаря применению *методов доступа*, которые представляют собой особые методы для присвоения и извлечения значений базового члена данных. Метод доступа set присваивает значения членам данных, а метод доступа get извлекает их.

Это можно продемонстрировать на примере класса `Person`, который содержит два свойства: `Name` (string) и `Age` (int). Для обоих свойств реализованы методы доступа `get` и `set`, благодаря чему они доступны и для чтения, и для записи.

Пример

```
class Person
{
    private string _name = "N/A";
    private int _age = 0;

    // Declare a Name property of type string:
    public string Name
    {
        get
        {
            return _name;
        }
        set
        {
            _name = value;
        }
    }

    // Declare an Age property of type int:
    public int Age
    {
        get
        {
            return _age;
        }

        set
        {
            _age = value;
        }
    }

    public override string ToString()
    {
        return "Name = " + Name + ", Age = " + Age;
    }
}

class TestPerson
{
```

```
{
    static void Main()
    {
        // Create a new Person object:
        Person person = new Person();

        // Print out the name and the age associated with the person:
        Console.WriteLine("Person details - {0}", person);

        // Set some values on the person object:
        person.Name = "Joe";
        person.Age = 99;
        Console.WriteLine("Person details - {0}", person);

        // Increment the Age property:
        person.Age += 1;
        Console.WriteLine("Person details - {0}", person);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Person details - Name = N/A, Age = 0
    Person details - Name = Joe, Age = 99
    Person details - Name = Joe, Age = 100
*/
```

Отказоустойчивость

В предыдущем примере свойства `Name` и `Age` являются открытыми и содержат одновременно методы доступа `get` и `set`. Благодаря этому объект может считывать эти свойства и записывать их. Тем не менее в некоторых случаях необходимо исключить один из методов доступа. Например, свойство без метода доступа `set` будет доступно только для чтения:

```
public string Name
{
    get
    {
        return _name;
    }
    set
    {
        _name = value;
    }
}
```

Кроме того, можно сделать один метод доступа открытым, а другой оставить частным или защищенным. Дополнительные сведения см. в разделе Асимметричные методы доступа.

После объявления свойств их можно использовать так же, как поля класса. Это позволяет получить естественный синтаксис извлечения и установки значения свойства, как показано на примере следующих операторов:

```
person.Name = "Joe";
person.Age = 99;
```

Обратите внимание, что в методе `set` свойства доступна специальная переменная `value`. Она содержит значение, заданное пользователем, например:

```
_name = value;
```

Обратите внимание на простой синтаксис приращения свойства `Age` для объекта `Person`:

```
person.Age += 1;
```

Если для моделирования свойств использовать отдельные методы `set` и `get`, аналогичный код может иметь следующий вид:

```
person.SetAge(person.GetAge() + 1);
```

В этом примере переопределяется метод `ToString`:

```
public override string ToString()
{
    return "Name = " + Name + ", Age = " + Age;
}
```

Обратите внимание, что метод `ToString` не используется в этой программе явно. Он вызывается по умолчанию при вызове `WriteLine`.

См. также

- Руководство по программированию на C#
- Свойства
- Система типов C#

Автоматически реализуемые свойства (Руководство по программированию на C#)

18.12.2021 • 2 minutes to read

В C# 3.0 и более поздних версиях автоматически реализуемые свойства делают объявление свойств более лаконичным, когда в методах доступа к свойствам не требуется дополнительная логика. Они также позволяют клиентскому коду создавать объекты. При объявлении свойства, как показано в следующем примере, компилятор создает закрытое анонимное резервное поле, которое может быть доступно только через методы доступа `get` и `set` свойства. В C# 9 и более поздних версий методы доступа `init` также можно объявить как автоматически реализуемые свойства.

Пример

В следующем примере показан простой класс, имеющий несколько автоматически реализуемых свойств.

```
// This class is mutable. Its data can be modified from
// outside the class.
class Customer
{
    // Auto-implemented properties for trivial get and set
    public double TotalPurchases { get; set; }
    public string Name { get; set; }
    public int CustomerId { get; set; }

    // Constructor
    public Customer(double purchases, string name, int id)
    {
        TotalPurchases = purchases;
        Name = name;
        CustomerId = id;
    }

    // Methods
    public string GetContactInfo() { return "ContactInfo"; }
    public string GetTransactionHistory() { return "History"; }

    // .. Additional methods, events, etc.
}

class Program
{
    static void Main()
    {
        // Intialize a new object.
        Customer cust1 = new Customer(4987.63, "Northwind", 90108);

        // Modify a property.
        cust1.TotalPurchases += 499.99;
    }
}
```

В интерфейсах невозможно объявлять автоматически реализуемые свойства. Автоматически реализуемые свойства объявляют резервное поле частного экземпляра, а интерфейсы могут не объявлять поля экземпляров. Объявление свойства в интерфейсе без определения тела приводит к объявлению свойства с методами доступа, которые должны реализовываться каждым типом,

реализующим этот интерфейс.

В C# 6 и более поздних версиях можно инициализировать автоматически реализуемые свойства аналогично полям.

```
public string FirstName { get; set; } = "Jane";
```

Класс, который показан в предыдущем примере, является изменяемым. Клиентский код может изменить значения в объектах после создания. В сложных классах, которые содержат значительные возможности (методы) и данные, часто необходимо иметь открытые свойства. Но для небольших классов или структур, которые просто инкапсулируют набор значений (данных) без какого-либо поведения (или с минимальным поведением), следует использовать один из следующих параметров, чтобы сделать объекты неизменяемыми:

- Объявите только метод доступа `get` (неизменяемое значение везде, за исключением конструктора).
- Объявите методы доступа `get` и `init` (неизменяемое значение везде, за исключением создания объекта).
- Объявите метод доступа `set` с атрибутом private (неизменяемое значение для потребителей).

Дополнительные сведения см. в статье Практическое руководство. Реализация облегченного класса с автоматически реализуемыми свойствами.

См. также

- Свойства
- Модификаторы

Практическое руководство. Реализация облегченного класса с автоматически реализуемыми свойствами (руководство по программированию на C#)

18.12.2021 • 3 minutes to read

В этом примере показано, как создать неизменяемый упрощенный класс, служащий исключительно для инкапсуляции набора автоматически реализуемых свойств. Используйте такую конструкцию вместо структуры, когда требуется использовать семантику ссылочного типа.

Неизменяемое свойство можно создать следующими способами:

- Объявить только метод доступа get, который делает свойство неизменяемым везде, кроме конструктора типа.
- Объявить метод доступа init вместо `set`, который делает свойство изменяемым только в конструкторе или с помощью инициализатора объектов.
- Объявить метод доступа set как private. Свойство можно задать в типе, но оно является неизменяемым для потребителей.

  При объявлении закрытого метода доступа `set` для инициализации свойства нельзя использовать инициализатор объекта. Необходимо использовать конструктор или фабричный метод.

В следующем примере показано, как свойство с методом доступа get, отличается от свойства с методами доступа get и private set.

```
class Contact
{
    public string Name { get; }
    public string Address { get; private set; }

    public Contact(string contactName, string contactAddress)
    {
        // Both properties are accessible in the constructor.
        Name = contactName;
        Address = contactAddress;
    }

    // Name isn't assignable here. This will generate a compile error.
    //public void ChangeName(string newName) => Name = newName;

    // Address is assignable here.
    public void ChangeAddress(string newAddress) => Address = newAddress;
}
```

Пример

В следующем примере показаны два способа реализации неизменяемого класса с автоматически реализуемыми свойствами. Каждый способ объявляет одно из свойств с закрытым методом доступа `set` и одно из свойств только с методом доступа `get`. Первый класс использует конструктор только для инициализации свойства, а второй класс использует статический фабричный метод, вызывающий

конструктор.

```
// This class is immutable. After an object is created,
// it cannot be modified from outside the class. It uses a
// constructor to initialize its properties.
class Contact
{
    // Read-only property.
    public string Name { get; }

    // Read-write property with a private set accessor.
    public string Address { get; private set; }

    // Public constructor.
    public Contact(string contactName, string contactAddress)
    {
        Name = contactName;
        Address = contactAddress;
    }
}

// This class is immutable. After an object is created,
// it cannot be modified from outside the class. It uses a
// static method and private constructor to initialize its properties.
public class Contact2
{
    // Read-write property with a private set accessor.
    public string Name { get; private set; }

    // Read-only property.
    public string Address { get; }

    // Private constructor.
    private Contact2(string contactName, string contactAddress)
    {
        Name = contactName;
        Address = contactAddress;
    }

    // Public factory method.
    public static Contact2 CreateContact(string name, string address)
    {
        return new Contact2(name, address);
    }
}

public class Program
{
    static void Main()
    {
        // Some simple data sources.
        string[] names = {"Terry Adams","Fadi Fakhouri", "Hanying Feng",
                          "Cesar Garcia", "Debra Garcia"};
        string[] addresses = {"123 Main St.", "345 Cypress Ave.", "678 1st Ave",
                              "12 108th St.", "89 E. 42nd St."};

        // Simple query to demonstrate object creation in select clause.
        // Create Contact objects by using a constructor.
        var query1 = from i in Enumerable.Range(0, 5)
                     select new Contact(names[i], addresses[i]);

        // List elements cannot be modified by client code.
        var list = query1.ToList();
        foreach (var contact in list)
        {
            Console.WriteLine("{0}, {1}", contact.Name, contact.Address);
        }
```

```
        // Create Contact2 objects by using a static factory method.
        var query2 = from i in Enumerable.Range(0, 5)
                     select Contact2.CreateContact(names[i], addresses[i]);

        // Console output is identical to query1.
        var list2 = query2.ToList();

        // List elements cannot be modified by client code.
        // CS0272:
        // list2[0].Name = "Eugene Zabokritski";

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
    Terry Adams, 123 Main St.
    Fadi Fakhouri, 345 Cypress Ave.
    Hanying Feng, 678 1st Ave
    Cesar Garcia, 12 108th St.
    Debra Garcia, 89 E. 42nd St.
*/
```

Компилятор создает резервные поля для каждого автоматически реализуемого свойства. Эти поля недоступны непосредственно из исходного кода.

См. также

- Свойства
- struct
- Инициализаторы объектов и коллекций

Методы (Руководство по программированию на C#)

18.12.2021 • 8 minutes to read

Метод — это блок кода, содержащий ряд инструкций. Программа инициирует выполнение инструкций, вызывая метод и указывая все аргументы, необходимые для этого метода. В C# все инструкции выполняются в контексте метода.

Метод `Main` является точкой входа для каждого приложения C# и вызывается общеязыковой средой выполнения (CLR) при запуске программы. В приложении, использующем инструкции верхнего уровня, метод `Main` создается компилятором и содержит все инструкции верхнего уровня.

> **NOTE**
> В этой статье рассматриваются названные методы. Дополнительные сведения об анонимных функциях см. в статье Лямбда-выражения.

Сигнатуры методов

Методы объявляются в классе, структуре или интерфейсе путем указания уровня доступа, такого как `public` или `private`, необязательных модификаторов, таких как `abstract` или `sealed`, возвращаемого значения, имени метода и всех параметров этого метода. Все эти части вместе представляют собой сигнатуру метода.

> **IMPORTANT**
> Тип возврата метода не является частью сигнатуры метода в целях перегрузки метода. Однако он является частью сигнатуры метода при определении совместимости между делегатом и методом, который он указывает.

Параметры метода заключаются в скобки и разделяются запятыми. Пустые скобки указывают, что параметры методу не требуются. Этот класс содержит четыре метода:

```
abstract class Motorcycle
{
    // Anyone can call this.
    public void StartEngine() {/* Method statements here */ }

    // Only derived classes can call this.
    protected void AddGas(int gallons) { /* Method statements here */ }

    // Derived classes can override the base class implementation.
    public virtual int Drive(int miles, int speed) { /* Method statements here */ return 1; }

    // Derived classes must implement this.
    public abstract double GetTopSpeed();
}
```

Доступ к методу

Вызов метода в объекте аналогичен доступу к полю. После имени объекта добавьте точку, имя метода и

круглые скобки. Аргументы перечисляются в этих скобках и разделяются запятыми. Таким образом, методы класса `Motorcycle` могут вызываться, как показано в следующем примере:

```
class TestMotorcycle : Motorcycle
{

    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {

        TestMotorcycle moto = new TestMotorcycle();

        moto.StartEngine();
        moto.AddGas(15);
        moto.Drive(5, 20);
        double speed = moto.GetTopSpeed();
        Console.WriteLine("My top speed is {0}", speed);
    }
}
```

Параметры и аргументы метода

Определение метода задает имена и типы всех необходимых параметров. Когда вызывающий код вызывает метод, он предоставляет конкретные значения, называемые аргументами, для каждого параметра. Аргументы должны быть совместимы с типом параметра, но имя аргумента (если есть), используемое в вызывающем коде, не обязательно должно совпадать с именем параметра, указанным в методе. Пример:

```
public void Caller()
{
    int numA = 4;
    // Call with an int variable.
    int productA = Square(numA);

    int numB = 32;
    // Call with another int variable.
    int productB = Square(numB);

    // Call with an integer literal.
    int productC = Square(12);

    // Call with an expression that evaulates to int.
    productC = Square(productA * 3);
}

int Square(int i)
{
    // Store input argument in a local variable.
    int input = i;
    return input * input;
}
```

Передача по ссылке и передача по значению

По умолчанию при передаче в метод экземпляра типа значения вместо самого этого экземпляра передается его копия. Поэтому изменения в аргументе не оказывают влияния на исходный экземпляр в

вызывающем методе. Чтобы передать экземпляр типа значения по ссылке, используйте ключевое слово `ref`. Дополнительные сведения см. в разделе Передача параметров типа значения.

При передаче в метод объекта ссылочного типа передается ссылка на этот объект. То есть метод получает не сам объект, а аргумент, который указывает расположение объекта. При изменении члена объекта с помощью этой ссылки это изменение отражается в аргументе в вызывающем методе, даже если объект передается по значению.

Ссылочный тип создается с помощью ключевого слова `class`, как показано в следующем примере.

```
public class SampleRefType
{
    public int value;
}
```

Теперь, если передать объект, основанный на этом типе, в метод, то будет передана ссылка на объект. В следующем примере объект типа `SampleRefType` передается в метод `ModifyObject`:

```
public static void TestRefType()
{
    SampleRefType rt = new SampleRefType();
    rt.value = 44;
    ModifyObject(rt);
    Console.WriteLine(rt.value);
}

static void ModifyObject(SampleRefType obj)
{
    obj.value = 33;
}
```

В этом примере, в сущности, делается то же, что и в предыдущем примере, — аргумент по значению передается в метод. Но поскольку здесь используется ссылочный тип, результат будет другим. В данном случае в методе `ModifyObject` изменено поле `value` параметра `obj`, а также изменено поле `value` аргумента, `rt` в методе `TestRefType`. В качестве выходных данных метод `TestRefType` отображает 33.

Дополнительные сведения о передаче ссылочных типов по ссылке и по значению см. в разделах Передача параметров ссылочного типа и Ссылочные типы.

Возвращаемые значения

Методы могут возвращать значение вызывающему объекту. Если тип возврата, указываемый перед именем метода, не `void`, этот метод может возвращать значение с помощью оператора `return`. Инструкция с ключевым словом `return`, за которым следует значение, соответствующее типу возврата, будет возвращать это значение объекту, вызвавшему метод.

Значение может возвращаться вызывающему объекту по значению или, начиная с C# версии 7.0, по ссылке. Значения возвращаются вызывающему объекту по ссылке, если ключевое слово `ref` используется в сигнатуре метода и указывается после каждого ключевого слова `return`. Например, следующая сигнатура метода и оператор return указывают, что метод возвращает переменную с именем `estDistance` вызывающему объекту по ссылке.

```
public ref double GetEstimatedDistance()
{
    return ref estDistance;
}
```

Ключевое слове `return` также останавливает выполнение метода. Если тип возврата — `void`, инструкцию `return` без значения по-прежнему можно использовать для завершения выполнения метода. Без ключевого слова `return` этот метод будет останавливать выполнение при достижении конца блока кода. Методы с типом возврата, отличным от void, должны использовать ключевое слово `return` для возврата значения. Например, в следующих двух методах ключевое слово `return` используется для возврата целочисленных значений.

```
class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2)
    {
        return number1 + number2;
    }

    public int SquareANumber(int number)
    {
        return number * number;
    }
}
```

Чтобы использовать значение, возвращаемое из метода, вызывающий метод может применять сам вызов метода везде, где будет достаточно значения того же типа. Можно также назначить возвращаемое значение переменной. Например, следующие два примера кода достигают одной и той же цели.

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

```
result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));
// The result is 9.
Console.WriteLine(result);
```

Использование локальной переменной, в данном случае `result`, для сохранения значения является необязательным. Это может улучшить читаемость кода или может оказаться необходимым, если нужно сохранить исходное значение аргумента для всей области метода.

Чтобы использовать значение, возвращаемое по ссылке из метода, необходимо объявить локальную ссылочную переменную, если планируется изменение значения. Например, если метод `Planet.GetEstimatedDistance` возвращает значение Double по ссылке, можно определить его как локальную ссылочную переменную с использованием кода следующего вида:

```
ref int distance = Planet.GetEstimatedDistance();
```

Возвращать многомерный массив из метода `M`, который изменяет содержимое массива, необязательно, если вызывающая функция передает массив в `M`. В целях оптимизации можно возвращать полученный массив из `M` или функциональный поток значений, однако это необязательно. Это связано с тем, что C# передает все ссылочные типы по значению, а значение ссылки на массив представляет собой указатель на массив. В методе `M` любые изменения содержимого массива отслеживаются любым кодом, имеющим ссылку на массив, как показано в приведенном ниже примере:

```
static void Main(string[] args)
{
    int[,] matrix = new int[2, 2];
    FillMatrix(matrix);
    // matrix is now full of -1
}

public static void FillMatrix(int[,] matrix)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
    {
        for (int j = 0; j < matrix.GetLength(1); j++)
        {
            matrix[i, j] = -1;
        }
    }
}
```

Асинхронные методы

С помощью функции async можно вызывать асинхронные методы, не прибегая к использованию явных обратных вызовов или ручному разделению кода между несколькими методами или лямбда-выражениями.

Если пометить метод с помощью модификатора async , можно использовать в этом методе инструкцию await . Когда управление достигает выражения await в асинхронном методе, управление возвращается вызывающему объекту и выполнение метода приостанавливается до завершения выполнения ожидающей задачи. После завершения задачи можно возобновить выполнение в методе.

> **NOTE**
> Асинхронный метод возвращается в вызывающий объект, когда он встречает первый ожидаемый объект, выполнение которого еще не завершено, или когда выполнение асинхронного метода доходит до конца — в зависимости от того, что происходит раньше.

Асинхронный метод обычно имеет тип возвращаемого значения Task<TResult>, Task, IAsyncEnumerable<T> или `void`. Тип возвращаемого значения `void` в основном используется для определения обработчиков событий, где требуется возвращать тип `void`. Асинхронный метод, который возвращает тип `void`, не может быть ожидающим. Вызывающий объект метода, возвращающего значение типа void, не может перехватывать исключения, которые выдает этот метод. Начиная с C# 7.0 асинхронный метод может возвращать любой тип вида задачи.

В следующем примере `DelayAsync` является асинхронным методом с типом возврата Task<TResult>. `DelayAsync` имеет инструкцию `return`, которая возвращает целое число. Поэтому объявление метода `DelayAsync` должно иметь тип возврата `Task<int>`. Поскольку тип возврата — `Task<int>`, вычисление выражения `await` в `DoSomethingAsync` создает целое число, как показывает следующая инструкция: `int result = await delayTask`.

В следующем примере метод `Main` служит примером асинхронного метода с типом возврата Task. Он переходит к методу `DoSomethingAsync` и, поскольку он выражается в одной строке, он может опустить ключевые слова `async` и `await`. Поскольку `DoSomethingAsync` является асинхронным методом, задача для вызова `DoSomethingAsync` должна быть ожидаемой, как показывает следующая инструкция: `await DoSomethingAsync();`.

```
using System;
using System.Threading.Tasks;

class Program
{
    static Task Main() => DoSomethingAsync();

    static async Task DoSomethingAsync()
    {
        Task<int> delayTask = DelayAsync();
        int result = await delayTask;

        // The previous two statements may be combined into
        // the following statement.
        //int result = await DelayAsync();

        Console.WriteLine($"Result: {result}");
    }

    static async Task<int> DelayAsync()
    {
        await Task.Delay(100);
        return 5;
    }
}
// Example output:
//   Result: 5
```

Асинхронный метод не может объявить все параметры ref или out , но может вызывать методы, которые имеют такие параметры.

Дополнительные сведения об асинхронных методах см. в разделах Асинхронное программирование с использованием ключевых слов async и await (C#) и Типы возвращаемых значений асинхронных операций.

Определения текста выражений

Часто используются определения методов, которые просто немедленно возвращаются с результатом выражения или которые имеют единственную инструкцию в тексте метода. Для определения таких методов существует сокращенный синтаксис с использованием `=>`:

```
public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public void Print() => Console.WriteLine(First + " " + Last);
// Works with operators, properties, and indexers too.
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);
```

Если метод возвращает `void` или является асинхронным методом, то текст метода должен быть выражением инструкции (так же, как при использовании лямбда-выражений). Свойства и индексаторы должны быть только для чтения, и вы не должны использовать ключевое слово `get` метода доступа.

Iterators

Итератор выполняет настраиваемую итерацию по коллекции, например по списку или массиву. Итератор использует инструкцию yield return для возврата всех элементов по одному. Когда достигается инструкция yield return , текущее расположение в коде запоминается. При следующем вызове итератора выполнение возобновляется с этого места.

Итератор вызывается из клиентского кода с помощью инструкции foreach .

Тип возврата итератора может быть IEnumerable, IEnumerable<T>, IEnumerator или IEnumerator<T>.

Дополнительные сведения см. в разделе Итераторы.

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Руководство по программированию на C#
- Система типов C#
- Модификаторы доступа
- Статические классы и члены статических классов
- Наследование
- Абстрактные и запечатанные классы и члены классов
- params
- out
- ref
- Передача параметров

Локальные функции (руководство по программированию на C#)

18.12.2021 • 9 minutes to read

Начиная с версии 7.0 в языке C# поддерживаются *локальные функции*. Локальные функции представляют собой частные методы типа, вложенные в другой элемент. Они могут вызываться только из того элемента, в который вложены. Ниже перечислены элементы, в которых можно объявлять и из которых можно вызывать локальные функции:

- Методы, в частности методы итератора и асинхронные методы
- Конструкторы
- Методы доступа свойств
- Методы доступа событий
- Анонимные методы
- Лямбда-выражения
- Методы завершения
- Другие локальные функции

Тем не менее локальные функции нельзя объявлять внутри элемента, воплощающего выражение.

> **NOTE**
> В некоторых случаях для реализации возможностей, поддерживаемых локальными функциями, также можно использовать лямбда-выражения. Дополнительные сведения см. в разделе Локальные функции или лямбда-выражения.

Применение локальных функций позволяет сделать предназначение кода более понятным. Другие пользователи, читающие ваш код, смогут видеть, что соответствующий метод вызывается только внутри того метода, в который он вложен. В случае с командными проектами это также гарантирует, что другой разработчик не сможет ошибочно вызвать метод напрямую из любого другого места в классе или структуре.

Синтаксис локальной функции

Локальная функция определяется как вложенный метод внутри содержащего ее элемента. Ниже приведен синтаксис определения локальной функции:

```
<modifiers> <return-type> <method-name> <parameter-list>
```

С локальной функцией можно использовать следующие модификаторы:

- `async`
- `unsafe`
- `static` (в C# 8.0 и более поздних версий). Статическая локальная функция не может сохранять локальные переменные или состояние экземпляра.
- `extern` (в C# 9.0 и более поздних версий). Внешней локальной функцией должна быть `static`.

Все локальные переменные, определенные в содержащем функцию элементе (включая параметры

метода), доступны в нестатической локальной функции.

В отличие от определения метода, определение локальной функции не может содержать модификатор доступа к элементу. Поскольку все локальные функции являются частными, при использовании модификатора доступа (например, ключевого слова `private`) возникает ошибка компилятора CS0106, "Модификатор "private" недопустим для этого элемента".

В следующем примере определяется локальная функция `AppendPathSeparator`, которая является частной для метода `GetText`:

```
private static string GetText(string path, string filename)
{
     var reader = File.OpenText($"{AppendPathSeparator(path)}{filename}");
     var text = reader.ReadToEnd();
     return text;

     string AppendPathSeparator(string filepath)
     {
        return filepath.EndsWith(@"\") ? filepath : filepath + @"\";
     }
}
```

Начиная с C# 9.0 можно применять атрибуты к локальной функции, ее параметрам и параметрам типа, как показано в следующем примере:

```
#nullable enable
private static void Process(string?[] lines, string mark)
{
    foreach (var line in lines)
    {
        if (IsValid(line))
        {
            // Processing logic...
        }
    }

    bool IsValid([NotNullWhen(true)] string? line)
    {
        return !string.IsNullOrEmpty(line) && line.Length >= mark.Length;
    }
}
```

В предыдущем примере используется специальный атрибут для помощи компилятору в статическом анализе в контексте, допускающем значение NULL.

Локальные функции и исключения

Полезной особенностью локальных функций является то, что они допускают немедленную обработку исключений. В случае с итераторами метода исключения обрабатываются только после перечисления возвращаемой последовательности, а не в момент извлечения итератора. В случае с асинхронными методами любые исключения, возникшие в таком методе, наблюдаются в тот момент, когда возвращаемая задача находится в состоянии ожидания.

В следующем примере определяется метод `OddSequence`, который перечисляет нечетные числа в заданном диапазоне. Поскольку он передает в метод перечислителя `OddSequence` число больше 100, этот метод вызывает исключение ArgumentOutOfRangeException. Как видно из выходных данных этого примера, исключение обрабатывается только в момент перебора чисел, а не при извлечении перечислителя.

```
using System;
using System.Collections.Generic;

public class IteratorWithoutLocalExample
{
   public static void Main()
   {
      IEnumerable<int> xs = OddSequence(50, 110);
      Console.WriteLine("Retrieved enumerator...");

      foreach (var x in xs)  // line 11
      {
         Console.Write($"{x} ");
      }
   }

   public static IEnumerable<int> OddSequence(int start, int end)
   {
      if (start < 0 || start > 99)
         throw new ArgumentOutOfRangeException(nameof(start), "start must be between 0 and 99.");
      if (end > 100)
         throw new ArgumentOutOfRangeException(nameof(end), "end must be less than or equal to 100.");
      if (start >= end)
         throw new ArgumentException("start must be less than end.");

      for (int i = start; i <= end; i++)
      {
         if (i % 2 == 1)
            yield return i;
      }
   }
}
// The example displays the output like this:
//
//    Retrieved enumerator...
//    Unhandled exception. System.ArgumentOutOfRangeException: end must be less than or equal to 100.
(Parameter 'end')
//    at IteratorWithoutLocalExample.OddSequence(Int32 start, Int32 end)+MoveNext() in
IteratorWithoutLocal.cs:line 22
//    at IteratorWithoutLocalExample.Main() in IteratorWithoutLocal.cs:line 11
```

Если поместить логику итератора в локальную функцию, при получении перечислителя вызываются исключения проверки аргументов, как показано в следующем примере:

```
using System;
using System.Collections.Generic;

public class IteratorWithLocalExample
{
   public static void Main()
   {
      IEnumerable<int> xs = OddSequence(50, 110);  // line 8
      Console.WriteLine("Retrieved enumerator...");

      foreach (var x in xs)
      {
         Console.Write($"{x} ");
      }
   }

   public static IEnumerable<int> OddSequence(int start, int end)
   {
      if (start < 0 || start > 99)
         throw new ArgumentOutOfRangeException(nameof(start), "start must be between 0 and 99.");
      if (end > 100)
         throw new ArgumentOutOfRangeException(nameof(end), "end must be less than or equal to 100.");
      if (start >= end)
         throw new ArgumentException("start must be less than end.");

      return GetOddSequenceEnumerator();

      IEnumerable<int> GetOddSequenceEnumerator()
      {
         for (int i = start; i <= end; i++)
         {
            if (i % 2 == 1)
               yield return i;
         }
      }
   }
}
// The example displays the output like this:
//
//    Unhandled exception. System.ArgumentOutOfRangeException: end must be less than or equal to 100.
(Parameter 'end')
//    at IteratorWithLocalExample.OddSequence(Int32 start, Int32 end) in IteratorWithLocal.cs:line 22
//    at IteratorWithLocalExample.Main() in IteratorWithLocal.cs:line 8
```

Локальные функции или лямбда-выражения

На первый взгляд, локальные функции и лямбда-выражения во многом похожи. Во многих случаях выбор между использованием лямбда-выражений и локальных функций определяется стилем и личными предпочтениями. Однако существуют реальные различия в использовании этих сущностей, о которых нужно знать.

Рассмотрим различия в реализации алгоритма вычисления факториала с использованием локальной функции и лямбда-выражения. В этой версии используется локальная функция:

```
public static int LocalFunctionFactorial(int n)
{
    return nthFactorial(n);

    int nthFactorial(int number) => number < 2
        ? 1
        : number * nthFactorial(number - 1);
}
```

В этой версии используются лямбда-выражения:

```
public static int LambdaFactorial(int n)
{
    Func<int, int> nthFactorial = default(Func<int, int>);

    nthFactorial = number => number < 2
        ? 1
        : number * nthFactorial(number - 1);

    return nthFactorial(n);
}
```

Именование

Локальные функции явно именуются как методы. Лямбда-выражения представляют собой анонимные методы и должны назначаться переменным типа `delegate`, как правило, типа `Action` или `Func`. Процесс объявления локальной функции аналогичен написанию обычного метода: вы объявляете тип возвращаемого значения и сигнатуру функции.

Сигнатуры функций и типы лямбда-выражений

Лямбда-выражения используют тип переменной `Action` / `Func`, которой они назначаются, для определения типов аргументов и возвращаемых значений. Поскольку синтаксис локальных функций во многом аналогичен обычному методу, типы аргументов и возвращаемых значений уже входят в объявление функции.

Начиная с C# 10, некоторые лямбда-выражения имеют *естественный тип*, который позволяет компилятору определить тип возвращаемого значения и типы параметров лямбда-выражения.

Определенное назначение

Лямбда-выражения — это объекты, которые объявляются и назначаются во время выполнения. Чтобы использовать лямбда-выражение, его необходимо определенно назначить: для этого необходимо объявить переменную `Action` / `Func`, которой оно будет назначено, и назначить ей лямбда-выражение. Обратите внимание на то, что `LambdaFactorial` должно объявить и инициализировать лямбда-выражение `nthFactorial`, прежде чем его определить. В противном случае возникает ошибка компилятора, связанная со ссылкой на объект `nthFactorial`, который еще не был назначен.

Локальные функции определяются во время компиляции. Они не назначаются переменным, в связи с чем на них можно ссылаться из любого расположения кода, **в области действия которого они находятся**. В первом примере `LocalFunctionFactorial` мы можем объявить локальную функцию выше или ниже инструкции `return`, и это не приведет к ошибкам компилятора.

Эти различия означают, что рекурсивные алгоритмы легче создавать, используя локальные функции. Можно объявить и определить локальную функцию, которая вызывает саму себя. Необходимо объявить лямбда-выражения и назначить им значение по умолчанию, прежде чем их можно будет переназначить телу, которое ссылается на то же лямбда-выражение.

Реализация в виде делегата

Лямбда-выражения преобразуются в делегаты при объявлении. Локальные функции являются более гибкими и могут определяться в виде традиционного метода *или* делегата. Локальные функции преобразуются в делегаты только при ***использовании*** в качестве делегата.

Если объявить локальную функцию и сослаться на нее только путем вызова этой функции в качестве метода, она не будет преобразована в делегат.

Захват переменной

Правила определенного назначения также влияют на любые переменные, захватываемые локальной

функцией или лямбда-выражением. Компилятор может выполнять статический анализ, что позволяет локальным функциям определенно назначать захватываемые переменные во включающей области. Рассмотрим следующий пример.

```
int M()
{
    int y;
    LocalFunction();
    return y;

    void LocalFunction() => y = 0;
}
```

Компилятор может определить, что `LocalFunction` определенно назначает `y` при вызове. Поскольку `LocalFunction` вызывается перед оператором `return`, `y` определенно назначается в операторе `return`.

Обратите внимание, что когда локальная функция захватывает переменные во включающей области, эта функция реализуется как тип делегата.

Распределение куч

В зависимости от использования при работе с локальными функциями можно избежать распределения куч, которое всегда необходимо выполнять при работе с лямбда-выражениями. Если локальная функция никогда не преобразуется в делегат и ни одна из переменных, захваченных локальной функцией, не захвачена другими лямбда-выражениями или локальными функциями, которые преобразуются в делегаты, компилятор может избежать распределения куч.

Рассмотрим следующий асинхронный пример:

```
public async Task<string> PerformLongRunningWorkLambda(string address, int index, string name)
{
    if (string.IsNullOrWhiteSpace(address))
        throw new ArgumentException(message: "An address is required", paramName: nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index), message: "The index must be non-
negative");
    if (string.IsNullOrWhiteSpace(name))
        throw new ArgumentException(message: "You must supply a name", paramName: nameof(name));

    Func<Task<string>> longRunningWorkImplementation = async () =>
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}. Enjoy.";
    };

    return await longRunningWorkImplementation();
}
```

Замыкание для этого лямбда-выражения содержит переменные `address`, `index` и `name`. При использовании локальных функций объект, который реализует замыкание, может иметь тип `struct`. Этот тип структуры будет передан в локальную функцию посредством ссылки. Эта разница в реализации позволяет избежать распределения.

Создание экземпляра, необходимое для лямбда-выражений, означает выделение дополнительной памяти, что в критически важном коде может ухудшить производительность. Локальные функции не создают этой перегрузки. В приведенном выше примере в версии с локальной функцией используется на две операции выделения памяти меньше по сравнению с версией на основе лямбда-выражения.

Если известно, что локальная функция не будет преобразована в делегат и ни одна из захватываемых ею

переменных не захватывается другими лямбда-выражениями или локальными функциями, которые преобразуются в делегаты, то можно гарантировать, что локальная функция не будет распределяться в куче за счет объявления в качестве локальной функции `static`. Обратите внимание, что эта возможность доступна в C# 8.0 и более поздних версиях.

> **NOTE**
>
> В эквивалентном этому методе на основе локальной функции также используется класс для замыкания. Реализация замыкания для локальной функции в формате `class` или `struct` зависит от особенностей реализации. Локальная функция может использовать `struct`, тогда как в лямбда-выражениях всегда используется `class`.

```
public async Task<string> PerformLongRunningWork(string address, int index, string name)
{
    if (string.IsNullOrWhiteSpace(address))
        throw new ArgumentException(message: "An address is required", paramName: nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index), message: "The index must be non-
negative");
    if (string.IsNullOrWhiteSpace(name))
        throw new ArgumentException(message: "You must supply a name", paramName: nameof(name));

    return await longRunningWorkImplementation();

    async Task<string> longRunningWorkImplementation()
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}. Enjoy.";
    }
}
```

Использование ключевого слова `yield`

Еще одно преимущество локальных функций, которое не показано в этом примере, заключается в том, что они могут быть реализованы в качестве итераторов с использованием синтаксиса `yield return` для создания последовательности значений.

```
public IEnumerable<string> SequenceToLowercase(IEnumerable<string> input)
{
    if (!input.Any())
    {
        throw new ArgumentException("There are no items to convert to lowercase.");
    }

    return LowercaseIterator();

    IEnumerable<string> LowercaseIterator()
    {
        foreach (var output in input.Select(item => item.ToLower()))
        {
            yield return output;
        }
    }
}
```

В лямбда-выражениях не допускается использование инструкции `yield return`. См. описание ошибки компилятора CS1621.

Локальные функции могут показаться избыточными для лямбда-выражений, поскольку обычно

применяются иначе и в других целях. Локальные функции более эффективны в случаях, когда вам нужно написать функцию, которая будет вызываться только из контекста другого метода.

См. также

- Методы

Возвращаемые ссылочные значения и ссылочные локальные переменные

18.12.2021 • 6 minutes to read

Начиная с версии 7.0 язык C# поддерживает значения, возвращаемые по ссылке (возвращаемые ссылочные значения). Возвращаемое ссылочное значение позволяет методу вернуть вызывающей стороне ссылку на переменную, а не фиксированное значение. После этого вызывающий может самостоятельно решить, как обрабатывать полученную переменную: по значению или по ссылке. Вызывающий может создать новую переменную и присвоить ей ссылку на полученное значение (локальная ссылочная переменная).

Что представляет собой возвращаемое ссылочное значение?

Большинство разработчиков прекрасно знакомы с передачей аргумента в вызываемый метод *по ссылке*. Список аргументов вызываемого метода включает переменную, переданную по ссылке. Вызывающий может отследить любые изменения этого значения в вызываемом методе. *Возвращаемое ссылочное значение* означает, что метод возвращает *ссылку* на некоторую переменную (или ее псевдоним). В область действия переменной должен входить этот метод. Время существования переменной должно продолжаться после того, как метод возвращает управление. Все изменения, которые вызывающий производит с возвращаемым значением метода, применяются к возвращенной переменной.

Если для метода объявлено *возвращаемое ссылочное значение*, значит он возвращает псевдоним переменной. Чаще всего это нужно для того, чтобы вызывающий код получил псевдоним для доступа к этой переменной, в том числе для ее изменения. Следовательно, методы с возвращаемыми ссылочными значениями не могут иметь тип возвращаемого значения `void`.

Есть несколько ограничений для выражений, которые можно использовать в качестве возвращаемых ссылочных значений метода. К ним относятся указанные ниже ограничения.

- Время существования возвращаемого значения должно превышать период выполнения метода. Другими словами, нельзя возвращать ссылку на локальную переменную вызываемого метода. Это может быть экземпляр статического поля или класса, а также переданный в метод аргумент. При попытке возвратить локальную переменную возникает ошибка компилятора CS8168 "Невозможно вернуть по ссылке локальный "obj", так как это не локальная переменная ref".

- Возвращаемое значение не может быть литералом `null`. При возврате `null` возникает ошибка компилятора CS8156 "Выражение невозможно использовать в данном контексте, так как его невозможно вернуть по ссылке".

  Метод с возвращаемым ссылочным значением может возвращать псевдоним для переменной с текущим значением NULL (которой не присвоены экземпляры) или тип, допускающий значение NULL.

- Возвращаемое значение не может быть константой, элементом перечисления, полученным по значению, возвращаемым значением свойства, а также методом `class` или `struct`. При нарушении этого правила возникает ошибка компилятора CS8156 "Выражение невозможно использовать в данном контексте, так как его невозможно вернуть по ссылке".

Кроме того, возвращаемые ссылочные значения недопустимы в асинхронных методах. Асинхронный метод может вернуть управление до того, как будет завершено его выполнение и станет известно его возвращаемое значение.

Определение возвращаемого ссылочного значения

Метод, который возвращает *возвращаемое значение ссылки*, должен удовлетворять следующим двум условиям:

- Сигнатура метода включает ключевое слово ref перед типом возвращаемого значения.
- Каждый оператор return в теле метода включает ключевое слово ref перед именем возвращаемого экземпляра.

В следующем примере показан метод, который удовлетворяет указанным условиям и возвращает ссылку на объект `Person` с именем `p`:

```
public ref Person GetContactInformation(string fname, string lname)
{
    // ...method implementation...
    return ref p;
}
```

Использование возвращаемого ссылочного значения

Возвращаемое ссылочное значение является псевдонимом для другой переменной в области вызываемого метода. Любое применение возвращаемого ссылочного значения можно рассматривать как применение псевдонима соответствующей переменной.

- Присваивая новое значение псевдониму, вы присваиваете это значение переменной, на которую он ссылается.
- Считывая значение псевдонима, вы получаете значение переменной, на которую он ссылается.
- Возвращая псевдоним *по ссылке*, вы возвращаете новый псевдоним для той же переменной.
- Передавая псевдоним другому методу *по ссылке*, вы передаете ссылку на переменную, на которую ссылается псевдоним.
- Создавая для псевдонима локальную ссылочную переменную, вы создаете новый псевдоним для той же переменной.

Ссылочные локальные переменные

Предположим, что для метода `GetContactInformation` объявлено возвращаемое ссылочное значение:

```
public ref Person GetContactInformation(string fname, string lname)
```

При назначении по значению считывается значение переменной и присваивается это значение новой переменной:

```
Person p = contacts.GetContactInformation("Brandie", "Best");
```

В предыдущем назначении `p` объявлена как локальная переменная. Исходное значение копируется из значения, которое возвращает `GetContactInformation`. Все последующие назначения `p` не затронут значения переменной, которую возвращает `GetContactInformation`. Переменная `p` теперь не является псевдонимом для возвращаемой переменной.

Вы объявляете *локальную ссылочную переменную* и сохраняете в ней псевдоним для исходного значения. В следующем назначении `p` является псевдонимом для переменной, возвращаемой из `GetContactInformation`.

```
ref Person p = ref contacts.GetContactInformation("Brandie", "Best");
```

При дальнейшем применении `p` действует так же, как и возвращаемая из `GetContactInformation` переменная, так как `p` является псевдонимом для этой переменной. Любые изменения `p` затрагивают и переменную, возвращаемую из `GetContactInformation`.

Ключевое слово `ref` используется перед объявлением локальной переменной *и* до вызова метода.

Вы можете таким же образом обратиться к значению по ссылке. В некоторых случаях обращение к значению по ссылке повышает производительность, поскольку позволяет избежать потенциально затратной операции копирования. Например, в следующей инструкции показано, как можно определить локальное ссылочное значение, используемое для ссылки на значение.

```
ref VeryLargeStruct reflocal = ref veryLargeStruct;
```

Ключевое слово `ref` используется перед объявлением локальной переменной *и* перед значением во втором примере. Если не указать ключевые слова `ref` одновременно в объявлении и присвоении переменной в обоих примерах, возникает ошибка компилятора CS8172 "Невозможно инициализировать значением переменную по ссылке".

До версии C# 7.3 ссылочные локальные переменные нельзя было переназначить после инициализации так, чтобы они ссылались на другое хранилище. Это ограничение было снято. В следующем примере демонстрируется переназначение:

```
ref VeryLargeStruct reflocal = ref veryLargeStruct; // initialization
refLocal = ref anotherVeryLargeStruct; // reassigned, refLocal refers to different storage.
```

Ссылочные локальные переменные по-прежнему необходимо инициализировать во время объявления.

Пример использования возвращаемых ссылочных значений и ссылочных локальных переменных

В следующем примере определяется класс `NumberStore`, в котором хранится массив целочисленных значений. Метод `FindNumber` возвращает по ссылке первое число, которое не меньше переданного в аргументе значения. Если такое число не найдено, метод возвращает число с индексом 0.

```
using System;

class NumberStore
{
    int[] numbers = { 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023 };

    public ref int FindNumber(int target)
    {
        for (int ctr = 0; ctr < numbers.Length; ctr++)
        {
            if (numbers[ctr] >= target)
                return ref numbers[ctr];
        }
        return ref numbers[0];
    }

    public override string ToString() => string.Join(" ", numbers);
}
```

В следующем примере вызывается метод `NumberStore.FindNumber`, который извлекает первое значение не меньше 16. После этого вызывающий объект удваивает значение, возвращаемое методом. Как видно из выходных данных этого примера, изменение отражается в значении элементов массива в экземпляре `NumberStore`.

```
var store = new NumberStore();
Console.WriteLine($"Original sequence: {store.ToString()}");
int number = 16;
ref var value = ref store.FindNumber(number);
value *= 2;
Console.WriteLine($"New sequence:      {store.ToString()}");
// The example displays the following output:
//       Original sequence: 1 3 7 15 31 63 127 255 511 1023
//       New sequence:      1 3 7 15 62 63 127 255 511 1023
```

Без использования возвращаемых ссылочных значений такая операция выполняется путем возврата индекса элемента массива вместе с его значением. После этого вызывающий объект использует индекс для изменения значения в отдельном вызове метода. Тем не менее вызывающий объект также может изменить индекс для доступа к другим значениям массива и их изменения.

В следующем примере показано, как можно изменить метод `FindNumber` в версии C# 7.3 или более поздней, чтобы использовать переназначение ссылочной локальной переменной:

```
using System;

class NumberStore
{
    int[] numbers = { 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023 };

    public ref int FindNumber(int target)
    {
        ref int returnVal = ref numbers[0];
        var ctr = numbers.Length - 1;
        while ((ctr >= 0) && (numbers[ctr] >= target))
        {
            returnVal = ref numbers[ctr];
            ctr--;
        }
        return ref returnVal;
    }

    public override string ToString() => string.Join(" ", numbers);
}
```

Эта вторая версия работает эффективнее с более длинными последовательностями в сценариях, в которых искомое число находится ближе к концу массива, так как итерация массива выполняется от конца к началу, что приводит к проверке меньшего числа элементов.

См. также

- Ключевое слово ref
- Написание безопасного и эффективного кода

Передача параметров (Руководство по программированию в C#)

18.12.2021 • 2 minutes to read

В C# аргументы могут передаваться параметрам либо по значению, либо по ссылке. Передача по ссылке позволяет изменять и сохранять измененные значения параметров членов функций, методов, свойств, индексаторов, операторов и конструкторов в вызывающей среде. Чтобы передать параметр по ссылке, намереваясь изменить значение, используйте ключевое слово `ref` или `out`. Чтобы передать по ссылке, намереваясь предотвратить копирование, но не изменение значения, используйте модификатор `in`. Для простоты в этих примерах используется только ключевое слово `ref`. Дополнительные сведения о различиях между ключевыми словами `in`, `ref` и `out` см. в разделах in, ref, out.

В приведенном ниже примере показано различие между параметрами-значениями и ссылочными параметрами.

```
class Program
{
    static void Main(string[] args)
    {
        int arg;

        // Passing by value.
        // The value of arg in Main is not changed.
        arg = 4;
        squareVal(arg);
        Console.WriteLine(arg);
        // Output: 4

        // Passing by reference.
        // The value of arg in Main is changed.
        arg = 4;
        squareRef(ref arg);
        Console.WriteLine(arg);
        // Output: 16
    }

    static void squareVal(int valParameter)
    {
        valParameter *= valParameter;
    }

    // Passing by reference
    static void squareRef(ref int refParameter)
    {
        refParameter *= refParameter;
    }
}
```

Дополнительные сведения см. в следующих разделах:

- Передача параметров типа значения
- Передача параметров ссылочного типа

Спецификация языка C#

Дополнительные сведения см. в разделе Список аргументов в Спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Руководство по программированию на C#
- Методы

Передача параметров типа значения (Руководство по программированию в C#)

18.12.2021 • 3 minutes to read

Переменная типа значения напрямую содержит данные, в отличие от переменной ссылочного типа, которая содержит только ссылку на данные. Передавая в метод переменную типа значения, вы передаете ему копию этой переменной. Любые изменения параметра, которые происходят внутри метода, не влияют на исходные данные, хранимые в переменной. Если вы хотите, чтобы вызываемый метод изменял значение аргумента, его необходимо передать по ссылке с помощью ключевых слов ref или out. Можно также использовать ключевое слово in для передачи параметра значения по ссылке, чтобы избежать копирования и гарантировать неизменность значения. В следующих примерах мы для простоты используем `ref`.

Передача переменных типа значения по значению

Следующий пример демонстрирует передачу параметров типа значения по значению. Переменная `n` передается по значению в метод `SquareIt`. Любые изменения, выполненные внутри метода, не влияют на исходное значение переменной.

```
class PassingValByVal
{
    static void SquareIt(int x)
    // The parameter x is passed by value.
    // Changes to x will not affect the original value of x.
    {
        x *= x;
        System.Console.WriteLine("The value inside the method: {0}", x);
    }
    static void Main()
    {
        int n = 5;
        System.Console.WriteLine("The value before calling the method: {0}", n);

        SquareIt(n);  // Passing the variable by value.
        System.Console.WriteLine("The value after calling the method: {0}", n);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    The value before calling the method: 5
    The value inside the method: 25
    The value after calling the method: 5
*/
```

Переменная `n` имеет тип значения. Она содержит данные, в нашем примере это значение `5`. Когда вызывается `SquareIt`, содержимое `n` копируется в параметр `x`, который используется исключительно внутри метода. Но в `Main` значение `n` всегда останется прежним после выполнения метода `SquareIt`. Изменения, выполненные внутри метода, влияют только на локальную переменную `x`.

Передача переменных типа значения по ссылке

Следующий пример полностью совпадает с предыдущим, но теперь аргумент передается как параметр `ref`. Значение базового аргумента `n` будет изменяться, когда метод изменяет значение `x`.

```
class PassingValByRef
{
    static void SquareIt(ref int x)
    // The parameter x is passed by reference.
    // Changes to x will affect the original value of x.
    {
        x *= x;
        System.Console.WriteLine("The value inside the method: {0}", x);
    }
    static void Main()
    {
        int n = 5;
        System.Console.WriteLine("The value before calling the method: {0}", n);

        SquareIt(ref n);  // Passing the variable by reference.
        System.Console.WriteLine("The value after calling the method: {0}", n);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    The value before calling the method: 5
    The value inside the method: 25
    The value after calling the method: 25
*/
```

В этом примере передается не значение `n`, а ссылка на `n`. Теперь параметр `x` будет не значением типа int, а ссылкой на `int`. В нашем примере это ссылка на `n`. Таким образом, при передаче `x` в метод передается только информация о том, что `x` ссылается на `n`.

Изменение типов значений

Распространенным примером изменения значений аргументов является метод замены, при котором в метод передаются две переменные, а метод меняет местами их содержимое. Для метода замены аргументы необходимо передавать по ссылке. В противном случае замена затронет лишь локальные копии параметров в этом методе, но в вызывающем методе не произойдет никаких изменений. В следующем примере меняются местами целочисленные значения.

```
static void SwapByRef(ref int x, ref int y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

При вызове метода `SwapByRef` используйте ключевое слово `ref`, как показано в следующем примере.

```
static void Main()
{
    int i = 2, j = 3;
    System.Console.WriteLine("i = {0}  j = {1}" , i, j);

    SwapByRef (ref i, ref j);

    System.Console.WriteLine("i = {0}  j = {1}" , i, j);

    // Keep the console window open in debug mode.
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}
/* Output:
    i = 2  j = 3
    i = 3  j = 2
*/
```

См. также

- Руководство по программированию на C#
- Передача параметров
- Передача параметров ссылочного типа

Передача параметров ссылочного типа (Руководство по программированию в C#)

18.12.2021 • 3 minutes to read

Переменная ссылочного типа содержит не сами данные, а ссылку на них. При передаче параметра ссылочного типа по значению можно изменять данные, относящиеся к объекту, на который указывает ссылка, например, значение члена класса. Тем не менее вы не можете изменить значение самой ссылки. Например, вы не можете использовать одну и ту же ссылку, чтобы выделить память для нового объекта и сохранить его вне этого метода. Для этого необходимо передать параметр с использованием ключевого слова ref или out. В следующих примерах мы для простоты используем `ref`.

Передача ссылочных типов по значению

В следующем примере демонстрируется передача параметра ссылочного типа `arr` по значению в метод `Change`. Поскольку этот параметр является ссылкой на `arr`, можно изменять элементы массива. Тем не менее переназначение параметра другому блоку памяти возможно только внутри самого метода и не влияет на исходную переменную `arr`.

```
class PassingRefByVal
{
    static void Change(int[] pArray)
    {
        pArray[0] = 888;  // This change affects the original element.
        pArray = new int[5] {-3, -1, -2, -3, -4};   // This change is local.
        System.Console.WriteLine("Inside the method, the first element is: {0}", pArray[0]);
    }

    static void Main()
    {
        int[] arr = {1, 4, 5};
        System.Console.WriteLine("Inside Main, before calling the method, the first element is: {0}", arr
[0]);

        Change(arr);
        System.Console.WriteLine("Inside Main, after calling the method, the first element is: {0}", arr
[0]);
    }
}
/* Output:
    Inside Main, before calling the method, the first element is: 1
    Inside the method, the first element is: -3
    Inside Main, after calling the method, the first element is: 888
*/
```

В предыдущем примере массив `arr` имеет ссылочный тип и передается в метод без параметра `ref`. В таком случае в метод будет передана копия ссылки, которая указывает на `arr`. Как видно из выходных данных, метод может изменять содержимое элемента массива (в данном случае с `1` на `888`). Тем не менее при выделении нового блока памяти с помощью оператора new внутри метода `Change` переменная `pArray` будет ссылаться на новый массив. Таким образом, любые выполненные после этого изменения не будут отражаться в исходном массиве `arr`, который был создан внутри `Main`. Фактически, в этом примере создается два массива: один внутри `Main`, а другой — в методе `Change`.

Передача ссылочных типов по ссылке

Следующий пример аналогичен предыдущему, однако в нем в заголовок и вызов метода добавляется ключевое слово `ref`. Любые изменения, выполняемые в методе, влияют на исходную переменную в вызывающей программе.

```
class PassingRefByRef
{
    static void Change(ref int[] pArray)
    {
        // Both of the following changes will affect the original variables:
        pArray[0] = 888;
        pArray = new int[5] {-3, -1, -2, -3, -4};
        System.Console.WriteLine("Inside the method, the first element is: {0}", pArray[0]);
    }

    static void Main()
    {
        int[] arr = {1, 4, 5};
        System.Console.WriteLine("Inside Main, before calling the method, the first element is: {0}",
arr[0]);

        Change(ref arr);
        System.Console.WriteLine("Inside Main, after calling the method, the first element is: {0}",
arr[0]);
    }
}
/* Output:
    Inside Main, before calling the method, the first element is: 1
    Inside the method, the first element is: -3
    Inside Main, after calling the method, the first element is: -3
*/
```

Все изменения, выполняемые внутри метода, влияют на исходный массив в `Main`. Фактически, происходит перемещение исходного массива с помощью оператора `new`. Таким образом, после вызова метода `Change` любые ссылки на `arr` будут указывать на массив из пяти элементов, который создается в методе `Change`.

Перестановка двух строк

Перестановка строк представляет собой наглядный пример передачи параметров ссылочного типа по ссылке. В этом примере две строки (`str1` и `str2`) инициализируются в `Main`, а затем передаются в метод `SwapStrings` в качестве параметров, измененных по ключевому слову `ref`. Перестановка двух строк выполняется как в этом методе, так и в `Main`.

```
class SwappingStrings
{
    static void SwapStrings(ref string s1, ref string s2)
    // The string parameter is passed by reference.
    // Any changes on parameters will affect the original variables.
    {
        string temp = s1;
        s1 = s2;
        s2 = temp;
        System.Console.WriteLine("Inside the method: {0} {1}", s1, s2);
    }

    static void Main()
    {
        string str1 = "John";
        string str2 = "Smith";
        System.Console.WriteLine("Inside Main, before swapping: {0} {1}", str1, str2);

        SwapStrings(ref str1, ref str2);   // Passing strings by reference
        System.Console.WriteLine("Inside Main, after swapping: {0} {1}", str1, str2);
    }
}
/* Output:
    Inside Main, before swapping: John Smith
    Inside the method: Smith John
    Inside Main, after swapping: Smith John
*/
```

В этом примере параметры должны передаваться по ссылке, чтобы обеспечить изменение переменных в вызывающей программе. Если удалить ключевое слово `ref` из заголовка и вызова метода, в вызывающей программе не будут выполнены никакие изменения.

Дополнительные сведения о строках см. в этом разделе.

См. также

- Руководство по программированию на C#
- Передача параметров
- ref
- in
- out
- Ссылочные типы

Практическое руководство. Определение различия между передачей структуры и ссылки класса в метод (руководство по программированию на C#)

18.12.2021 • 2 minutes to read

В следующем примере демонстрируется, чем передача структуры в метод отличается от передачи экземпляра класса в метод. В этом примере оба аргумента (структура и экземпляр класса) передаются по значению, и оба метода изменяют значение одного поля аргумента. Тем не менее результаты этих двух методов будут отличаться, поскольку в случае со структурой передаваемое содержимое отличается от передачи экземпляра класса.

Поскольку структура является типом значения, при передаче структуры по значению в метод этот метод получает и обрабатывает копию аргумента структуры. При этом метод не имеет доступа к исходной структуре в вызывающем методе и, соответственно, никак не может изменить ее. В этом случае метод может изменять только копию.

Экземпляр класса является ссылочным типом, а не типом значения. При передаче ссылочного типа по значению в метод этот метод получает копию ссылки на экземпляр класса. Таким образом, вызванный метод получает копию адреса экземпляра, а вызывающий метод сохраняет исходный адрес экземпляра. Экземпляр класса в вызывающем методе содержит адрес, а параметр в вызываемом методе — копию этого адреса, которая ссылается на тот же объект. Поскольку параметр содержит только копию адреса, вызываемый метод не может изменить адрес экземпляра класса в вызывающем методе. Тем не менее вызываемый метод может использовать копию адреса, чтобы обращаться к членам класса, на которые ссылается как исходный адрес, так и копия адреса. Если вызываемый метод изменяет член класса, также изменится исходный экземпляр класса в вызывающем методе.

Различия демонстрируются в выходных данных следующего примера. Значение поля `willIChange` экземпляра класса изменяется в результате вызова метода `ClassTaker`, поскольку этот метод находит указанное поле экземпляра класса по адресу, содержащемуся в параметре. Поле `willIChange` структуры в вызывающем методе не изменяется в результате вызова метода `StructTaker`, поскольку значением аргумента является копия самой структуры, а не ее адреса. `StructTaker` изменяет саму копию, которая будет утрачена после завершения вызова `StructTaker`.

Пример

```
using System;

class TheClass
{
    public string willIChange;
}

struct TheStruct
{
    public string willIChange;
}

class TestClassAndStruct
{
    static void ClassTaker(TheClass c)
    {
        c.willIChange = "Changed";
    }

    static void StructTaker(TheStruct s)
    {
        s.willIChange = "Changed";
    }

    static void Main()
    {
        TheClass testClass = new TheClass();
        TheStruct testStruct = new TheStruct();

        testClass.willIChange = "Not Changed";
        testStruct.willIChange = "Not Changed";

        ClassTaker(testClass);
        StructTaker(testStruct);

        Console.WriteLine("Class field = {0}", testClass.willIChange);
        Console.WriteLine("Struct field = {0}", testStruct.willIChange);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Class field = Changed
    Struct field = Not Changed
*/
```

См. также

- Руководство по программированию на C#
- Классы
- Типы структур
- Передача параметров

Неявно типизированные локальные переменные (руководство по программированию на C#)

18.12.2021 • 4 minutes to read

Локальные переменные можно объявлять без указания конкретного типа. Ключевое слово `var` указывает, что компилятор должен вывести тип переменной из выражения справа от оператора инициализации. Выведенный тип может быть встроенным, анонимным, определяемым пользователем либо типом, определяемым в библиотеке классов .NET. Дополнительные сведения об инициализации массивов с `var` см. в разделе Неявно типизированные массивы.

В приведенных ниже примерах показаны различные способы объявления локальных переменных с помощью `var`:

```
// i is compiled as an int
var i = 5;

// s is compiled as a string
var s = "Hello";

// a is compiled as int[]
var a = new[] { 0, 1, 2 };

// expr is compiled as IEnumerable<Customer>
// or perhaps IQueryable<Customer>
var expr =
    from c in customers
    where c.City == "London"
    select c;

// anon is compiled as an anonymous type
var anon = new { Name = "Terry", Age = 34 };

// list is compiled as List<int>
var list = new List<int>();
```

Важно понимать, что ключевое слово `var` не означает "variant" и не означает, что переменная является слабо типизированной или имеет позднее связывание. Он указывает только на то, что компилятор определяет и назначает наиболее подходящий тип.

Ключевое слово `var` можно использовать в следующих контекстах:

- С локальными переменными (переменными, объявленными в области метода), как показано в предыдущем примере.
- В операторе инициализации for.

  ```
  for (var x = 1; x < 10; x++)
  ```

- В операторе инициализации foreach.

  ```
  foreach (var item in list) {...}
  ```

- В операторе using.

```
using (var file = new StreamReader("C:\\myfile.txt")) {...}
```

Дополнительные сведения см. в разделе Практическое руководство. Использование неявно типизированных локальных переменных и массивов в выражении запроса.

Переменная var и анонимные типы

Во многих случаях переменная `var` не является обязательной и предназначена только для синтаксического удобства. Тем не менее если переменная инициализируется с помощью анонимного типа и вам потребуется доступ к свойствам объекта на более позднем этапе, ее необходимо объявить как `var`. Это распространенный сценарий в выражениях запросов LINQ. Дополнительные сведения см. в статье Анонимные типы.

С точки зрения исходного кода анонимный тип безымянен. Таким образом, если переменная запроса инициализирована с помощью `var`, то единственный способ получить доступ к свойствам в возвращаемой последовательности объектов — это использовать `var` как тип переменной итерации в операторе `foreach`.

```
class ImplicitlyTypedLocals2
{
    static void Main()
    {
        string[] words = { "aPPLE", "BlUeBeRrY", "cHeRry" };

        // If a query produces a sequence of anonymous types,
        // then use var in the foreach statement to access the properties.
        var upperLowerWords =
             from w in words
             select new { Upper = w.ToUpper(), Lower = w.ToLower() };

        // Execute the query
        foreach (var ul in upperLowerWords)
        {
            Console.WriteLine("Uppercase: {0}, Lowercase: {1}", ul.Upper, ul.Lower);
        }
    }
}
/* Outputs:
    Uppercase: APPLE, Lowercase: apple
    Uppercase: BLUEBERRY, Lowercase: blueberry
    Uppercase: CHERRY, Lowercase: cherry
 */
```

Примечания

В объявлениям неявно типизированных переменных применяются следующие ограничения:

- `var` можно использовать только в том случае, если локальная переменная объявляется и инициализируется в одном и том же операторе; переменная не может инициализироваться в нулевое значение, в группу методов или в анонимную функцию.
- `var` нельзя применять к полям в области видимости класса.
- Переменные, объявленные с помощью `var`, нельзя использовать в выражении инициализации. Другими словами, это выражение является допустимым выражением: `int i = (i = 20);`, но вызывает ошибку времени компиляции: `var i = (i = 20);`
- Инициализировать сразу несколько неявно типизированных переменных в одном и том же

операторе нельзя.

- Если тип с именем `var` входит в область видимости, то ключевое слово `var` разрешится в это имя типа и не будет обрабатываться как часть объявления неявно типизированной локальной переменной.

Неявное типизирование с ключевым словом `var` может применяться только к переменным в области локального метода. Неявное типизирование недоступно для полей класса C#, так как при обработке кода компилятор столкнется с логическим парадоксом: компилятор должен знать тип поля, но он не может определить тип, пока не проанализирует выражение присваивания, и не может вычислить выражение, не зная тип. Рассмотрим следующий код.

```
private var bookTitles;
```

`bookTitles` — это поле класса, которому присваивается тип `var`. Так как поле не имеет выражения для оценки, то компилятор не сможет вывести тип `bookTitles`. Кроме того, добавления выражения в поле (так же, как для локальной переменной) тоже недостаточно:

```
private var bookTitles = new List<string>();
```

Когда компилятор обнаруживает поля во время компиляции кода, он записывает тип каждого поля перед обработкой любого выражения, связанного с ним. Компилятор обнаруживает тот же парадокс при попытке анализа `bookTitles`: он должен знать тип поля, но обычно он определяет тип `var` путем анализа выражения, который невозможно определить, если заранее не знать тип.

Переменную `var` можно также использовать в выражениях запросов, где точный сконструированный тип переменной запроса определить непросто. Подобная ситуация может возникнуть в операциях группировки и сортировки.

Кроме того, ключевое слово `var` может пригодиться, если конкретный тип переменной сложно набрать с клавиатуры, а также если он очевиден либо затрудняет чтение кода. Использовать `var` таким образом можно, например, с вложенными универсальными типами — подобные типы применяются в групповых операциях. В следующем запросе переменная запроса имеет тип `IEnumerable<IGrouping<string, Student>>`. Если вы и другие пользователи, которые работают с кодом, это понимаете, использовать неявный ввод для удобства и краткости кода можно без проблем.

```
// Same as previous example except we use the entire last name as a key.
// Query variable is an IEnumerable<IGrouping<string, Student>>
var studentQuery3 =
    from student in students
    group student by student.Last;
```

Ключевое слово `var` позволяет упростить код, однако его следует использовать только там, где оно действительно необходимо или где оно облегчает понимание кода. Дополнительные сведения о том, когда следует использовать ключевое слово `var`, см. в разделе Неявно типизированные локальные переменные в статье с рекомендациями по написанию кода C#.

См. также

- Справочник по C#
- Неявно типизированные массивы
- Практическое руководство. Использование неявно типизированных локальных переменных и массивов в выражении запроса

- Анонимные типы
- Инициализаторы объектов и коллекций
- var
- LINQ в C#
- Встроенный язык запросов LINQ
- Инструкции итерации
- Оператор using

Практическое руководство. Использование неявно типизированных локальных переменных и массивов в выражении запроса (руководство по программированию на C#)

18.12.2021 • 2 minutes to read

Неявно типизированные локальные переменные можно использовать в тех случаях, когда требуется, чтобы компилятор определял тип локальной переменной. Неявно типизированные локальные переменные необходимо использовать для хранения анонимных типов, которые часто используются в выражениях запроса. В следующих примерах демонстрируется обязательное и необязательное использование неявно типизированных локальных переменных в запросах.

Неявно типизированные локальные переменные объявляются с помощью контекстного ключевого слова var. Дополнительные сведения см. в разделах Неявно типизированные локальные переменные и Неявно типизированные массивы.

Примеры

В следующем примере показан общий сценарий, в котором использование ключевого слова `var` является обязательным: выражение запроса, создающее последовательность анонимных типов. В этом случае переменная запроса и переменная итерации в операторе `foreach` должны быть неявно типизированы с помощью ключевого слова `var`, поскольку доступ к имени анонимного типа отсутствует. Дополнительные сведения об анонимных типах см. в разделе Анонимные типы.

```
private static void QueryNames(char firstLetter)
{
    // Create the query. Use of var is required because
    // the query produces a sequence of anonymous types:
    // System.Collections.Generic.IEnumerable<????>.
    var studentQuery =
        from student in students
        where student.FirstName[0] == firstLetter
        select new { student.FirstName, student.LastName };

    // Execute the query and display the results.
    foreach (var anonType in studentQuery)
    {
        Console.WriteLine("First = {0}, Last = {1}", anonType.FirstName, anonType.LastName);
    }
}
```

В следующем примере ключевое слово `var` используется в схожей ситуации, однако при этом ключевое слово `var` является необязательным. Поскольку `student.LastName` является строкой, при выполнении запроса возвращается последовательность строк. Таким образом, `queryID` можно объявить как `System.Collections.Generic.IEnumerable<string>` вместо `var`. Ключевое слово `var` используется для удобства. В этом примере переменная итерации в операторе `foreach` явно типизируется как строка, но вместо этого может быть объявлена с помощью ключевого слова `var`. Так как тип переменной итерации не является анонимным, использование ключевого слова `var` возможно, но не обязательно. Обратите внимание, что само по себе ключевое слово `var` является не типом, а инструкцией, предписывающей

компилятору определение и присвоение типа.

```
// Variable queryId could be declared by using
// System.Collections.Generic.IEnumerable<string>
// instead of var.
var queryId =
    from student in students
    where student.Id > 111
    select student.LastName;

// Variable str could be declared by using var instead of string.
foreach (string str in queryId)
{
    Console.WriteLine("Last name: {0}", str);
}
```

См. также

- Руководство по программированию на C#
- Методы расширения
- Встроенный язык запросов LINQ
- var
- LINQ в C#

Методы расширения (Руководство по программированию в C#)

18.12.2021 • 9 minutes to read

Методы расширения позволяют "добавлять" методы в существующие типы без создания нового производного типа, перекомпиляции и иного изменения первоначального типа. Методы расширения представляют собой разновидность статического метода, но вызываются так же, как методы экземпляра в расширенном типе. Для клиентского кода, написанного на языках C#, F# и Visual Basic, нет видимого различия между вызовом метода расширения и вызовом методов, определенных в типе.

Самые распространенные методы расширения — стандартные операторы запросов LINQ, которые добавляют функции запросов в существующие типы System.Collections.IEnumerable и System.Collections.Generic.IEnumerable<T>. Для использования стандартных операторов запросов их необходимо ввести в область действия с помощью директивы `using System.Linq`. Тогда каждый тип, реализующий тип IEnumerable<T>, будет иметь методы экземпляра, в частности GroupBy, OrderBy, Average и т. д. Эти дополнительные методы можно видеть в завершении операторов IntelliSense при вводе точки после экземпляра типа IEnumerable<T>, например List<T> или Array.

Пример OrderBy

В следующем примере показано, как вызывать метод стандартного оператора запроса `OrderBy` для массива целых чисел. Выражение в скобках называется лямбда-выражением. Многие стандартные операторы запроса принимают лямбда-выражения в качестве параметров, но это необязательно для методов расширения. Дополнительные сведения см. в разделе Лямбда-выражения.

```
class ExtensionMethods2
{

    static void Main()
    {
        int[] ints = { 10, 45, 15, 39, 21, 26 };
        var result = ints.OrderBy(g => g);
        foreach (var i in result)
        {
            System.Console.Write(i + " ");
        }
    }
}
//Output: 10 15 21 26 39 45
```

Методы расширения определяются как статические методы, но вызываются с помощью синтаксиса обращения к методу экземпляра. Их первый параметр определяет, с каким типом оперирует метод. Параметру предшествует модификатор this. Методы расширения находятся в области действия, только если пространство имен было явно импортировано в исходный код с помощью директивы `using`.

В приведенном ниже примере показан метод расширения, определенный для класса System.String. Этот метод определяется внутри невложенного, неуниверсального статического класса:

```
namespace ExtensionMethods
{
    public static class MyExtensions
    {
        public static int WordCount(this String str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
                             StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
```

Метод расширения `WordCount` можно ввести в область действия с помощью следующей директивы `using`:

```
using ExtensionMethods;
```

Его можно вызвать из приложения с помощью следующего синтаксиса:

```
string s = "Hello Extension Methods";
int i = s.WordCount();
```

В созданном коде метод расширения вызывается с помощью синтаксиса обращения к методу экземпляра. Промежуточный язык (IL), создаваемый компилятором, преобразует код в вызов статического метода. Принцип инкапсуляции фактически не нарушается. Методы расширения не могут получать доступ к частным переменным типа, для расширения которого они используются.

Класс `MyExtensions` и метод `WordCount` являются `static`, и доступ к ним можно получить так же, как ко всем остальным элементам `static`. Метод `WordCount` может быть вызван так же, как другие методы `static`, следующим образом:

```
string s = "Hello Extension Methods";
int i = MyExtensions.WordCount(s);
```

В приведенном выше коде C#:

- Объявляет и назначает новый `string` с именем `s` и значением `"Hello Extension Methods"`.
- Вызывает `MyExtensions.WordCount` с учетом аргумента `s`

Дополнительные сведения см. в разделе Практическое руководство. Реализация и вызов пользовательского метода расширения.

Вообще, обычно гораздо чаще вызываются методы расширения, чем реализуются собственные методы. Так как методы расширения вызываются с помощью синтаксиса обращения к методу экземпляра, для использования их из клиентского кода специальные знания не требуются. Чтобы включить методы расширения для определенного типа, необходимо просто добавить директиву `using` для пространства имен, в котором эти методы определяются. Например, чтобы использовать стандартные операторы запроса, нужно добавить в код следующую директиву `using`:

```
using System.Linq;
```

(Также может потребоваться добавить ссылку на библиотеку System.Core.dll.) Обратите внимание, что стандартные операторы запроса теперь появляются в IntelliSense в виде дополнительных методов, доступных для большинства типов IEnumerable<T>.

Привязка методов расширения во время компиляции

Методы расширения можно использовать для расширения класса или интерфейса, но не для их переопределения. Метод расширения, имеющий те же имя и сигнатуру, что и интерфейс или метод класса, никогда не вызывается. Во время компиляции методы расширения всегда имеют более низкий приоритет, чем методы экземпляра, определенные в самом типе. Другими словами, если тип имеет метод `Process(int i)`, а также есть метод расширения с такой же сигнатурой, компилятор будет всегда выполнять привязку к методу экземпляра. Если компилятор обнаруживает вызов метода, он сначала ищет совпадения с методами экземпляра типа. Если такое совпадение не найдено, компилятор выполняет поиск методов расширения, определенных для соответствующего типа, и создает привязку к первому обнаруженному методу расширения. В следующем примере кода демонстрируется, как компилятор определяет, к какому методу расширения или методу экземпляра необходимо выполнить привязку.

Пример

В следующем примере демонстрируются правила, которые компилятор C# соблюдает при определении того, к чему необходимо привязать вызов метода — к методу экземпляра типа или к методу расширения. Статический класс `Extensions` содержит методы расширения, определяемые для любого типа, реализующего интерфейс `IMyInterface`. Все три класса — `A`, `B` и `C` — реализуют этот интерфейс.

Метод расширения `MethodB` никогда не вызывается, потому что его имя и сигнатура точно совпадают с методами, уже реализованными этими классами.

Если компилятор не может найти метод экземпляра с совпадающей сигнатурой, он выполняет привязку к совпадающему методу расширения, если такой существует.

```
// Define an interface named IMyInterface.
namespace DefineIMyInterface
{
    using System;

    public interface IMyInterface
    {
        // Any class that implements IMyInterface must define a method
        // that matches the following signature.
        void MethodB();
    }
}

// Define extension methods for IMyInterface.
namespace Extensions
{
    using System;
    using DefineIMyInterface;

    // The following extension methods can be accessed by instances of any
    // class that implements IMyInterface.
    public static class Extension
    {
        public static void MethodA(this IMyInterface myInterface, int i)
        {
            Console.WriteLine
                ("Extension.MethodA(this IMyInterface myInterface, int i)");
        }

        public static void MethodA(this IMyInterface myInterface, string s)
        {
            Console.WriteLine
                ("Extension.MethodA(this IMyInterface myInterface, string s)");
        }
```

```
        // This method is never called in ExtensionMethodsDemo1, because each
        // of the three classes A, B, and C implements a method named MethodB
        // that has a matching signature.
        public static void MethodB(this IMyInterface myInterface)
        {
            Console.WriteLine
                ("Extension.MethodB(this IMyInterface myInterface)");
        }
    }
}

// Define three classes that implement IMyInterface, and then use them to test
// the extension methods.
namespace ExtensionMethodsDemo1
{
    using System;
    using Extensions;
    using DefineIMyInterface;

    class A : IMyInterface
    {
        public void MethodB() { Console.WriteLine("A.MethodB()"); }
    }

    class B : IMyInterface
    {
        public void MethodB() { Console.WriteLine("B.MethodB()"); }
        public void MethodA(int i) { Console.WriteLine("B.MethodA(int i)"); }
    }

    class C : IMyInterface
    {
        public void MethodB() { Console.WriteLine("C.MethodB()"); }
        public void MethodA(object obj)
        {
            Console.WriteLine("C.MethodA(object obj)");
        }
    }

    class ExtMethodDemo
    {
        static void Main(string[] args)
        {
            // Declare an instance of class A, class B, and class C.
            A a = new A();
            B b = new B();
            C c = new C();

            // For a, b, and c, call the following methods:
            //      -- MethodA with an int argument
            //      -- MethodA with a string argument
            //      -- MethodB with no argument.

            // A contains no MethodA, so each call to MethodA resolves to
            // the extension method that has a matching signature.
            a.MethodA(1);           // Extension.MethodA(IMyInterface, int)
            a.MethodA("hello");     // Extension.MethodA(IMyInterface, string)

            // A has a method that matches the signature of the following call
            // to MethodB.
            a.MethodB();            // A.MethodB()

            // B has methods that match the signatures of the following
            // method calls.
            b.MethodA(1);           // B.MethodA(int)
            b.MethodB();            // B.MethodB()

            // B has no matching method for the following call, but
```

```
            // class Extension does.
            b.MethodA("hello");     // Extension.MethodA(IMyInterface, string)

            // C contains an instance method that matches each of the following
            // method calls.
            c.MethodA(1);           // C.MethodA(object)
            c.MethodA("hello");     // C.MethodA(object)
            c.MethodB();            // C.MethodB()
        }
    }
}
/* Output:
    Extension.MethodA(this IMyInterface myInterface, int i)
    Extension.MethodA(this IMyInterface myInterface, string s)
    A.MethodB()
    B.MethodA(int i)
    B.MethodB()
    Extension.MethodA(this IMyInterface myInterface, string s)
    C.MethodA(object obj)
    C.MethodA(object obj)
    C.MethodB()
 */
```

Общие варианты использования

Функциональные возможности коллекций

В прошлом было распространено создание классов коллекций, которые реализовали интерфейс System.Collections.Generic.IEnumerable<T> для данного типа и содержали функциональные возможности, действующие на коллекции этого типа. Хотя в создании этого типа объекта коллекции нет ничего плохого, те же функциональные возможности можно получить, используя расширение System.Collections.Generic.IEnumerable<T>. Преимущество расширений заключается в том, что они позволяют вызывать функциональные возможности из любой коллекции, например System.Array или System.Collections.Generic.List<T>, которая реализует System.Collections.Generic.IEnumerable<T> для этого типа. Пример использования массива Int32 описан ранее в этой статье.

Функциональные возможности конкретного слоя

При использовании многослойной архитектуры или другой многослойной модели приложения обычно используется набор сущностей предметной области или объектов передачи данных, которые можно использовать для обмена данными между границами приложений. Обычно эти объекты содержат только минимальные функциональные возможности (или вовсе их не содержат), применимые ко всем слоям приложения. Методы расширения можно использовать, чтобы добавить функциональные возможности для каждого конкретного слоя приложения без загрузки объекта с помощью методов, которые не нужны или не требуются для других слоев.

```
public class DomainEntity
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

static class DomainEntityExtensions
{
    static string FullName(this DomainEntity value)
        => $"{value.FirstName} {value.LastName}";
}
```

Расширение предопределенных типов

Если необходимо создать многократно используемые функциональные возможности, вы можете

расширить существующий тип, например тип .NET или CLR, чтобы не создавать дополнительные объекты. Например, если методы расширения не используются, можно создать класс `Engine` или `Query`, чтобы выполнить запрос к SQL Server, который можно вызвать из нескольких расположений в коде. Однако вместо этого можно расширить класс System.Data.SqlClient.SqlConnection с помощью методов расширения, чтобы выполнить этот запрос из любого расположения, где установлено подключение с SQL Server. В качестве второго примера можно привести добавление общих функциональных возможностей в класс System.String, расширение возможностей обработки данных объектов System.IO.File и System.IO.Stream, а также объектов System.Exception для функциональных возможностей обработки конкретных ошибок. Сценарии использования ограничиваются только воображением и здравым смыслом.

Расширение предопределенных типов с помощью типов `struct` может быть сложным, так как они передаются методам по значению. Это означает, что любые изменения структуры вносятся в ее копию. Эти изменения не отображаются после выхода из метода расширения. Начиная с C# 7.2, вы можете добавить модификатор `ref` к первому аргументу метода расширения. Добавление модификатора `ref` означает, что первый аргумент передается по ссылке. Это позволяет создавать методы расширения, изменяющие состояние расширяемой структуры.

Общие рекомендации

Хотя по-прежнему предпочтительнее добавлять функциональные возможности путем изменения кода объекта или создания производного типа, когда это целесообразно и возможно, методы расширения стали ключевым вариантом для создания многократно используемых функциональных возможностей во всей экосистеме .NET. Методы расширения также являются отличным выбором, если вы не управляете исходным источником, если производный объект недопустим или невозможен или если функциональные возможности должны быть недоступными за пределами применимой области.

Дополнительные сведения о производных типах см. в статье Наследование (Руководство по программированию на C#).

При использовании метода расширения для расширения типа, исходный код которого невозможно изменить, возникает риск того, что изменение в реализации типа вызовет сбой метода расширения.

В случае реализации методов расширения для какого-либо типа необходимо помнить о следующих фактах:

- Метод расширения никогда не будет вызван, если он имеет ту же сигнатуру, что и метод, определенный в типе.
- Методы расширения вводятся в область действия на уровне пространства имен. Например, при наличии нескольких статических классов, содержащих методы расширения в единственном пространстве имен с именем `Extensions`, все они будут введены в область действия директивой `using Extensions;`.

Для реализованной библиотеки классов не следует использовать методы расширения во избежание увеличения номера версии сборки. Если требуется добавить значительную функциональность в библиотеку, владельцем исходного кода которой вы являетесь, необходимо соблюдать стандартные правила .NET по управлению версиями сборок. Дополнительные сведения см. в разделе Версии сборок.

См. также

- Руководство по программированию на C#
- Parallel Programming Samples (Образцы параллельного программирования, включают множество примеров методов расширения)
- Лямбда-выражения

- Общие сведения о стандартных операторах запроса
- Conversion Rules for Instance Parameters and their Impact (Правила преобразования для параметров экземпляра и их влияние)
- Extension Methods Interoperability between Languages (Взаимодействие между языками с помощью методов расширения)
- Extension Methods and Curried Delegates (Методы расширения и каррированные делегаты)
- Extension method Binding and Error reporting

Практическое руководство. Реализация и вызов пользовательского метода расширения (руководство по программированию на C#)

18.12.2021 • 2 minutes to read

Этот раздел описывает, как реализовать свои методы расширения для любого типа .NET Framework. Клиентский код может использовать методы расширения путем добавления ссылки на содержащую их библиотеку DLL и добавления директивы using, которая указывает пространство имен, в котором определены методы расширения.

Определение и вызов метода расширения

1. Определите статический класс, который будет содержать метод расширения.

   Класс должен быть видимым для клиентского кода. Дополнительные сведения о правилах доступа см. в разделах Модификаторы доступа.

2. Реализуйте метод расширения как статический метод как минимум с тем же уровнем видимости, что и содержащий класс.

3. Первый параметр метода указывает тип, с которым работает метод. Ему должен предшествовать модификатор this.

4. В вызывающем коде добавьте директиву `using`, чтобы задать пространство имен, содержащее класс метода расширения.

5. Вызовите методы, как если бы они являлись методами экземпляра для типа.

   Обратите внимание, что первый параметр не указан вызывающим кодом, поскольку он представляет тип, к которому применяется оператор, и компилятору уже известен тип объекта. Вам необходимо предоставить аргументы только для параметров со 2 по `n`.

Пример

В приведенном ниже примере реализуется метод расширения с именем `WordCount` в классе `CustomExtensions.StringExtension`. Метод работает с классом String класса, который указан как первый параметр метода. Пространство имен `CustomExtensions` импортируется в пространство имен приложения, и метод вызывается внутри метода `Main`.

```
using System.Linq;
using System.Text;
using System;

namespace CustomExtensions
{
    // Extension methods must be defined in a static class.
    public static class StringExtension
    {
        // This is the extension method.
        // The first parameter takes the "this" modifier
        // and specifies the type for which the method is defined.
        public static int WordCount(this String str)
        {
            return str.Split(new char[] {' ', '.','?'}, StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
namespace Extension_Methods_Simple
{
    // Import the extension method namespace.
    using CustomExtensions;
    class Program
    {
        static void Main(string[] args)
        {
            string s = "The quick brown fox jumped over the lazy dog.";
            // Call the method as if it were an
            // instance method on the type. Note that the first
            // parameter is not specified by the calling code.
            int i = s.WordCount();
            System.Console.WriteLine("Word count of s is {0}", i);
        }
    }
}
```

Безопасность .NET

Методы расширения не предоставляют определенных уязвимостей безопасности. Они не могут использоваться для олицетворения существующих методов для типа, поскольку все конфликты имен разрешаются в пользу метода экземпляра или статического метода, определяемого самим типом. Методы расширения не могут получить доступ к любым конфиденциальным данным в расширенном классе.

См. также

- Руководство по программированию на C#
- Методы расширения
- Встроенный язык запросов LINQ
- Статические классы и члены статических классов
- protected
- internal
- public
- this
- namespace

Практическое руководство. Создание нового метода для перечисления (руководство по программированию на C#)

18.12.2021 • 2 minutes to read

Методы расширения позволяют добавить функциональные возможности, характерные для определенного типа перечисления.

Пример

В следующем примере перечисление `Grades` содержит возможные буквенные оценки, которые учащийся может получить в классе. Метод расширения с именем `Passing` добавляется в тип `Grades`, чтобы каждый экземпляр этого типа "знал", проходной это балл или нет.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Linq;

namespace EnumExtension
{
    // Define an extension method in a non-nested static class.
    public static class Extensions
    {
        public static Grades minPassing = Grades.D;
        public static bool Passing(this Grades grade)
        {
            return grade >= minPassing;
        }
    }

    public enum Grades { F = 0, D=1, C=2, B=3, A=4 };
    class Program
    {
        static void Main(string[] args)
        {
            Grades g1 = Grades.D;
            Grades g2 = Grades.F;
            Console.WriteLine("First {0} a passing grade.", g1.Passing() ? "is" : "is not");
            Console.WriteLine("Second {0} a passing grade.", g2.Passing() ? "is" : "is not");

            Extensions.minPassing = Grades.C;
            Console.WriteLine("\r\nRaising the bar!\r\n");
            Console.WriteLine("First {0} a passing grade.", g1.Passing() ? "is" : "is not");
            Console.WriteLine("Second {0} a passing grade.", g2.Passing() ? "is" : "is not");
        }
    }
  }
/* Output:
    First is a passing grade.
    Second is not a passing grade.

    Raising the bar!

    First is not a passing grade.
    Second is not a passing grade.
 */
```

Обратите внимание на то, что класс `Extensions` также содержит статическую переменную, которая обновляется динамически, а возвращаемое значение метода расширения отражает текущее значение этой переменной. Это показывает, что в фоновом режиме методы расширения вызываются непосредственно для статического класса, в котором они определены.

См. также

- Руководство по программированию на C#
- Методы расширения

Именованные и необязательные аргументы (Руководство по программированию на C#)

18.12.2021 • 7 minutes to read

C# 4 вводит именованные и необязательные аргументы. *Именованные аргументы* позволяют указать аргумент для параметра, связав этот аргумент с именем параметра, а не с его позицией в списке параметров. *Необязательные аргументы* позволяют опускать аргументы для некоторых параметров. Оба варианта можно использовать с методами, индексаторами, конструкторами и делегатами.

При использовании именованных и необязательных аргументов аргументы оцениваются в том порядке, в котором они отображаются в списке аргументов, а не в списке параметров.

Именованные и дополнительные параметры позволяют указать аргументы для выбранных параметров. Эта возможность значительно упрощает вызов интерфейсов COM, таких как API автоматизации Microsoft Office.

Именованные аргументы

Именованные аргументы освобождают вас от необходимости сопоставлять порядок параметров в списках параметров вызванных методов. Параметр для каждого аргумента можно указать, используя имя параметра. Например, функция, которая выводит сведения о заказе (имя продавца, номер заказа и наименование товара и т. д.), может вызываться путем передачи аргументов по позиции в порядке, определяемом функцией.

```
PrintOrderDetails("Gift Shop", 31, "Red Mug");
```

Если вы не помните порядок параметров, но знаете их имена, можете передать аргументы в любом порядке.

```
PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift Shop");
PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop", orderNum: 31);
```

Именованные аргументы также делают код более удобным для чтения, поскольку указывают, чему соответствует каждый аргумент. В приведенном ниже примере метода `sellerName` не может быть равен NULL или пробелу. Так как и `sellerName`, и `productName` являются строковыми типами, вместо передачи аргументов по позиции имеет смысл использовать именованные аргументы, чтобы устранить данную неоднозначность и сделать код более удобочитаемым.

Именованные аргументы при использовании с позиционными аргументами допустимы при условии, что

- за ними не следуют позиционные аргументы, либо,

  ```
  PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
  ```

- *начиная с C# 7.2*, они используются в правильной позиции. В примере ниже параметр `orderNum` находится в правильной позиции, но не имеет явно заданного имени.

```
PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red Mug");
```

Позиционные аргументы после внеочередных именованных аргументов недопустимы.

```
// This generates CS1738: Named argument specifications must appear after all fixed arguments have been
specified.
PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
```

Пример

Приведенный ниже код реализует как примеры из этого раздела, так и некоторые дополнительные примеры.

```
class NamedExample
{
    static void Main(string[] args)
    {
        // The method can be called in the normal way, by using positional arguments.
        PrintOrderDetails("Gift Shop", 31, "Red Mug");

        // Named arguments can be supplied for the parameters in any order.
        PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift Shop");
        PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop", orderNum: 31);

        // Named arguments mixed with positional arguments are valid
        // as long as they are used in their correct position.
        PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
        PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red Mug");    // C# 7.2 onwards
        PrintOrderDetails("Gift Shop", orderNum: 31, "Red Mug");                   // C# 7.2 onwards

        // However, mixed arguments are invalid if used out-of-order.
        // The following statements will cause a compiler error.
        // PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
        // PrintOrderDetails(31, sellerName: "Gift Shop", "Red Mug");
        // PrintOrderDetails(31, "Red Mug", sellerName: "Gift Shop");
    }

    static void PrintOrderDetails(string sellerName, int orderNum, string productName)
    {
        if (string.IsNullOrWhiteSpace(sellerName))
        {
            throw new ArgumentException(message: "Seller name cannot be null or empty.", paramName:
nameof(sellerName));
        }

        Console.WriteLine($"Seller: {sellerName}, Order #: {orderNum}, Product: {productName}");
    }
}
```

Необязательные аргументы

Определение метода, конструктора, индексатора или делегата может указывать, являются его параметры обязательными или нет. Любой вызов должен содержать аргументы для всех обязательных параметров; аргументы для необязательных параметров можно опустить.

Определение каждого необязательного параметра содержит его значение по умолчанию. Если аргумент для параметра не передается, используется значение по умолчанию. Значением по умолчанию должен быть один из следующих типов выражений:

- константное выражение;

- выражение в форме `new ValType()`, где `ValType` — это тип значения, например, enum или struct;
- выражение в форме default(ValType), где `ValType` — это тип значения.

Необязательные параметры определяются в конце списка параметров после всех обязательных параметров. Если вызывающий объект предоставляет аргумент для любого из последующих необязательных параметров, он должен содержать аргументы для всех предыдущих необязательных параметров. Пробелы, разделенные запятыми, в списке аргументов не поддерживаются. Например, в следующем коде метод экземпляра `ExampleMethod` определяется одним обязательным и двумя необязательными параметрами.

```
public void ExampleMethod(int required, string optionalstr = "default string",
    int optionalint = 10)
```

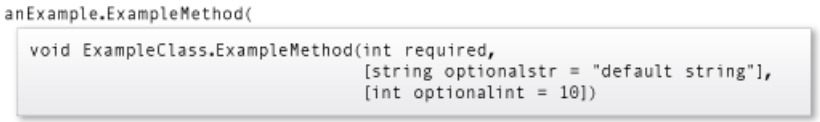
Следующий вызов `ExampleMethod` вызывает ошибку компилятора, поскольку аргумент предоставляется для третьего параметра, но не для второго.

```
//anExample.ExampleMethod(3, ,4);
```

Если вы знаете имя третьего параметра, можете использовать для выполнения задачи именованный аргумент.

```
anExample.ExampleMethod(3, optionalint: 4);
```

В IntelliSense необязательные параметры заключаются в квадратные скобки, как показано на следующем рисунке:



> **NOTE**
>
> Необязательные параметры также можно объявлять с помощью класса .NET OptionalAttribute. Для параметров `OptionalAttribute` значение по умолчанию не требуется.

Пример

В следующем примере конструктор `ExampleClass` имеет один параметр, который является необязательным. У метода экземпляра `ExampleMethod` есть один обязательный параметр, `required`, и два необязательных параметра, `optionalstr` и `optionalint`. Код в `Main` демонстрирует различные способы, которые можно использовать для вызова конструктора и метода.

```
namespace OptionalNamespace
{
    class OptionalExample
    {
        static void Main(string[] args)
        {
            // Instance anExample does not send an argument for the constructor's
            // optional parameter.
            ExampleClass anExample = new ExampleClass();
            anExample.ExampleMethod(1, "One", 1);
            anExample.ExampleMethod(2, "Two");
            anExample.ExampleMethod(3);
```

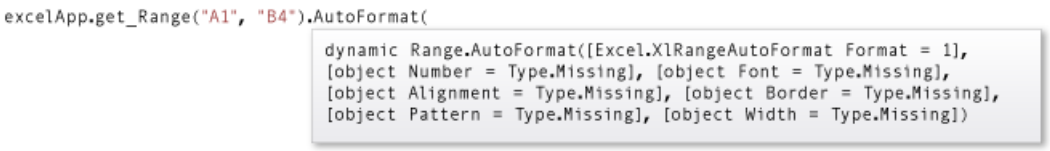
```
            // Instance anotherExample sends an argument for the constructor's
            // optional parameter.
            ExampleClass anotherExample = new ExampleClass("Provided name");
            anotherExample.ExampleMethod(1, "One", 1);
            anotherExample.ExampleMethod(2, "Two");
            anotherExample.ExampleMethod(3);

            // The following statements produce compiler errors.

            // An argument must be supplied for the first parameter, and it
            // must be an integer.
            //anExample.ExampleMethod("One", 1);
            //anExample.ExampleMethod();

            // You cannot leave a gap in the provided arguments.
            //anExample.ExampleMethod(3, ,4);
            //anExample.ExampleMethod(3, 4);

            // You can use a named parameter to make the previous
            // statement work.
            anExample.ExampleMethod(3, optionalint: 4);
        }
    }

    class ExampleClass
    {
        private string _name;

        // Because the parameter for the constructor, name, has a default
        // value assigned to it, it is optional.
        public ExampleClass(string name = "Default name")
        {
            _name = name;
        }

        // The first parameter, required, has no default value assigned
        // to it. Therefore, it is not optional. Both optionalstr and
        // optionalint have default values assigned to them. They are optional.
        public void ExampleMethod(int required, string optionalstr = "default string",
            int optionalint = 10)
        {
            Console.WriteLine(
                $"{_name}: {required}, {optionalstr}, and {optionalint}.");
        }
    }

    // The output from this example is the following:
    // Default name: 1, One, and 1.
    // Default name: 2, Two, and 10.
    // Default name: 3, default string, and 10.
    // Provided name: 1, One, and 1.
    // Provided name: 2, Two, and 10.
    // Provided name: 3, default string, and 10.
    // Default name: 3, default string, and 4.
}
```

Приведенный выше код выводит число примеров с некорректно примененными необязательными параметрами. Первый пример демонстрирует, что для первого параметра (обязательный) необходимо указать аргумент.

интерфейсы COM

Именованные и необязательные аргументы, а также поддержка динамических объектов значительно улучшают взаимодействие с API COM, такими как API автоматизации Office.

Например, метод AutoFormatв интерфейсе Microsoft Office Excel Range имеет семь параметров и все они необязательные. Эти параметры показаны на следующем рисунке:



В C# 3.0 и более ранних версиях аргумент необходимо указывать для каждого параметра, как показано в следующем примере.

```
// In C# 3.0 and earlier versions, you need to supply an argument for
// every parameter. The following call specifies a value for the first
// parameter, and sends a placeholder value for the other six. The
// default values are used for those parameters.
var excelApp = new Microsoft.Office.Interop.Excel.Application();
excelApp.Workbooks.Add();
excelApp.Visible = true;

var myFormat =
    Microsoft.Office.Interop.Excel.XlRangeAutoFormat.xlRangeAutoFormatAccounting1;

excelApp.get_Range("A1", "B4").AutoFormat(myFormat, Type.Missing,
    Type.Missing, Type.Missing, Type.Missing, Type.Missing, Type.Missing);
```

При этом вызов `AutoFormat` можно значительно упростить, используя именованные и необязательные аргументы, представленные в C# 4.0. Именованные и необязательные аргументы позволяют опускать аргументы для необязательных параметров, если значение параметра по умолчанию менять не нужно. В следующем вызове значение задается только для одного из семи параметров.

```
// The following code shows the same call to AutoFormat in C# 4.0. Only
// the argument for which you want to provide a specific value is listed.
excelApp.Range["A1", "B4"].AutoFormat( Format: myFormat );
```

См. сведения и примеры в руководствах по использованию именованных и необязательных аргументов в программировании Office и получению доступа к объектам взаимодействия Office с помощью функций языка C#.

Разрешение перегрузки

Использование именованных и необязательных аргументов влияет на разрешение перегрузки описанным ниже образом.

- Метод, индексатор или конструктор является кандидатом на выполнение, если каждый из его параметров необязателен либо по имени или позиции соответствует одному и тому же аргументу в операторе вызова, и этот аргумент можно преобразовать в тип параметра.
- Если найдено более одного кандидата, правила разрешения перегрузки для предпочтительных преобразований применяются к аргументам, указанным явно. Опущенные аргументы для необязательных параметров игнорируются.
- Если два кандидата определяются как равно подходящие, предпочтение отдается кандидату без необязательных параметров, аргументы которых в вызове были опущены. При разрешении перегрузки обычно используются кандидаты с меньшим числом параметров.

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является

предписывающим источником информации о синтаксисе и использовании языка C#.

Практическое руководство. Использование именованных и необязательных аргументов в программировании приложений Office (руководство по программированию на C#)

18.12.2021 • 4 minutes to read

Появившиеся в C# 4 именованные и необязательные аргументы повышают удобство, гибкость и удобочитаемость программирования на C#. Кроме того, эти функции значительно упрощают доступ к COM-интерфейсам, таким как интерфейсы API автоматизации Microsoft Office.

В следующем примере у метода ConvertToTable имеется шестнадцать параметров, представляющих характеристики таблицы, например число столбцов и строк, форматирование, границы и цвета. Все шестнадцать параметров являются необязательными, поскольку в большинстве случаев не требуется задавать конкретные значения для всех этих параметров. Однако без именованных и необязательных аргументов приходилось указывать значение или значение-заполнитель для каждого из параметров. Именованные и необязательные параметры позволяют задавать значения только для тех параметров, которые требуются в конкретном проекте.

Для выполнения этих процедур на компьютере должно быть установлено приложение Microsoft Office Word.

> **NOTE**
> Отображаемые на компьютере имена или расположения некоторых элементов пользовательского интерфейса Visual Studio могут отличаться от указанных в следующих инструкциях. Это зависит от имеющегося выпуска Visual Studio и используемых параметров. Дополнительные сведения см. в разделе Персонализация среды IDE.

Создание нового проекта консольного приложения

1. Запустите Visual Studio.
2. В меню **Файл** выберите пункт **Создать**, а затем команду **Проект**.
3. В области **Категории шаблонов** разверните узел Visual C# и выберите Windows.
4. В верхней части области **Шаблоны** в поле **Требуемая версия** .NET Framework должно отображаться значение .NET Framework 4.
5. В области **Шаблоны** щелкните **Консольное приложение**.
6. Введите имя проекта в поле **Имя**.
7. Нажмите кнопку **ОК**.

   В **обозревателе решений** появится новый проект.

Добавление ссылки

1. В **обозревателе решений** щелкните имя проекта правой кнопкой мыши и выберите пункт **Добавить ссылку**. Откроется диалоговое окно **Добавление ссылки**.

2. На странице .NET выберите Microsoft.Office.Interop.Word в списке **Имя компонента**.
3. Нажмите кнопку **ОК**.

Добавление необходимых директив using

1. В **обозревателе решений** щелкните правой кнопкой мыши файл *Program.cs* и выберите пункт **Просмотреть код**.
2. В начало файла кода добавьте следующие директивы `using`:

```
using Word = Microsoft.Office.Interop.Word;
```

Отображение текста в документ Word

1. В класс `Program` в файле *Program.cs* добавьте следующий метод для создания приложения Word и документа Word. Метод Add имеет четыре необязательных параметра. В этом примере используются значения по умолчанию. Поэтому в операторе вызова указывать аргументы не требуется.

```
static void DisplayInWord()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;
    // docs is a collection of all the Document objects currently
    // open in Word.
    Word.Documents docs = wordApp.Documents;

    // Add a document to the collection and name it doc.
    Word.Document doc = docs.Add();
}
```

2. Добавьте в конец метода следующий код, чтобы определить место отображения текста в документе и текст для отображения:

```
// Define a range, a contiguous area in the document, by specifying
// a starting and ending character position. Currently, the document
// is empty.
Word.Range range = doc.Range(0, 0);

// Use the InsertAfter method to insert a string at the end of the
// current range.
range.InsertAfter("Testing, testing, testing. . .");
```

Запуск приложения

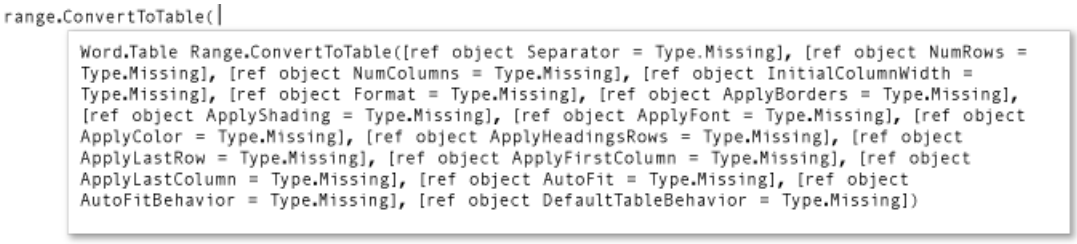
1. Добавьте в метод Main следующую инструкцию:

```
DisplayInWord();
```

2. Нажмите клавиши CTRL+F5, чтобы запустить проект. Появится документ Word, содержащий указанный текст.

Замена текста на таблицу

1. Метод `ConvertToTable` служит для преобразования текста в таблицу. У метода имеется шестнадцать необязательных параметров. IntelliSense заключает необязательные параметры в квадратные скобки, как показано на следующем рисунке.



   Именованные и необязательные аргументы позволяют задавать значения только для тех параметров, которые требуется изменить. Добавьте в конец метода `DisplayInWord` следующий код, чтобы создать простую таблицу. Аргумент указывает, что запятые в текстовой строке в `range` разделяют ячейки таблицы.

```
// Convert to a simple table. The table will have a single row with
// three columns.
range.ConvertToTable(Separator: ",");
```

   В более ранних версиях C# для вызова `ConvertToTable` каждому параметру требовался ссылочный аргумент, как показано в следующем коде:

```
// Call to ConvertToTable in Visual C# 2008 or earlier. This code
// is not part of the solution.
var missing = Type.Missing;
object separator = ",";
range.ConvertToTable(ref separator, ref missing, ref missing,
    ref missing, ref missing, ref missing, ref missing,
    ref missing, ref missing, ref missing, ref missing,
    ref missing, ref missing, ref missing, ref missing,
    ref missing);
```

2. Нажмите клавиши CTRL+F5, чтобы запустить проект.

Экспериментирование с другими параметрами

1. Для изменения таблицы, чтобы она содержала один столбец и три строки, замените последнюю строку метода `DisplayInWord` следующей инструкцией и нажмите сочетание клавиш CTRL+F5.

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1);
```

2. Чтобы использовать заранее определенный формат таблицы, замените последнюю строку метода `DisplayInWord` следующей инструкцией и нажмите сочетание клавиш CTRL+F5. В качестве формата можно использовать любую из констант WdTableFormat.

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1,
    Format: Word.WdTableFormat.wdTableFormatElegant);
```

Пример

Следующий код включает полный пример:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Word = Microsoft.Office.Interop.Word;

namespace OfficeHowTo
{
    class WordProgram
    {
        static void Main(string[] args)
        {
            DisplayInWord();
        }

        static void DisplayInWord()
        {
            var wordApp = new Word.Application();
            wordApp.Visible = true;
            // docs is a collection of all the Document objects currently
            // open in Word.
            Word.Documents docs = wordApp.Documents;

            // Add a document to the collection and name it doc.
            Word.Document doc = docs.Add();

            // Define a range, a contiguous area in the document, by specifying
            // a starting and ending character position. Currently, the document
            // is empty.
            Word.Range range = doc.Range(0, 0);

            // Use the InsertAfter method to insert a string at the end of the
            // current range.
            range.InsertAfter("Testing, testing, testing. . .");

            // You can comment out any or all of the following statements to
            // see the effect of each one in the Word document.

            // Next, use the ConvertToTable method to put the text into a table.
            // The method has 16 optional parameters. You only have to specify
            // values for those you want to change.

            // Convert to a simple table. The table will have a single row with
            // three columns.
            range.ConvertToTable(Separator: ",");

            // Change to a single column with three rows..
            range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1);

            // Format the table.
            range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1,
                Format: Word.WdTableFormat.wdTableFormatElegant);
        }
    }
}
```

См. также

- Именованные и необязательные аргументы

Конструкторы (руководство по программированию на C#)

18.12.2021 • 2 minutes to read

Каждый раз, когда создается класс или структура, вызывается конструктор. Класс или структура может иметь несколько конструкторов, принимающих различные аргументы. Конструкторы позволяют программисту задавать значения по умолчанию, ограничивать число установок и писать код, который является гибким и удобным для чтения. Дополнительные сведения и примеры см. в разделах Конструкторы экземпляров и Использование конструкторов.

Синтаксис конструктора

Конструктор — это метод, имя которого совпадает с именем его типа. Его сигнатура метода содержит только имя метода и список параметров. Она не содержит возвращаемый тип. В приведенном ниже примере демонстрируется конструктор для класса с именем `Person`.

```
public class Person
{
   private string last;
   private string first;

   public Person(string lastName, string firstName)
   {
      last = lastName;
      first = firstName;
   }

   // Remaining implementation of Person class.
}
```

Если конструктор поддерживает реализацию в виде оператора, можно использовать определение тела выражения. В следующем примере определяется класс `Location`, конструктор которого имеет один строковый параметр *name*. Определение тела выражения присваивает аргумент полю `locationName`.

```
public class Location
{
   private string locationName;

   public Location(string name) => Name = name;

   public string Name
   {
      get => locationName;
      set => locationName = value;
   }
}
```

Статические конструкторы

В приведенных выше примерах показаны конструкторы экземпляров, которые создают новый объект. В классе или структуре также может быть статический конструктор, который инициализирует статические члены типа. Статические конструкторы не имеют параметров. Если вы не предоставили статический

конструктор для инициализации статических полей, компилятор C# инициализирует статические поля значениями по умолчанию, как показано в статье Значения по умолчанию типов C#.

В следующем примере статический конструктор используется для инициализации статического поля.

```
public class Adult : Person
{
   private static int minimumAge;

   public Adult(string lastName, string firstName) : base(lastName, firstName)
   { }

   static Adult()
   {
      minimumAge = 18;
   }

   // Remaining implementation of Adult class.
}
```

Можно также определить статический конструктор с помощью определения тела выражения, как показано в следующем примере.

```
public class Child : Person
{
   private static int maximumAge;

   public Child(string lastName, string firstName) : base(lastName, firstName)
   { }

   static Child() => maximumAge = 18;

   // Remaining implementation of Child class.
}
```

Дополнительные сведения и примеры см. в разделе Статические конструкторы.

В этом разделе

Использование конструкторов

Конструкторы экземпляров

Закрытые конструкторы

Статические конструкторы

Практическое руководство. Создание конструктора копий

См. также

- Руководство по программированию на C#
- Система типов C#
- Методы завершения
- static
- Why Do Initializers Run In The Opposite Order As Constructors? Part One (Почему инициализаторы выполняются в порядке, обратном действию конструкторов? Часть 1)

Использование конструкторов (Руководство по программированию на C#)

18.12.2021 • 3 minutes to read

Каждый раз, когда создается класс или структура, вызывается конструктор. Конструкторы имеют имя, совпадающее с именем класса или структуры, и обычно инициализируют члены данных нового объекта.

В следующем примере класс с именем `Taxi` определяется с помощью простого конструктора. Затем оператор new создает экземпляр этого класса. Конструктор `Taxi` вызывается оператором `new` сразу после того, как новому объекту будет выделена память.

```
public class Taxi
{
    public bool IsInitialized;

    public Taxi()
    {
        IsInitialized = true;
    }
}

class TestTaxi
{
    static void Main()
    {
        Taxi t = new Taxi();
        Console.WriteLine(t.IsInitialized);
    }
}
```

Конструктор, который не принимает никаких параметров, называется *конструктором без параметров*. Конструкторы без параметров вызываются всякий раз, когда создается экземпляр объекта с помощью оператора `new`, а аргументы в `new` не передаются. Дополнительные сведения см. в разделе Конструкторы экземпляров.

Если класс не является статическим, компилятор C# выделяет классам без конструкторов открытый конструктор без параметров, позволяющий создавать экземпляры классов. Дополнительные сведения см. в статье Статические классы и члены статических классов.

Создание экземпляров класса можно запретить, сделав конструктор закрытым, следующим образом:

```
class NLog
{
    // Private Constructor:
    private NLog() { }

    public static double e = Math.E;  //2.71828...
}
```

Дополнительные сведения см. в разделе Закрытые конструкторы.

Конструкторы для типов struct похожи на конструкторы классов, однако `structs` не может содержать явный конструктор без параметров, так как он предоставляется компилятором автоматически. Этот конструктор инициализирует каждое поле в `struct` со значением по умолчанию. При этом конструктор

без параметров вызывается только в том случае, если экземпляр `struct` создается с помощью переменной `new`. Например, в этом коде конструктор без параметров используется для Int32 — это обеспечивает инициализацию целого числа:

```
int i = new int();
Console.WriteLine(i);
```

> **NOTE**
> Начиная с C# 10 тип структуры может содержать явный конструктор без параметров. Дополнительные сведения см. в разделе Конструкторы без параметров и инициализаторы полей статьи Типы структур.

В то же время следующий код вызывает ошибку компилятора, поскольку не использует `new`, и потому, что использует не инициализированный объект:

```
int i;
Console.WriteLine(i);
```

Кроме того, объекты на основе `structs` (включая все встроенные числовые типы) можно инициализировать или назначить, а затем использовать, как в следующем примере:

```
int a = 44;  // Initialize the value type...
int b;
b = 33;      // Or assign it before using it.
Console.WriteLine("{0}, {1}", a, b);
```

В связи с этим вызывать конструктор без параметров для типа значения необязательно.

Оба класса и `structs` могут определять конструкторы, принимающие параметры. Конструкторы, принимающие параметры, необходимо вызывать с помощью оператора `new` или base. Классы и `structs` могут определять также несколько конструкторов; для определения конструктора без параметров ни один их них не требуется. Пример:

```
public class Employee
{
    public int Salary;

    public Employee() { }

    public Employee(int annualSalary)
    {
        Salary = annualSalary;
    }

    public Employee(int weeklySalary, int numberOfWeeks)
    {
        Salary = weeklySalary * numberOfWeeks;
    }
}
```

Этот класс можно создать, воспользовавшись одним из следующих операторов:

```
Employee e1 = new Employee(30000);
Employee e2 = new Employee(500, 52);
```

Конструктор может использовать ключевое слово `base` для вызова конструктора базового класса. Пример:

```
public class Manager : Employee
{
    public Manager(int annualSalary)
        : base(annualSalary)
    {
        //Add further instructions here.
    }
}
```

В этом примере конструктор базового класса вызывается перед выполнением соответствующего ему блока. Ключевое слово `base` можно использовать как с параметрами, так и без них. Любые параметры для конструктора можно использовать как параметры для `base` или как часть выражения. Дополнительные сведения см. в разделе base.

В производном классе, если конструктор базового класса не вызывается явным образом с помощью ключевого слова `base`, конструктор без параметров, если он существует, вызывается неявно. Это означает, что следующие объявления конструкторов действуют одинаково:

```
public Manager(int initialData)
{
    //Add further instructions here.
}
```

```
public Manager(int initialData)
    : base()
{
    //Add further instructions here.
}
```

Если базовый класс не предлагает конструктор без параметров, производный класс должен явно вызвать конструктор базового класса с помощью `base`.

Конструктор может вызывать другой конструктор в том же объекте с помощью ключевого слова this. Как и `base`, `this` можно использовать с параметрами или без, а все параметры в конструкторе доступны как параметры `this` или как часть выражения. Например, второй конструктор в предыдущем примере можно переписать, используя `this`:

```
public Employee(int weeklySalary, int numberOfWeeks)
    : this(weeklySalary * numberOfWeeks)
{
}
```

Применение ключевого слова `this` в приведенном выше примере привело к вызову конструктора:

```
public Employee(int annualSalary)
{
    Salary = annualSalary;
}
```

Конструкторы могут иметь пометку public, private, protected, internal, protected internal или private protected. Эти модификаторы доступа определяют, каким образом пользователи класса смогут создавать класс. Дополнительные сведения см. в статье Модификаторы доступа.

Конструктор можно объявить статическим, используя ключевое слово static. Статические конструкторы вызываются автоматически непосредственно перед доступом к статическим полям и обычно используются для инициализации членов статического класса. Дополнительные сведения см. в разделе Статические конструкторы.

Спецификация языка C#

Дополнительные сведения см. в разделах Конструкторы экземпляров и Статические конструкторы в Спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Руководство по программированию на C#
- Система типов C#
- Конструкторы
- Методы завершения

Конструкторы экземпляров (руководство по программированию на C#)

18.12.2021 • 2 minutes to read

Конструктор экземпляра объявляется для указания кода, выполняемого при создании нового экземпляра типа с помощью выражения `new`. Для инициализации класса static или статических переменных в нестатическом классе определяется статический конструктор.

Как показано в следующем примере, можно объявить несколько конструкторов экземпляров в одном типе:

```
using System;

class Coords
{
    public Coords()
        : this(0, 0)
    {  }

    public Coords(int x, int y)
    {
        X = x;
        Y = y;
    }

    public int X { get; set; }
    public int Y { get; set; }

    public override string ToString() => $"({X},{Y})";
}

class Example
{
    static void Main()
    {
        var p1 = new Coords();
        Console.WriteLine($"Coords #1 at {p1}");
        // Output: Coords #1 at (0,0)

        var p2 = new Coords(5, 3);
        Console.WriteLine($"Coords #2 at {p2}");
        // Output: Coords #2 at (5,3)
    }
}
```

В предыдущем примере первый конструктор, без параметров, вызывает второй, и оба аргумента равны `0`. Для этого воспользуйтесь ключевым словом `this`.

При объявлении конструктора экземпляра в производном классе можно вызвать конструктор базового класса. Для этого используйте ключевое слово `base`, как показано в следующем примере:

```
using System;

abstract class Shape
{
    public const double pi = Math.PI;
    protected double x, y;

    public Shape(double x, double y)
    {
        this.x = x;
        this.y = y;
    }

    public abstract double Area();
}

class Circle : Shape
{
    public Circle(double radius)
        : base(radius, 0)
    {  }

    public override double Area() => pi * x * x;
}

class Cylinder : Circle
{
    public Cylinder(double radius, double height)
        : base(radius)
    {
        y = height;
    }

    public override double Area() => (2 * base.Area()) + (2 * pi * x * y);
}

class Example
{
    static void Main()
    {
        double radius = 2.5;
        double height = 3.0;

        var ring = new Circle(radius);
        Console.WriteLine($"Area of the circle = {ring.Area():F2}");
        // Output: Area of the circle = 19.63

        var tube = new Cylinder(radius, height);
        Console.WriteLine($"Area of the cylinder = {tube.Area():F2}");
        // Output: Area of the cylinder = 86.39
    }
}
```

Конструкторы без параметров

Если у *класса* нет явных конструкторов экземпляров, C# предоставляет конструктор без параметров, который можно использовать для создания экземпляра этого класса, как показано в следующем примере:

```
using System;

public class Person
{
    public int age;
    public string name = "unknown";
}

class Example
{
    static void Main()
    {
        var person = new Person();
        Console.WriteLine($"Name: {person.name}, Age: {person.age}");
        // Output:  Name: unknown, Age: 0
    }
}
```

Этот конструктор инициализирует поля и свойства экземпляров в соответствии с необходимыми инициализаторами. Если поле или свойство не имеет инициализатора, его значение устанавливается в значение по умолчанию для типа поля или свойства. Если в классе объявлен хотя бы один конструктор экземпляра, C# не предоставляет конструктор без параметров.

Тип *структуры* всегда предоставляет конструктор без параметров, как показано ниже:

- В C# 9.0 и более ранних версий это неявный конструктор без параметров, который создает значение по умолчанию типа.
- В C# 10 и более поздних версий это либо неявный конструктор без параметров, который создает значение по умолчанию для типа, либо явно объявленный конструктор без параметров. Дополнительные сведения см. в разделе Конструкторы без параметров и инициализаторы полей статьи Типы структур.

См. также

- Руководство по программированию на C#
- Классы, структуры и записи
- Конструкторы
- Методы завершения
- base
- this

Закрытые конструкторы (Руководство по программированию на C#)

18.12.2021 • 2 minutes to read

Закрытый конструктор — это особый конструктор экземпляров. Обычно он используется в классах, содержащих только статические элементы. Если в классе один или несколько закрытых конструкторов и ни одного открытого конструктора, то прочие классы (за исключением вложенных классов) не смогут создавать экземпляры этого класса. Пример:

```
class NLog
{
    // Private Constructor:
    private NLog() { }

    public static double e = Math.E;  //2.71828...
}
```

Объявление пустого конструктора запрещает автоматическое создание конструктора без параметров. Обратите внимание, что если не использовать с конструктором модификатор доступа, то по умолчанию он все равно будет закрытым. Однако обычно используется модификатор private, чтобы явно обозначить невозможность создания экземпляров этого класса.

Закрытые конструкторы используются, чтобы не допустить создания экземпляров класса при отсутствии полей или методов экземпляра, например для класса Math, или когда осуществляется вызов метода для получения экземпляра класса. Если все методы в классе являются статическими, имеет смысл сделать статическим весь класс. Дополнительные сведения см. в разделе Статические классы и члены статических классов.

Пример

Ниже приведен пример класса с закрытым конструктором.

```
public class Counter
{
    private Counter() { }

    public static int currentCount;

    public static int IncrementCount()
    {
        return ++currentCount;
    }
}

class TestCounter
{
    static void Main()
    {
        // If you uncomment the following statement, it will generate
        // an error because the constructor is inaccessible:
        // Counter aCounter = new Counter();   // Error

        Counter.currentCount = 100;
        Counter.IncrementCount();
        Console.WriteLine("New count: {0}", Counter.currentCount);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: New count: 101
```

Обратите внимание, что если в примере раскомментировать следующий оператор, возникнет ошибка, так как конструктор недоступен из-за уровня защиты:

```
// Counter aCounter = new Counter();   // Error
```

Спецификация языка C#

Дополнительные сведения см. в разделе Закрытые конструкторы в Спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Руководство по программированию на C#
- Система типов C#
- Конструкторы
- Методы завершения
- private
- public

Статические конструкторы (Руководство по программированию в C#)

18.12.2021 • 4 minutes to read

Статический конструктор используется для инициализации любых статических данных или для выполнения определенного действия, которое требуется выполнить только один раз. Он вызывается автоматически перед созданием первого экземпляра или ссылкой на какие-либо статические члены.

```
class SimpleClass
{
    // Static variable that must be initialized at run time.
    static readonly long baseline;

    // Static constructor is called at most one time, before any
    // instance constructor is invoked or member is accessed.
    static SimpleClass()
    {
        baseline = DateTime.Now.Ticks;
    }
}
```

Примечания

Статические конструкторы обладают следующими свойствами.

- Статический конструктор не принимает модификаторы доступа и не имеет параметров.
- Класс или структура могут иметь только один статический конструктор.
- Статические конструкторы не могут быть унаследованы или перегружены.
- Статический конструктор нельзя вызывать напрямую. Он предназначен только для вызова из общеязыковой среды выполнения (CLR). Он запускается автоматически.
- Пользователь не управляет временем, в течение которого статический конструктор выполняется в программе.
- Статический конструктор вызывается автоматически. Он инициализирует класс перед созданием первого экземпляра или ссылок на какие-либо статические элементы. Статический конструктор выполняется раньше, чем конструктор экземпляра. Статический конструктор типа вызывается при вызове статического метода, назначенного делегату или событию, а не при его назначении. Если в классе со статическим конструктором присутствуют инициализаторы переменных для статических полей, они выполняются в том порядке, в котором они появляются в объявлении класса. Эти инициализаторы выполняются непосредственно перед выполнением статического конструктора.
- Если вы не предоставили статический конструктор для инициализации статических полей, все статические поля инициализируются значениями по умолчанию, которые указаны в разделе Значения по умолчанию типов C#.
- Если статический конструктор создает исключение, среда выполнения не вызывает его во второй раз, и тогда тип остается неинициализированным на все время существования домена приложения. Чаще всего исключение TypeInitializationException вызывается, когда статический конструктор не может создать экземпляр типа или в статическом конструкторе создается

необработанное исключение. Для устранения неполадок со статическими конструкторами, которые не определены явным образом в исходном коде, может потребоваться проверка кода на промежуточном языке (IL).

- Наличие статического конструктора не позволяет добавлять атрибут типа BeforeFieldInit. Это ограничивает возможности оптимизации во время выполнения.
- Поле, объявленное как `static readonly`, может быть присвоено только при его объявлении или в статическом конструкторе. Если явный статический конструктор не требуется, инициализируйте статические поля в объявлении, а не через статический конструктор, чтобы оптимизировать среду выполнения.
- Среда выполнения вызывает статический конструктор не более одного раза для каждого домена приложения. Этот вызов выполняется в заблокированном регионе на основе определенного типа класса. В тексте статического конструктора не нужны дополнительные механизмы блокировки. Чтобы избежать риска взаимоблокировки, не блокируйте текущий поток в статических конструкторах и инициализаторах. Например, не следует ожидать выполнения задач, потоков, обработчиков ожидания или событий, а также создавать блокировки или выполнять блокирующие параллельные операции, как, например, параллельные циклы, `Parallel.Invoke` и запросы Parallel LINQ.

> **NOTE**
> Наличие явного статического конструктора следует документировать, несмотря на отсутствие прямого доступа к нему, поскольку это важно для устранения неполадок с исключениями при инициализации.

Использование

- Типичным использованием статических конструкторов является случай, когда класс использует файл журнала и конструктор применяется для добавления записей в этот файл.
- Статические конструкторы также полезны при создании классов-оболочек для неуправляемого кода, когда конструктор может вызвать метод `LoadLibrary`.
- Кроме того, статические конструкторы очень удобны для проведения проверок времени выполнения для параметров типов, которые нельзя проверить во время компиляции с помощью ограничений параметров типов.

Пример

В этом примере класс `Bus` имеет статический конструктор. При создании первого экземпляра класса `Bus` (`bus1`) для инициализации класса вызывается статический конструктор. В выходных данных этого примера можно увидеть, что статический конструктор выполняется только один раз, несмотря на то, что создается два экземпляра класса `Bus`. Кроме того, этот конструктор вызывается до выполнения конструктора экземпляра.

```
public class Bus
{
    // Static variable used by all Bus instances.
    // Represents the time the first bus of the day starts its route.
    protected static readonly DateTime globalStartTime;

    // Property for the number of each bus.
    protected int RouteNumber { get; set; }

    // Static constructor to initialize the static variable.
    // It is invoked before the first instance constructor is run.
    static Bus()
```

```
    {
        globalStartTime = DateTime.Now;

        // The following statement produces the first line of output,
        // and the line occurs only once.
        Console.WriteLine("Static constructor sets global start time to {0}",
            globalStartTime.ToLongTimeString());
    }

    // Instance constructor.
    public Bus(int routeNum)
    {
        RouteNumber = routeNum;
        Console.WriteLine("Bus #{0} is created.", RouteNumber);
    }

    // Instance method.
    public void Drive()
    {
        TimeSpan elapsedTime = DateTime.Now - globalStartTime;

        // For demonstration purposes we treat milliseconds as minutes to simulate
        // actual bus times. Do not do this in your actual bus schedule program!
        Console.WriteLine("{0} is starting its route {1:N2} minutes after global start time {2}.",
                          this.RouteNumber,
                          elapsedTime.Milliseconds,
                          globalStartTime.ToShortTimeString());
    }
}

class TestBus
{
    static void Main()
    {
        // The creation of this instance activates the static constructor.
        Bus bus1 = new Bus(71);

        // Create a second bus.
        Bus bus2 = new Bus(72);

        // Send bus1 on its way.
        bus1.Drive();

        // Wait for bus2 to warm up.
        System.Threading.Thread.Sleep(25);

        // Send bus2 on its way.
        bus2.Drive();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Sample output:
    Static constructor sets global start time to 3:57:08 PM.
    Bus #71 is created.
    Bus #72 is created.
    71 is starting its route 6.00 minutes after global start time 3:57 PM.
    72 is starting its route 31.00 minutes after global start time 3:57 PM.
*/
```

Спецификация языка C#

Дополнительные сведения см. в разделе Статические конструктор в спецификации языка C#.

См. также

- Руководство по программированию на C#
- Система типов C#
- Конструкторы
- Статические классы и члены статических классов
- Методы завершения
- Рекомендации по разработке конструкторов
- Предупреждение о безопасности CA2121. Статические конструкторы должны быть частными

Практическое руководство. Создание конструктора копий (руководство по программированию на C#)

18.12.2021 • 2 minutes to read

Для записей C# предоставляет конструктор копирования для объектов, но для классов его необходимо писать самостоятельно.

Пример

В следующем примере класс `Person` определяет конструктор копии, который использует экземпляр `Person` в качестве аргумента. Значения свойств аргумента присваиваются свойствам нового экземпляра `Person`. Код содержит дополнительный конструктор копии, который отправляет свойства `Name` и `Age` экземпляра, который необходимо скопировать конструктору экземпляра класса.

```
using System;

class Person
{
    // Copy constructor.
    public Person(Person previousPerson)
    {
        Name = previousPerson.Name;
        Age = previousPerson.Age;
    }

    //// Alternate copy constructor calls the instance constructor.
    //public Person(Person previousPerson)
    //    : this(previousPerson.Name, previousPerson.Age)
    //{
    //}

    // Instance constructor.
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    public int Age { get; set; }

    public string Name { get; set; }

    public string Details()
    {
        return Name + " is " + Age.ToString();
    }
}

class TestPerson
{
    static void Main()
    {
        // Create a Person object by using the instance constructor.
        Person person1 = new Person("George", 40);

        // Create another Person object, copying person1.
        Person person2 = new Person(person1);

        // Change each person's age.
        person1.Age = 39;
        person2.Age = 41;

        // Change person2's name.
        person2.Name = "Charles";

        // Show details to verify that the name and age fields are distinct.
        Console.WriteLine(person1.Details());
        Console.WriteLine(person2.Details());

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:
// George is 39
// Charles is 41
```

См. также

- ICloneable
- Записи
- Руководство по программированию на C#
- Система типов C#
- Конструкторы
- Методы завершения

Методы завершения (руководство по программированию в C#)

18.12.2021 • 4 minutes to read

Методы завершения (также называемые **деструкторами**) используются для любой необходимой окончательной очистки, когда сборщик мусора окончательно удаляет экземпляра класса. В большинстве случаев можно избежать написания метода завершения, используя System.Runtime.InteropServices.SafeHandle или производные классы для переноса любого неуправляемого дескриптора.

Примечания

- В структурах определение методов завершения невозможно. Они применяются только в классах.
- Каждый класс может иметь только один метод завершения.
- Методы завершения не могут быть унаследованы или перегружены.
- Методы завершения невозможно вызвать. Они запускаются автоматически.
- Метод завершения не принимает модификаторов и не имеет параметров.

Например, ниже показано объявление метода завершения для класса `Car`.

```
class Car
{
    ~Car()  // finalizer
    {
        // cleanup statements...
    }
}
```

Метод завершения можно также реализовать как определение тела выражения, как показано в следующем примере.

```
using System;

public class Destroyer
{
   public override string ToString() => GetType().Name;

   ~Destroyer() => Console.WriteLine($"The {ToString()} finalizer is executing.");
}
```

Метод завершения неявно вызывает метод Finalize для базового класса объекта. В связи с этим вызов метода завершения неявно преобразуется в следующий код:

```
protected override void Finalize()
{
    try
    {
        // Cleanup statements...
    }
    finally
    {
        base.Finalize();
    }
}
```

Это означает, что метод `Finalize` вызывается рекурсивно для всех экземпляров цепочки наследования начиная с самого дальнего и заканчивая самым первым.

> **NOTE**
> Не следует использовать пустые методы завершения. Если класс содержит метод завершения, то в очереди метода `Finalize` создается запись. Эта очередь обрабатывается сборщиком мусора. Когда сборщик мусора обрабатывает очередь, он вызывает каждый метод завершения. Ненужные методы завершения, включая пустые методы завершения, методы завершения, которые вызывают только метод завершения базового класса, или методы завершения, которые вызывают только условно порожденные методы, приводят к ненужному снижению производительности.

Программист не может управлять моментом вызова метода завершения, поскольку это определяется сборщиком мусора. Сборщик мусора проверяет наличие объектов, которые больше не используются приложением. Если он считает, что какой-либо объект требует уничтожения, то вызывает метод завершения (при наличии) и освобождает память, используемую для хранения этого объекта. Сборку мусора можно выполнить принудительно, вызвав метод Collect, но в большинстве случаев этого следует избегать из-за возможных проблем с производительностью.

> **NOTE**
> От конкретной реализации .NET зависит, выполняются ли методы завершения как часть завершения работы приложения. При завершении работы приложения .NET Framework прилагает все возможные усилия для вызова методов завершения для всех своих объектов, которые еще не были собраны сборщиком мусора, если такая очистка не была подавлена (например, вызовом метода библиотеки `GC.SuppressFinalize`). .NET 5 (включая .NET Core) и более поздние версии не вызывают методы завершения как часть завершения приложения. Дополнительные сведения об этой проблеме см. в Проблема dotnet/csharpstandard #291, рассмотренная на сайте GitHub.

Если необходимо надежно выполнить очистку при наличии приложения, зарегистрируйте обработчик для события System.AppDomain.ProcessExit. Этот обработчик обеспечит вызов IDisposable.Dispose() (или, IAsyncDisposable.DisposeAsync()) для всех объектов, для которых требуется очистка перед выходом из приложения. Поскольку нельзя вызвать *метод завершения* напрямую и нельзя гарантировать, что сборщик мусора вызовет все методы завершения до выхода, необходимо использовать `Dispose` или `DisposeAsync`, чтобы освободить ресурсы.

Использование методов завершения для освобождения ресурсов

В целом язык C# не требует управления памятью в той степени, в какой это требуется в случае разработки кода на языке, не рассчитанном на среду выполнения со сборкой мусора. Это связано с тем, что сборщик мусора платформы .NET неявным образом управляет выделением и высвобождением памяти для объектов. Однако при инкапсуляции приложением неуправляемых ресурсов, например окон,

файлов и сетевых подключений, для высвобождения этих ресурсов следует использовать методы завершения. Если объект допускает завершение, то сборщик мусора выполняет метод `Finalize` этого объекта.

Освобождение ресурсов явным образом

В случае, когда приложением используется ценный внешний ресурс, также рекомендуется обеспечить способ высвобождения этого ресурса явным образом, прежде чем сборщик мусора освободит объект. Для высвобождения ресурса реализуется метод `Dispose` интерфейса IDisposable, который выполняет необходимую для объекта очистку. Это может значительно повысить производительность приложения. Даже в случае использования такого явного управления ресурсами метод завершения становится резервным средством очистки ресурсов, если вызов метода `Dispose` выполнить не удастся.

Дополнительные сведения об очистке ресурсов см. в следующих статьях:

- Очистка неуправляемых ресурсов
- Реализация метода dispose
- Реализация метода DisposeAsync
- Оператор using

Пример

В приведенном ниже примере создаются три класса, образующих цепочку наследования. Класс `First` является базовым, класс `Second` является производным от класса `First`, а класс `Third` является производным от класса `Second`. Все три класса имеют методы завершения. В методе `Main` создается экземпляр самого дальнего в цепочке наследования класса. Выходные данные этого кода зависят от того, на какую реализацию .NET нацелено приложение:

- .NET Framework: выходные данные показывают, что методы завершения для трех классов вызываются автоматически при завершении работы приложения, в порядке от наиболее производного до наименее производного.
- .NET 5 (включая .NET Core) или более поздняя версия: нет выходных данных, так как эта реализация .NET не вызывает методы завершения при завершении работы приложения.

```
class First
{
    ~First()
    {
        System.Diagnostics.Trace.WriteLine("First's finalizer is called.");
    }
}

class Second : First
{
    ~Second()
    {
        System.Diagnostics.Trace.WriteLine("Second's finalizer is called.");
    }
}

class Third : Second
{
    ~Third()
    {
        System.Diagnostics.Trace.WriteLine("Third's finalizer is called.");
    }
}

/*
Test with code like the following:
    Third t = new Third();
    t = null;

When objects are finalized, the output would be:
Third's finalizer is called.
Second's finalizer is called.
First's finalizer is called.
*/
```

Спецификация языка C#

Дополнительные сведения см. в разделе Деструкторыспецификация языка C# 6.0.

См. также

- IDisposable
- Руководство по программированию на C#
- Конструкторы
- Сборка мусора

Инициализаторы объектов и коллекций (Руководство по программированию в C#)

18.12.2021 • 8 minutes to read

C# позволяет создать экземпляр объекта или коллекции и выполнять присваивания его членов в одной инструкции.

Инициализаторы объектов

Инициализаторы объектов позволяют присваивать значения всем доступным полям и свойствам объекта во время создания без вызова конструктора, за которым следуют строки операторов присваивания. Синтаксис инициализатора объекта позволяет задавать аргументы конструктора или опускать их (и синтаксис в скобках). В приведенном ниже примере демонстрируется использование инициализатора объекта с именованным типом `Cat` и вызов конструктора без параметров. Обратите внимание на использование в классе `Cat` автоматически внедренных свойств. Дополнительные сведения см. в разделе Автоматически реализуемые свойства.

```
public class Cat
{
    // Auto-implemented properties.
    public int Age { get; set; }
    public string Name { get; set; }

    public Cat()
    {
    }

    public Cat(string name)
    {
        this.Name = name;
    }
}
```

```
Cat cat = new Cat { Age = 10, Name = "Fluffy" };
Cat sameCat = new Cat("Fluffy"){ Age = 10 };
```

Синтаксис инициализаторов объектов позволяет создать экземпляр, а затем присваивает созданный объект, включая назначенные ему свойства, переменной в назначении.

Начиная с C# 6 инициализаторы объектов могут задавать индексаторы, кроме назначения полей и свойств. Рассмотрим следующий базовый класс `Matrix`:

```
public class Matrix
{
    private double[,] storage = new double[3, 3];

    public double this[int row, int column]
    {
        // The embedded array will throw out of range exceptions as appropriate.
        get { return storage[row, column]; }
        set { storage[row, column] = value; }
    }
}
```

Можно инициализировать единичную матрицу следующим кодом:

```
var identity = new Matrix
{
    [0, 0] = 1.0,
    [0, 1] = 0.0,
    [0, 2] = 0.0,

    [1, 0] = 0.0,
    [1, 1] = 1.0,
    [1, 2] = 0.0,

    [2, 0] = 0.0,
    [2, 1] = 0.0,
    [2, 2] = 1.0,
};
```

Любой доступный индексатор, который содержит доступный метод задания, можно использовать как одно из выражений в инициализаторе объекта независимо от количества или типов аргументов. Аргументы индекса формируют левую часть присваивания, а значение стоит в его правой части. Например, все нижеприведенные конструкции допустимы, если `IndexersExample` имеет соответствующие индексаторы:

```
var thing = new IndexersExample {
    name = "object one",
    [1] = '1',
    [2] = '4',
    [3] = '9',
    Size = Math.PI,
    ['C',4] = "Middle C"
}
```

Для компиляции приведенного выше кода тип `IndexersExample` должен иметь следующие члены:

```
public string name;
public double Size { set { ... }; }
public char this[int i] { set { ... }; }
public string this[char c, int i] {  set { ... }; }
```

Инициализаторы объектов с анонимными типами

Хотя инициализаторы объектов можно использовать в любом контексте, они особенно полезны в выражениях запросов LINQ. В выражениях запросов часто используются анонимные типы, которые можно инициализировать только с помощью инициализаторов объектов, как показано в приведенном ниже объявлении.

```
var pet = new { Age = 10, Name = "Fluffy" };
```

Анонимные типы позволяют предложению `select` в выражении запроса LINQ преобразовывать объекты исходной последовательности в объекты, значение и форма которых могут отличаться от исходных. Это бывает полезно, если требуется сохранить лишь часть информации от каждого объекта последовательности. В приведенном ниже примере предполагается, что у объекта продукта (`p`) имеется множество полей и методов, и требуется создать последовательность объектов, содержащую только имя продукта и его цену.

```
var productInfos =
    from p in products
    select new { p.ProductName, p.UnitPrice };
```

При выполнении этого запроса переменная `productInfos` будет содержать последовательность объектов, доступных в операторе `foreach` , как показано в следующем примере.

```
foreach(var p in productInfos){...}
```

Каждый объект нового анонимного типа содержит два общедоступных свойства, имеющих те же имена, что и у свойств или полей исходного объекта. Кроме того, при создании анонимного типа можно переименовать поле; в следующем примере выполняется переименование поля `UnitPrice` в `Price` .

```
select new {p.ProductName, Price = p.UnitPrice};
```

Инициализаторы коллекций

Инициализаторы коллекций позволяют задавать один или несколько инициализаторов элементов при инициализации типа коллекции, который реализует интерфейс IEnumerable и включает `Add` с соответствующей сигнатурой как метод экземпляра или метод расширения. Инициализаторами элементов могут быть простые значения, выражения и инициализаторы объектов. При использовании инициализатора коллекции не требуется указывать несколько вызовов; эти вызовы добавляет компилятор.

Ниже приведен пример двух простых инициализаторов коллекций.

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
List<int> digits2 = new List<int> { 0 + 1, 12 % 3, MakeInt() };
```

В следующем инициализаторе коллекции используются инициализаторы объектов класса `Cat` , определенного в предыдущем примере. Обратите внимание, что инициализаторы отдельных объектов заключены в скобки и разделены запятыми.

```
List<Cat> cats = new List<Cat>
{
    new Cat{ Name = "Sylvester", Age=8 },
    new Cat{ Name = "Whiskers", Age=2 },
    new Cat{ Name = "Sasha", Age=14 }
};
```

В качестве элемента инициализатора коллекции можно указать значение null, если метод `Add` коллекции допускает это.

```
List<Cat> moreCats = new List<Cat>
{
    new Cat{ Name = "Furrytail", Age=5 },
    new Cat{ Name = "Peaches", Age=4 },
    null
};
```

Можно указать индексированные элементы, если коллекция поддерживает индексирование чтения и записи.

```
var numbers = new Dictionary<int, string>
{
    [7] = "seven",
    [9] = "nine",
    [13] = "thirteen"
};
```

В предыдущем примере создается код, который вызывает Item[TKey] для задания значений. В версиях C# ниже 6 можно было инициализировать словари и другие ассоциативные контейнеры, используя указанный ниже синтаксис. Обратите внимание, что вместо синтаксиса индексатора с круглыми скобками и присваиванием он использует объект с несколькими значениями:

```
var moreNumbers = new Dictionary<int, string>
{
    {19, "nineteen" },
    {23, "twenty-three" },
    {42, "forty-two" }
};
```

В этом примере инициализатор вызывает Add(TKey, TValue) для добавления трех элементов в словарь. Эти два разных способа инициализации ассоциативных коллекций немного отличаются из-за вызовов методов, которые создает компилятор. Оба варианта могут работать с классом `Dictionary`. Другие типы могут поддерживать только один из них в зависимости от своего открытого API.

Инициализаторы объектов с инициализацией свойств коллекции только для чтения

Некоторые классы могут иметь свойства коллекции, где свойство доступно только для чтения, например свойство `Cats` класса `CatOwner` в следующих случаях:

```
public class CatOwner
{
    public IList<Cat> Cats { get; } = new List<Cat>();
}
```

Вы не сможете использовать описанный синтаксис инициализатора коллекции, так как этому свойству нельзя присвоить новый список:

```
CatOwner owner = new CatOwner
{
    Cats = new List<Cat>
    {
        new Cat{ Name = "Sylvester", Age=8 },
        new Cat{ Name = "Whiskers", Age=2 },
        new Cat{ Name = "Sasha", Age=14 }
    }
};
```

Тем не менее новые записи можно добавить к `Cats`, используя синтаксис инициализации без создания списка (`new List<Cat>`), как показано далее:

```
CatOwner owner = new CatOwner
{
    Cats =
    {
        new Cat{ Name = "Sylvester", Age=8 },
        new Cat{ Name = "Whiskers", Age=2 },
        new Cat{ Name = "Sasha", Age=14 }
    }
};
```

Набор добавляемых записей просто заключается в фигурные скобки. Описанное выше аналогично следующему примеру:

```
CatOwner owner = new CatOwner();
owner.Cats.Add(new Cat{ Name = "Sylvester", Age=8 });
owner.Cats.Add(new Cat{ Name = "Whiskers", Age=2 });
owner.Cats.Add(new Cat{ Name = "Sasha", Age=14 });
```

Примеры

В следующем примере объединяются понятия инициализаторов коллекций и объектов.

```
public class InitializationSample
{
    public class Cat
    {
        // Auto-implemented properties.
        public int Age { get; set; }
        public string Name { get; set; }

        public Cat() { }

        public Cat(string name)
        {
            Name = name;
        }
    }

    public static void Main()
    {
        Cat cat = new Cat { Age = 10, Name = "Fluffy" };
        Cat sameCat = new Cat("Fluffy"){ Age = 10 };

        List<Cat> cats = new List<Cat>
        {
            new Cat { Name = "Sylvester", Age = 8 },
            new Cat { Name = "Whiskers", Age = 2 },
            new Cat { Name = "Sasha", Age = 14 }
        };

        List<Cat> moreCats = new List<Cat>
        {
            new Cat { Name = "Furrytail", Age = 5 },
            new Cat { Name = "Peaches", Age = 4 },
            null
        };

        // Display results.
        System.Console.WriteLine(cat.Name);

        foreach (Cat c in cats)
            System.Console.WriteLine(c.Name);

        foreach (Cat c in moreCats)
            if (c != null)
                System.Console.WriteLine(c.Name);
            else
                System.Console.WriteLine("List element has null value.");
    }
    // Output:
    //Fluffy
    //Sylvester
    //Whiskers
    //Sasha
    //Furrytail
    //Peaches
    //List element has null value.
}
```

В примере ниже представлен объект, в котором реализован интерфейс IEnumerable. Он содержит метод `Add` с несколькими параметрами, позволяющими использовать инициализаторы коллекций с несколькими элементами для каждого пункта списка, который соответствует сигнатуре метода `Add`.

```
    public class FullExample
    {
        class FormattedAddresses : IEnumerable<string>
        {
            private List<string> internalList = new List<string>();
            public IEnumerator<string> GetEnumerator() => internalList.GetEnumerator();

            System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() =>
internalList.GetEnumerator();

            public void Add(string firstname, string lastname,
                string street, string city,
                string state, string zipcode) => internalList.Add(
                $@"{firstname} {lastname}
{street}
{city}, {state} {zipcode}"
                );
        }

        public static void Main()
        {
            FormattedAddresses addresses = new FormattedAddresses()
            {
                {"John", "Doe", "123 Street", "Topeka", "KS", "00000" },
                {"Jane", "Smith", "456 Street", "Topeka", "KS", "00000" }
            };

            Console.WriteLine("Address Entries:");

            foreach (string addressEntry in addresses)
            {
                Console.WriteLine("\r\n" + addressEntry);
            }
        }

        /*
         * Prints:

            Address Entries:

            John Doe
            123 Street
            Topeka, KS 00000

            Jane Smith
            456 Street
            Topeka, KS 00000
         */
    }
```

В методах `Add` можно использовать ключевое слово `params`, чтобы принимать переменное число аргументов, как показано в приведенном ниже примере. Здесь также демонстрируется пользовательская реализация индексатора, которая также применяется для инициализации коллекции с помощью индексов.

```
public class DictionaryExample
{
    class RudimentaryMultiValuedDictionary<TKey, TValue> : IEnumerable<KeyValuePair<TKey, List<TValue>>>
    {
        private Dictionary<TKey, List<TValue>> internalDictionary = new Dictionary<TKey, List<TValue>>();

        public IEnumerator<KeyValuePair<TKey, List<TValue>>> GetEnumerator() =>
internalDictionary.GetEnumerator();

        System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() =>
```

```
internalDictionary.GetEnumerator();

        public List<TValue> this[TKey key]
        {
            get => internalDictionary[key];
            set => Add(key, value);
        }

        public void Add(TKey key, params TValue[] values) => Add(key, (IEnumerable<TValue>)values);

        public void Add(TKey key, IEnumerable<TValue> values)
        {
            if (!internalDictionary.TryGetValue(key, out List<TValue> storedValues))
                internalDictionary.Add(key, storedValues = new List<TValue>());

            storedValues.AddRange(values);
        }
    }

    public static void Main()
    {
        RudimentaryMultiValuedDictionary<string, string> rudimentaryMultiValuedDictionary1
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                {"Group1", "Bob", "John", "Mary" },
                {"Group2", "Eric", "Emily", "Debbie", "Jesse" }
            };
        RudimentaryMultiValuedDictionary<string, string> rudimentaryMultiValuedDictionary2
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                ["Group1"] = new List<string>() { "Bob", "John", "Mary" },
                ["Group2"] = new List<string>() { "Eric", "Emily", "Debbie", "Jesse" }
            };
        RudimentaryMultiValuedDictionary<string, string> rudimentaryMultiValuedDictionary3
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                {"Group1", new string []{ "Bob", "John", "Mary" } },
                { "Group2", new string[]{ "Eric", "Emily", "Debbie", "Jesse" } }
            };

        Console.WriteLine("Using first multi-valued dictionary created with a collection initializer:");

        foreach (KeyValuePair<string, List<string>> group in rudimentaryMultiValuedDictionary1)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");

            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }

        Console.WriteLine("\r\nUsing second multi-valued dictionary created with a collection initializer
using indexing:");

        foreach (KeyValuePair<string, List<string>> group in rudimentaryMultiValuedDictionary2)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");

            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }
        Console.WriteLine("\r\nUsing third multi-valued dictionary created with a collection initializer
using indexing:");

        foreach (KeyValuePair<string, List<string>> group in rudimentaryMultiValuedDictionary3)
        {
```

```
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");

            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }
    }

    /*
     * Prints:

        Using first multi-valued dictionary created with a collection initializer:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse

        Using second multi-valued dictionary created with a collection initializer using indexing:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse

        Using third multi-valued dictionary created with a collection initializer using indexing:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse
     */
}
```

См. также

- Руководство по программированию на C#
- LINQ в C#
- Анонимные типы

Практическое руководство. Инициализация объектов с помощью инициализатора объектов (руководство по программированию на C#)

18.12.2021 • 2 minutes to read

Инициализаторы объектов можно использовать для декларативной инициализации объектов типов без явного вызова конструктора для типа.

Следующие примеры демонстрируют использование инициализаторов объектов с именованными объектами. Компилятор выполняет обработку инициализаторов объектов, сначала получая доступ к конструктору экземпляра без параметров, а затем обрабатывая инициализации членов. Таким образом, если конструктор без параметров объявляется как `private` в классе, произойдет сбой инициализаторов объектов, требующих открытого доступа.

При определении анонимного типа использовать инициализатор объекта обязательно. Дополнительные сведения см. в разделе Практическое руководство. Возвращение поднаборов свойств элементов в запросе.

Пример

В следующем примере показана инициализация нового типа `StudentName` с помощью инициализаторов объектов. В этом примере задаются свойства в типе `StudentName`:

```
public class HowToObjectInitializers
{
    public static void Main()
    {
        // Declare a StudentName by using the constructor that has two parameters.
        StudentName student1 = new StudentName("Craig", "Playstead");

        // Make the same declaration by using an object initializer and sending
        // arguments for the first and last names. The parameterless constructor is
        // invoked in processing this declaration, not the constructor that has
        // two parameters.
        StudentName student2 = new StudentName
        {
            FirstName = "Craig",
            LastName = "Playstead"
        };

        // Declare a StudentName by using an object initializer and sending
        // an argument for only the ID property. No corresponding constructor is
        // necessary. Only the parameterless constructor is used to process object
        // initializers.
        StudentName student3 = new StudentName
        {
            ID = 183
        };

        // Declare a StudentName by using an object initializer and sending
        // arguments for all three properties. No corresponding constructor is
        // defined in the class.
        StudentName student4 = new StudentName
        {
            FirstName = "Craig",
            LastName = "Playstead",
```

```
            LastName = "Feldstead",
            ID = 116
        };

        Console.WriteLine(student1.ToString());
        Console.WriteLine(student2.ToString());
        Console.WriteLine(student3.ToString());
        Console.WriteLine(student4.ToString());
    }
    // Output:
    // Craig  0
    // Craig  0
    //   183
    // Craig  116

    public class StudentName
    {
        // This constructor has no parameters. The parameterless constructor
        // is invoked in the processing of object initializers.
        // You can test this by changing the access modifier from public to
        // private. The declarations in Main that use object initializers will
        // fail.
        public StudentName() { }

        // The following constructor has parameters for two of the three
        // properties.
        public StudentName(string first, string last)
        {
            FirstName = first;
            LastName = last;
        }

        // Properties.
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int ID { get; set; }

        public override string ToString() => FirstName + "  " + ID;
    }
}
```

Инициализаторы объектов можно использовать для задания индексаторов в объекте. В следующем примере определяется класс `BaseballTeam`, который использует индексатор для получения и задания игроков в различных положениях. Инициализатор может назначать игроков в зависимости от сокращенного обозначения положения или номера, используемого для каждой позиции в бейсбольных карточках.

```
public class HowToIndexInitializer
{
    public class BaseballTeam
    {
        private string[] players = new string[9];
        private readonly List<string> positionAbbreviations = new List<string>
        {
            "P", "C", "1B", "2B", "3B", "SS", "LF", "CF", "RF"
        };

        public string this[int position]
        {
            // Baseball positions are 1 - 9.
            get { return players[position-1]; }
            set { players[position-1] = value; }
        }
        public string this[string position]
        {
            get { return players[positionAbbreviations.IndexOf(position)]; }
            set { players[positionAbbreviations.IndexOf(position)] = value; }
        }
    }

    public static void Main()
    {
        var team = new BaseballTeam
        {
            ["RF"] = "Mookie Betts",
            [4] = "Jose Altuve",
            ["CF"] = "Mike Trout"
        };

        Console.WriteLine(team["2B"]);
    }
}
```

См. также

- Руководство по программированию на C#
- Инициализаторы объектов и коллекций

Практическое руководство. Инициализация словаря с помощью инициализатора коллекции (Руководство по программированию на C#)

18.12.2021 • 2 minutes to read

Dictionary<TKey,TValue> содержит коллекцию пар "ключ-значение". Ее метод Add принимает два параметра: один для ключа, другой — для значения. Для инициализации Dictionary<TKey,TValue> или любой коллекции, метод `Add` которой принимает несколько параметров, можно заключить каждый набор параметров в скобки, как показано в приведенном ниже примере. Другой вариант — использовать инициализатор индекса, как показано также в следующем примере.

Пример

В следующем примере кода Dictionary<TKey,TValue> инициализируется экземплярами типа `StudentName`. Первая инициализация использует метод `Add` с двумя аргументами. Компилятор создает вызов метода `Add` для каждой пары ключа `int` и значения `StudentName`. Вторая инициализация использует общедоступный метод индексатора чтения и записи класса `Dictionary`:

```
public class HowToDictionaryInitializer
{
    class StudentName
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int ID { get; set; }
    }

    public static void Main()
    {
        var students = new Dictionary<int, StudentName>()
        {
            { 111, new StudentName { FirstName="Sachin", LastName="Karnik", ID=211 } },
            { 112, new StudentName { FirstName="Dina", LastName="Salimzianova", ID=317 } },
            { 113, new StudentName { FirstName="Andy", LastName="Ruth", ID=198 } }
        };

        foreach(var index in Enumerable.Range(111, 3))
        {
            Console.WriteLine($"Student {index} is {students[index].FirstName} {students[index].LastName}");
        }
        Console.WriteLine();

        var students2 = new Dictionary<int, StudentName>()
        {
            [111] = new StudentName { FirstName="Sachin", LastName="Karnik", ID=211 },
            [112] = new StudentName { FirstName="Dina", LastName="Salimzianova", ID=317 } ,
            [113] = new StudentName { FirstName="Andy", LastName="Ruth", ID=198 }
        };

        foreach (var index in Enumerable.Range(111, 3))
        {
            Console.WriteLine($"Student {index} is {students2[index].FirstName}
{students2[index].LastName}");
        }
    }
}
```

Обратите внимание на две пары фигурных скобок в каждом элементе коллекции в первом объявлении. Во внутренней паре фигурных скобок заключен инициализатор объекта `StudentName`, а во внешней паре — инициализатор пары "ключ-значение", которая будет добавлена в коллекцию `students` Dictionary<TKey,TValue>. И наконец, весь инициализатор коллекции для словаря заключается в фигурные скобки. Во второй инициализации левая часть оператора присваивания является ключом, а правая часть — значением, использующим инициализатор объекта для `StudentName`.

См. также

- Руководство по программированию на C#
- Инициализаторы объектов и коллекций

Вложенные типы (Руководство по программированию на C#)

18.12.2021 • 2 minutes to read

Тип, определенный внутри класса, структуры или интерфейса, называется вложенным типом. Пример

```
public class Container
{
    class Nested
    {
        Nested() { }
    }
}
```

Независимо от того, является ли внешний тип классом, интерфейсом или структурой, вложенным типам по умолчанию присваивается модификатор private, из-за чего они доступны только из содержащего их типа. В предыдущем примере класс `Nested` недоступен для внешних типов.

Также можно указать модификатор доступа, определяющий доступность вложенного типа, как показано ниже:

- Вложенные типы **класса** могут иметь модификаторы public, protected, internal, protected internal, private или private protected.

  Тем не менее при определении вложенного класса `protected`, `protected internal` или `private protected` внутри запечатанного класса возникает предупреждение компилятора CS0628 "Новый защищенный член объявлен в запечатанном классе".

  Также учитывайте, что внешняя доступность вложенного типа нарушает правило качества кода CA1034 "Вложенные типы не должны быть видимыми".

- Вложенные типы **структуры** могут иметь модификаторы public, internal или private.

В следующем примере класс `Nested` определяется как открытый:

```
public class Container
{
    public class Nested
    {
        Nested() { }
    }
}
```

Вложенный (внутренний) тип может получить доступ к вмещающему (внешнему) типу. Чтобы получить доступ к вмещающему типу, передайте его в качестве аргумента в конструктор вложенного типа. Пример:

```
public class Container
{
    public class Nested
    {
        private Container parent;

        public Nested()
        {
        }
        public Nested(Container parent)
        {
            this.parent = parent;
        }
    }
}
```

Вложенный тип имеет доступ ко всем членам, которые доступны вмещающему типу. Он может получать доступ к закрытым и защищенным членам вмещающего типа, включая любые наследуемые защищенные члены.

В предыдущем объявлении полным именем класса `Nested` является `Container.Nested`. Это имя используется для создания нового экземпляра вложенного класса, как показано ниже:

```
Container.Nested nest = new Container.Nested();
```

См. также

- Руководство по программированию на C#
- Система типов C#
- Модификаторы доступа
- Конструкторы
- Правило CA1034

Разделяемые классы и методы (Руководство по программированию в C#)

18.12.2021 • 5 minutes to read

Можно разделить определение класса, структуры, интерфейса или метода между двумя или более исходными файлами. Каждый исходный файл содержит часть определения класса или метода, а во время компиляции приложения все части объединяются.

Разделяемые классы

Существует несколько ситуаций, когда желательно разделение определения класса.

- При работе над большими проектами распределение класса между различными файлами позволяет нескольким программистам работать с ним одновременно.
- При работе с использованием автоматически создаваемого источника код можно добавлять в класс без повторного создания файла источника. Visual Studio использует этот подход при создании форм Windows Forms, кода оболочки веб-службы и т. д. Можно создать код, который использует эти классы, без необходимости изменения файла, созданного в Visual Studio.
- При использовании генераторов источников для создания дополнительных функциональных возможностей в классе.

Чтобы разделить определение класса, используйте модификатор ключевого слова partial, как показано ниже:

```
public partial class Employee
{
    public void DoWork()
    {
    }
}

public partial class Employee
{
    public void GoToLunch()
    {
    }
}
```

Ключевое слово `partial` указывает, что другие части класса, структуры или интерфейса могут быть определены в пространстве имен. Все части должны использовать ключевое слово `partial`. Для формирования окончательного типа все части должны быть доступны во время компиляции. Все части должны иметь одинаковые модификаторы доступа, например `public`, `private` и т. д.

Если какая-либо из частей объявлена абстрактной, то весь тип будет считаться абстрактным. Если какая-либо из частей объявлена запечатанной, то весь тип будет считаться запечатанным. Если какая-либо из частей объявляет базовый тип, то весь тип будет наследовать данный класс.

Все части, указывающие базовый класс, должны быть согласованы друг с другом, а части, не использующие базовый класс, все равно наследуют базовый тип. Части могут указывать различные базовые интерфейсы, и окончательный тип будет реализовывать все интерфейсы, перечисленные во всех разделяемых объявлениях. Любые члены класса, структуры или интерфейса, объявленные в разделяемом объявлении, доступны для всех остальных частей. Окончательный тип представляет собой комбинацию

всех частей, выполненную во время компиляции.

> **NOTE**
> Модификатор `partial` недоступен в объявлениях делегатов или перечислений.

В следующем примере показано, что вложенные типы могут быть разделяемыми, даже если тип, в который они вложены, не является разделяемым.

```
class Container
{
    partial class Nested
    {
        void Test() { }
    }

    partial class Nested
    {
        void Test2() { }
    }
}
```

Во время компиляции атрибуты определений разделяемого типа объединяются. В качестве примера рассмотрим следующие объявления:

```
[SerializableAttribute]
partial class Moon { }

[ObsoleteAttribute]
partial class Moon { }
```

Они эквивалентны следующим объявлениям:

```
[SerializableAttribute]
[ObsoleteAttribute]
class Moon { }
```

Следующие элементы объединяются из всех определений разделяемого типа:

- XML-комментарии
- интерфейсы
- атрибуты параметров универсального параметра
- атрибуты классов
- члены

В качестве примера рассмотрим следующие объявления:

```
partial class Earth : Planet, IRotate { }
partial class Earth : IRevolve { }
```

Они эквивалентны следующим объявлениям:

```
class Earth : Planet, IRotate, IRevolve { }
```

Ограничения

Имеется несколько правил, которые необходимо выполнять при работе с определениями разделяемого класса.

- Все определения разделяемого типа, являющиеся частями одного типа, должны изменяться с использованием типа `partial`. Например, следующие объявления класса приведут к появлению ошибки: [!code-csharpAllDefinitionsMustBePartials#7]
- Модификатор `partial` должен находиться непосредственно перед ключевыми словами `class`, `struct` или `interface`.
- В определениях разделяемого типа могут присутствовать вложенные разделяемые типы, что показано в следующем примере: [!code-csharpNestedPartialTypes#8]
- Все определения разделяемого типа, являющиеся частями одного и того же типа, должны быть определены в одной сборке и в одном модуле (EXE-файл или DLL-файл). Разделяемые определения не могут находиться в разных модулях.
- Имя класса и параметры универсального типа должны соответствовать всем определениям разделяемого типа. Универсальные типы могут быть разделяемыми. Все объявления разделяемого типа должны использовать одинаковые имена параметров в одном и том же порядке.
- Приведенные ниже ключевые слова необязательно должны присутствовать в определении разделяемого типа, но если они присутствуют в одном определении разделяемого типа, то не должны конфликтовать с ключевыми словами, указанными в других определениях того же разделяемого типа.
  - public
  - private
  - protected
  - internal
  - abstract
  - sealed
  - базовый класс
  - модификатор new (вложенные части)
  - универсальные ограничения

Дополнительные сведения см. в разделе Ограничения параметров типа.

Примеры

В следующем примере поля и конструктор класса `Coords` объявлены в одном определении разделяемого класса, а член `PrintCoords` — в другом определении разделяемого класса.

```
public partial class Coords
{
    private int x;
    private int y;

    public Coords(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

public partial class Coords
{
    public void PrintCoords()
    {
        Console.WriteLine("Coords: {0},{1}", x, y);
    }
}

class TestCoords
{
    static void Main()
    {
        Coords myCoords = new Coords(10, 15);
        myCoords.PrintCoords();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: Coords: 10,15
```

В следующем примере показано, что можно также разработать разделяемые структуры и интерфейсы.

```
partial interface ITest
{
    void Interface_Test();
}

partial interface ITest
{
    void Interface_Test2();
}

partial struct S1
{
    void Struct_Test() { }
}

partial struct S1
{
    void Struct_Test2() { }
}
```

Разделяемые методы

Разделяемый класс или структура могут содержать разделяемый метод. Одна часть класса содержит сигнатуру метода. В той же или в другой части можно определить реализацию. Если реализация не предоставлена, метод и все вызовы метода удаляются во время компиляции. Реализация может потребоваться в зависимости от сигнатуры метода. Разделяемый метод может не иметь реализацию в

следующих случаях.

- У него нет модификаторов доступа (включая private по умолчанию).
- Он возвращает значение void.
- У него нет параметров out.
- У него нет ни одного из следующих модификаторов: virtual, override, sealed, new или extern.

Любой метод, не соответствующий всем этим ограничениям (например, метод `public virtual partial void`), должен предоставлять реализацию. Эта реализация может быть предоставлена *генератором источника*.

Разделяемые методы позволяют разработчику одного элемента класса объявить метод. Разработчик другого элемента класса может определить этот метод. Существует два сценария, в которых выгодно использовать разделяемые методы: шаблоны, создающие стереотипный код и генераторы источника.

- **Код шаблона**: шаблон резервирует имя и подпись метода, чтобы созданный код мог вызвать метод. Эти методы придерживаются ограничений, которые позволяют разработчику решить, следует ли реализовать этот метод. Если метод не реализован, то компилятор удаляет сигнатуру метода и все вызовы этого метода. Вызовы метода, включая любые результаты, которые могли бы произойти от оценки аргументов в вызовах, не имеют эффекта во время выполнения. Таким образом, любой код в разделяемом классе может свободно использовать разделяемый метод, даже если реализация не предоставлена. Во время компиляции и выполнения программы не возникнут никакие ошибки, если метод будет вызван, но не реализован.
- **Генераторы источников**: генераторы источников предоставляют реализацию методов. Разработчик может добавить объявление метода (часто с атрибутами, считанными генератором источника). Разработчик может написать код, который вызывает эти методы. Генератор источника выполняется во время компиляции и обеспечивает реализацию. В этом сценарии не соблюдаются ограничения для разделяемых методов, которые не могут реализовываться часто.

```
// Definition in file1.cs
partial void OnNameChanged();

// Implementation in file2.cs
partial void OnNameChanged()
{
  // method body
}
```

- Объявления разделяемого метода должны начинаться с контекстного ключевого слова partial.
- Сигнатуры разделяемого метода в обеих частях разделяемого типа должны совпадать.
- Разделяемые методы могут иметь модификаторы static и unsafe.
- Разделяемые методы могут быть универсальными. Ограничения налагаются на ту часть объявления разделяемого метода, где находится определение, и могут дополнительно повторяться в разделе реализации. Имена параметров и типов параметров необязательно должны совпадать в объявлении реализации и в объявлении определения.
- Можно использовать делегат в качестве определенного и реализованного разделяемого метода, но его нельзя использовать в качестве разделяемого метода, который только определен.

Спецификация языка C#

Дополнительные сведения см. в разделе Разделяемые типы в Спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Руководство по программированию на C#
- Классы
- Типы структур
- Интерфейсы
- partial (тип)

Практическое руководство. Возвращение поднаборов свойств элементов в запросе (руководство по программированию на C#)

18.12.2021 • 2 minutes to read

Используйте анонимный тип в выражении запроса, если выполняются оба следующих условия:

- требуется возвращать только некоторые свойства каждого исходного элемента;
- не требуется хранить результаты запроса за пределами области метода, в котором выполняется запрос.

Если требуется возвращать только одно свойство или поле из каждого исходного элемента, вы можете использовать оператор "точка" в предложении `select`. Например, чтобы вернуть только `ID` для каждого элемента `student`, напишите предложение `select` следующим образом:

```
select student.ID;
```

Пример

В следующем примере показано, как использовать анонимный тип для возвращения только конкретного набора свойств каждого исходного элемента, соответствующего указанному условию.

```
private static void QueryByScore()
{
    // Create the query. var is required because
    // the query produces a sequence of anonymous types.
    var queryHighScores =
        from student in students
        where student.ExamScores[0] > 95
        select new { student.FirstName, student.LastName };

    // Execute the query.
    foreach (var obj in queryHighScores)
    {
        // The anonymous type's properties were not named. Therefore
        // they have the same names as the Student properties.
        Console.WriteLine(obj.FirstName + ", " + obj.LastName);
    }
}
/* Output:
Adams, Terry
Fakhouri, Fadi
Garcia, Cesar
Omelchenko, Svetlana
Zabokritski, Eugene
*/
```

Обратите внимание, что анонимный тип использует имена исходных элементов для соответствующих свойств, если имена не заданы. Чтобы присваивать новые имена свойствам в анонимном типе, напишите инструкцию `select` следующим образом:

```
select new { First = student.FirstName, Last = student.LastName };
```

Если вы попробуете сделать это в предыдущем примере, также необходимо изменить инструкцию `Console.WriteLine`:

```
Console.WriteLine(student.First + " " + student.Last);
```

Компиляция кода

Чтобы выполнить этот код, скопируйте и вставьте класс в консольное приложение C# с директивой `using` для пространства имен System.Linq.

См. также

- Руководство по программированию на C#
- Анонимные типы
- LINQ в C#

Явная реализация интерфейса (Руководство по программированию в C#)

18.12.2021 • 2 minutes to read

Если класс реализует два интерфейса, содержащих член с одинаковой сигнатурой, при реализации такого члена в классе оба интерфейса будут использовать этот член в качестве собственной реализации. В следующем примере все вызовы `Paint` приводят к вызову одного и того же метода. Первый пример определяет типы:

```
public interface IControl
{
    void Paint();
}
public interface ISurface
{
    void Paint();
}
public class SampleClass : IControl, ISurface
{
    // Both ISurface.Paint and IControl.Paint call this method.
    public void Paint()
    {
        Console.WriteLine("Paint method in SampleClass");
    }
}
```

Следующий пример вызывает методы:

```
SampleClass sample = new SampleClass();
IControl control = sample;
ISurface surface = sample;

// The following lines all call the same method.
sample.Paint();
control.Paint();
surface.Paint();

// Output:
// Paint method in SampleClass
// Paint method in SampleClass
// Paint method in SampleClass
```

Но может потребоваться, чтобы одна и та же реализация вызывалась для обоих интерфейсов. Чтобы вызывать разную реализацию в зависимости от используемого интерфейса, можно явно реализовать член интерфейса. Явная реализация интерфейса — это член класса, который вызывается только через указанный интерфейс. Укажите имя интерфейса и точку в качестве префикса в имени члена класса. Пример:

```
public class SampleClass : IControl, ISurface
{
    void IControl.Paint()
    {
        System.Console.WriteLine("IControl.Paint");
    }
    void ISurface.Paint()
    {
        System.Console.WriteLine("ISurface.Paint");
    }
}
```

Член класса `IControl.Paint` доступен только через интерфейс `IControl`, а `ISurface.Paint` — только через интерфейс `ISurface`. Обе реализации метода будут разделены, и ни одна из них не будет доступна в классе напрямую. Пример:

```
SampleClass sample = new SampleClass();
IControl control = sample;
ISurface surface = sample;

// The following lines all call the same method.
//sample.Paint(); // Compiler error.
control.Paint();  // Calls IControl.Paint on SampleClass.
surface.Paint();  // Calls ISurface.Paint on SampleClass.

// Output:
// IControl.Paint
// ISurface.Paint
```

Явная реализация также применяется в случаях, когда в каждом из двух интерфейсов объявляются разные члены (например, свойство и метод) с одинаковыми именами. Чтобы реализовать оба интерфейса и избежать ошибок компилятора, класс должен использовать явную реализацию либо свойства `P`, либо метода `P`, либо одновременно обоих этих членов. Пример:

```
interface ILeft
{
    int P { get;}
}
interface IRight
{
    int P();
}

class Middle : ILeft, IRight
{
    public int P() { return 0; }
    int ILeft.P { get { return 0; } }
}
```

Явная реализация интерфейса не имеет модификатора доступа, так как она недоступна в качестве члена типа, в котором определена. Вместо этого она доступна только при вызове через экземпляр интерфейса. При указании модификатора доступа для явной реализации интерфейса возникает ошибка компилятора CS0106. Дополнительные сведения см. в разделе `interface` (справочник по C#).

Начиная с версии C# 8.0, вы можете определять реализацию для членов, объявленных в интерфейсе. Если класс наследует реализацию метода от интерфейса, этот метод доступен только через ссылку типа интерфейса. Наследуемый член не отображается как часть открытого интерфейса. Следующий пример определяет реализацию по умолчанию для метода интерфейса:

```
public interface IControl
{
    void Paint() => Console.WriteLine("Default Paint method");
}
public class SampleClass : IControl
{
    // Paint() is inherited from IControl.
}
```

Следующий пример вызывает реализацию по умолчанию:

```
var sample = new SampleClass();
//sample.Paint();// "Paint" isn't accessible.
var control = sample as IControl;
control.Paint();
```

Любой класс, реализующий интерфейс `IControl`, может переопределить метод `Paint` по умолчанию либо в качестве открытого метода, либо в качестве реализации интерфейса.

См. также

- Руководство по программированию на C#
- Объектно-ориентированное программирование
- Интерфейсы
- Наследование

Руководство по программированию на C#. Явная реализация элементов интерфейса

18.12.2021 • 2 minutes to read

В этом примере объявляются интерфейс `IDimensions` и класс `Box`, который явно реализует члены интерфейса `GetLength` и `GetWidth`. Доступ к членам осуществляется посредством экземпляра интерфейса `dimensions`.

Пример

```
interface IDimensions
{
    float GetLength();
    float GetWidth();
}

class Box : IDimensions
{
    float lengthInches;
    float widthInches;

    Box(float length, float width)
    {
        lengthInches = length;
        widthInches = width;
    }
    // Explicit interface member implementation:
    float IDimensions.GetLength()
    {
        return lengthInches;
    }
    // Explicit interface member implementation:
    float IDimensions.GetWidth()
    {
        return widthInches;
    }

    static void Main()
    {
        // Declare a class instance box1:
        Box box1 = new Box(30.0f, 20.0f);

        // Declare an interface instance dimensions:
        IDimensions dimensions = box1;

        // The following commented lines would produce compilation
        // errors because they try to access an explicitly implemented
        // interface member from a class instance:
        //System.Console.WriteLine("Length: {0}", box1.GetLength());
        //System.Console.WriteLine("Width: {0}", box1.GetWidth());

        // Print out the dimensions of the box by calling the methods
        // from an instance of the interface:
        System.Console.WriteLine("Length: {0}", dimensions.GetLength());
        System.Console.WriteLine("Width: {0}", dimensions.GetWidth());
    }
}
/* Output:
    Length: 30
    Width: 20
*/
```

Отказоустойчивость

- Обратите внимание, что следующие строки в методе `Main` закомментированы, поскольку при их компиляции возникнут ошибки. Член интерфейса, реализованный явным образом, недоступен из экземпляра класса:

  ```
  //System.Console.WriteLine("Length: {0}", box1.GetLength());
  //System.Console.WriteLine("Width: {0}", box1.GetWidth());
  ```

- Также обратите внимание, что следующие строки в методе `Main` успешно выводят на печать размеры поля, поскольку методы вызываются из экземпляра интерфейса:

```
System.Console.WriteLine("Length: {0}", dimensions.GetLength());
System.Console.WriteLine("Width: {0}", dimensions.GetWidth());
```

См. также

- Руководство по программированию на C#
- Объектно-ориентированное программирование
- Интерфейсы
- Практическое руководство. Явная реализация членов двух интерфейсов

Руководство по программированию на C#. Явная реализация элементов двух интерфейсов

18.12.2021 • 2 minutes to read

Явная реализация интерфейсов позволяет разработчику реализовать два интерфейса с одинаковыми именами членов и создать отдельные реализации каждого члена интерфейса. В этом примере размеры поля выводятся в метрической и английской системе мер. Класс Box реализует два интерфейса (IEnglishDimensions и IMetricDimensions), представляющие соответствующие системы мер. В обоих интерфейсах представлены члены с одинаковыми именами: Length и Width.

Пример

```
// Declare the English units interface:
interface IEnglishDimensions
{
    float Length();
    float Width();
}

// Declare the metric units interface:
interface IMetricDimensions
{
    float Length();
    float Width();
}

// Declare the Box class that implements the two interfaces:
// IEnglishDimensions and IMetricDimensions:
class Box : IEnglishDimensions, IMetricDimensions
{
    float lengthInches;
    float widthInches;

    public Box(float lengthInches, float widthInches)
    {
        this.lengthInches = lengthInches;
        this.widthInches = widthInches;
    }

    // Explicitly implement the members of IEnglishDimensions:
    float IEnglishDimensions.Length() => lengthInches;

    float IEnglishDimensions.Width() => widthInches;

    // Explicitly implement the members of IMetricDimensions:
    float IMetricDimensions.Length() => lengthInches * 2.54f;

    float IMetricDimensions.Width() => widthInches * 2.54f;

    static void Main()
    {
        // Declare a class instance box1:
        Box box1 = new Box(30.0f, 20.0f);

        // Declare an instance of the English units interface:
        IEnglishDimensions eDimensions = box1;

        // Declare an instance of the metric units interface:
        IMetricDimensions mDimensions = box1;

        // Print dimensions in English units:
        System.Console.WriteLine("Length(in): {0}", eDimensions.Length());
        System.Console.WriteLine("Width (in): {0}", eDimensions.Width());

        // Print dimensions in metric units:
        System.Console.WriteLine("Length(cm): {0}", mDimensions.Length());
        System.Console.WriteLine("Width (cm): {0}", mDimensions.Width());
    }
}
/* Output:
    Length(in): 30
    Width (in): 20
    Length(cm): 76.2
    Width (cm): 50.8
*/
```

Отказоустойчивость

Если требуется использовать единицы английской системы мер по умолчанию, реализуйте методы Length и Width обычным способом, после чего явно реализуйте методы Length и Width из интерфейса IMetricDimensions:

```
// Normal implementation:
public float Length() => lengthInches;
public float Width() => widthInches;

// Explicit implementation:
float IMetricDimensions.Length() => lengthInches * 2.54f;
float IMetricDimensions.Width() => widthInches * 2.54f;
```

В этом случае единицы английской системы мер будут доступны из экземпляра класса, а метрические единицы — из экземпляра интерфейса:

```
public static void Test()
{
    Box box1 = new Box(30.0f, 20.0f);
    IMetricDimensions mDimensions = box1;

    System.Console.WriteLine("Length(in): {0}", box1.Length());
    System.Console.WriteLine("Width (in): {0}", box1.Width());
    System.Console.WriteLine("Length(cm): {0}", mDimensions.Length());
    System.Console.WriteLine("Width (cm): {0}", mDimensions.Width());
}
```

См. также

- Руководство по программированию на C#
- Объектно-ориентированное программирование
- Интерфейсы
- Практическое руководство. Явная реализация членов интерфейса

Делегаты (Руководство по программированию на C#)

18.12.2021 • 2 minutes to read

Делегат — это тип, который представляет ссылки на методы с определенным списком параметров и типом возвращаемого значения. При создании экземпляра делегата этот экземпляр можно связать с любым методом с совместимой сигнатурой и типом возвращаемого значения. Метод можно вызвать (активировать) с помощью экземпляра делегата.

Делегаты используются для передачи методов в качестве аргументов к другим методам. Обработчики событий — это ничто иное, как методы, вызываемые с помощью делегатов. При создании пользовательского метода класс (например, элемент управления Windows) может вызывать этот метод при появлении определенного события. В следующем примере показано объявление делегата:

```
public delegate int PerformCalculation(int x, int y);
```

Делегату можно назначить любой метод из любого доступного класса или структуры, соответствующей типу делегата. Этот метод должен быть статическим методом или методом экземпляра. Такая гибкость позволяет программно изменять вызовы метода, а также включать новый код в существующие классы.

> **NOTE**
> В контексте перегрузки метода его сигнатура не содержит возвращаемое значение. Однако в контексте делегатов сигнатура метода содержит возвращаемое значение. Другими словами, метод должен иметь тот же возвращаемый тип, что и делегат.

Благодаря возможности ссылаться на метод как на параметр делегаты идеально подходят для определения методов обратного вызова. Можно написать метод, сравнивающий два объекта в приложении. Этот метод можно использовать в делегате для алгоритма сортировки. Поскольку код сравнения отделен от библиотеки, метод сортировки может быть более общим.

В C# 9 для похожих сценариев, где требуется больший контроль над соглашением о вызовах, были добавлены указатели на функции. Код, связанный с делегатом, вызывается с помощью виртуального метода, добавленного к типу делегата. Используя указатели функций, можно указать другие соглашения.

Общие сведения о делегатах

Делегаты имеют следующие свойства.

- Делегаты подобны указателям на функции в C++, но являются полностью объектно-ориентированными и, в отличие от указателей C++ на функции-члены, инкапсулируют экземпляр объекта вместе с методом.
- Делегаты допускают передачу методов в качестве параметров.
- Делегаты можно использовать для определения методов обратного вызова.
- Делегаты можно связывать друг с другом; например, при появлении одного события можно вызывать несколько методов.
- Точное соответствие методов типу делегата не требуется. Дополнительные сведения см. в разделе Использование вариативности в делегатах.

- Для краткой записи встроенных блоков кода введены лямбда-выражения. В результате компиляции лямбда-выражений (в определенном контексте) получаются типы делегатов. Дополнительные сведения о лямбда-выражениях см. в разделе Лямбда-выражения.

В этом разделе

- Использование делегатов
- Использование делегатов вместо интерфейсов (руководство по программированию на C#)
- Делегаты с именованными методами и делегаты с анонимными методами
- Использование расхождения в делегатах
- Практическое руководство. Объединение делегатов (многоадресные делегаты)
- Практическое руководство. Объявление, создание и использование делегата

Спецификация языка C#

Дополнительные сведения см. в разделе Делегаты в Спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

Главы в популярных книгах

- Делегаты, события и лямбда-выражения в справочном руководстве по C# 3.0, третье издание. Более 250 решений для программистов на C# 3.0
- Делегаты и события в статье Изучение C# 3.0: основные понятия C# 3.0

См. также раздел

- Delegate
- Руководство по программированию на C#
- События

Использование делегатов (Руководство по программированию на C#)

18.12.2021 • 4 minutes to read

Делегат — это тип, который безопасно инкапсулирует метод, схожий с указателем функции в C и C++. В отличие от указателей функций в C делегаты объектно-ориентированы, типобезопасны и безопасны. Тип делегата задается его именем. В следующем примере объявляется делегат с именем `Del`, который может инкапсулировать метод, использующий в качестве аргумента значение string и возвращающий значение void:

```
public delegate void Del(string message);
```

Объект делегата обычно создается путем предоставления имени метода, для которого делегат будет служить оболочкой, или с помощью лямбда-выражения. После создания экземпляра делегата вызов метода, выполненный в делегате передается делегатом в этот метод. Параметры, передаваемые делегату вызывающим объектом, передаются в метод, а возвращаемое методом значение (при его наличии) возвращается делегатом в вызывающий объект. Эта процедура называется вызовом делегата. Делегат, для которого создан экземпляр, можно вызвать, как если бы это был метод, для которого создается оболочка. Пример:

```
// Create a method for a delegate.
public static void DelegateMethod(string message)
{
    Console.WriteLine(message);
}
```

```
// Instantiate the delegate.
Del handler = DelegateMethod;

// Call the delegate.
handler("Hello World");
```

Типы делегатов являются производными от класса Delegate в .NET. Типы делегатов являются запечатанными — от них нельзя наследовать, а от Delegate нельзя создавать производные пользовательские классы. Поскольку созданный экземпляр делегата является объектом, его можно передавать как параметр или назначать свойству. Это позволяет методу принимать делегат в качестве параметра и вызывать делегат в дальнейшем. Эта процедура называется асинхронным обратным вызовом и обычно используется для уведомления вызывающего объекта о завершении длительной операции. Когда делегат используется таким образом, коду, использующему делегат, не требуются сведения о реализации используемого метода. Данные функциональные возможности аналогичны возможностям, предоставляемым интерфейсами инкапсуляции.

Обратный вызов также часто используется для задания настраиваемого метода сравнения и передачи этого делегата в метод сортировки. Это позволяет сделать коду вызывающего объекта частью алгоритма сортировки. В следующем примере метод использует тип `Del` тип в качестве параметра.

```
public static void MethodWithCallback(int param1, int param2, Del callback)
{
    callback("The number is: " + (param1 + param2).ToString());
}
```

Затем в данный метод можно передать созданный ранее делегат:

```
MethodWithCallback(1, 2, handler);
```

и получить следующие выходные данные в окне консоли:

```
The number is: 3
```

При использовании делегата в качестве абстракции методу `MethodWithCallback` не нужно выполнять непосредственный вызов консоли, то есть его можно создавать без учета консоли. Метод `MethodWithCallback` просто подготавливает строку и передает ее в другой метод. Это очень удобно, так как делегируемый метод может использовать любое количество параметров.

Если делегат создан в качестве оболочки для метода экземпляра, этот делегат ссылается и на экземпляр, и на метод. Делегат не имеет сведений о типе экземпляра, кроме полученных из метода, для которого он является оболочкой, поэтому делегат может ссылаться на любой тип объекта, если для этого объекта есть метод, соответствующий сигнатуре делегата. Если делегат создан в качестве оболочки для статического метода, он ссылается только на метод. Рассмотрим следующее объявление:

```
public class MethodClass
{
    public void Method1(string message) { }
    public void Method2(string message) { }
}
```

Вместе с рассмотренным ранее статическим методом `DelegateMethod` есть три метода, для которых можно создать оболочку с помощью экземпляра `Del`.

При вызове делегат может вызывать сразу несколько методов. Это называется многоадресностью. Чтобы добавить в список методов делегата (список вызова) дополнительный метод, необходимо просто добавить два делегата с помощью оператора сложения или назначения сложения ("+" или "+="). Пример:

```
var obj = new MethodClass();
Del d1 = obj.Method1;
Del d2 = obj.Method2;
Del d3 = DelegateMethod;

//Both types of assignment are valid.
Del allMethodsDelegate = d1 + d2;
allMethodsDelegate += d3;
```

На данном этапе список вызова делегата `allMethodsDelegate` содержит три метода — `Method1`, `Method2` и `DelegateMethod`. Три исходных делегата `d1`, `d2` и `d3` остаются без изменений. При вызове `allMethodsDelegate` все три метода вызываются по порядку. Если делегат использует параметры, передаваемые по ссылке, эта ссылка передается после каждого из трех методов, а все изменения одного из методов становятся видны в следующем методе. Если любой из методов вызывает неперехваченное исключение, это исключение передается в вызывающий делегат объект, а последующие методы в списке вызова не вызываются. Если делегат имеет возвращаемое значение и (или) выходные параметры, он

возвращает возвращаемое значение и параметры последнего вызванного метода. Чтобы удалить метод из списка вызовов, используйте вычитание или операторы присваивания вычитания (`-` или `-=`). Пример:

```
//remove Method1
allMethodsDelegate -= d1;

// copy AllMethodsDelegate while removing d2
Del oneMethodDelegate = allMethodsDelegate - d2;
```

Поскольку типы делегата являются производными от `System.Delegate` , в делегате можно вызывать методы и свойства, определенные этим классом. Например, чтобы определить число методов в списке вызова делегата, можно использовать код:

```
int invocationCount = d1.GetInvocationList().GetLength(0);
```

Делегаты, в списке вызова которых находятся несколько методов, является производным от MulticastDelegate, являющегося подклассом класса `System.Delegate` . Приведенный выше код работает в любом из случаев, так как оба класса поддерживают `GetInvocationList` .

Групповые делегаты часто используются при обработке событий. Объекты источников событий отправляют уведомления объектам получателей, зарегистрированным для получения данного события. Чтобы зарегистрироваться для получения события, объект получателя создает метод, предназначенный для обработки этого события, затем создает делегат для этого метода и передает его в источник события. Когда происходит событие, источник вызывает делегат. После этого делегат вызывает в объекте получателя обработки события, предоставив ему данные события. Тип делегата для данного события задается источником события. Дополнительные сведения см. в разделе События.

Назначение сравнения делегатов двух различных типов во время компиляции вызовет ошибку компиляции. Если экземпляры делегата статически относятся к типу `System.Delegate` , сравнение допустимо, но во время выполнения будет возвращено значение false. Пример:

```
delegate void Delegate1();
delegate void Delegate2();

static void method(Delegate1 d, Delegate2 e, System.Delegate f)
{
    // Compile-time error.
    //Console.WriteLine(d == e);

    // OK at compile-time. False if the run-time type of f
    // is not the same as that of d.
    Console.WriteLine(d == f);
}
```

См. также раздел

- Руководство по программированию на C#
- Делегаты
- Использование расхождения в делегатах
- Расхождение в делегатах
- Использование расхождения в универсальных методах-делегатах Func и Action
- События

Делегаты с именованными методами и Анонимные методы (Руководство по программированию в C#)

18.12.2021 • 2 minutes to read

Делегат можно связать с именованным методом. При создании экземпляра делегата с использованием именованного метода этот метод передается в качестве параметра, например:

```
// Declare a delegate.
delegate void Del(int x);

// Define a named method.
void DoWork(int k) { /* ... */ }

// Instantiate the delegate using the method as a parameter.
Del d = obj.DoWork;
```

Это называется использованием именованного метода. Делегаты, сконструированные с использованием именованного метода, могут инкапсулировать либо статический метод, либо метод экземпляра. Именованные методы являются единственным способом создать экземпляр делегата в ранних версиях C#. Тем не менее в ситуациях, когда создание нового метода нежелательно, C# позволяет создать экземпляр делегата напрямую, указав код блока, который делегат будет обрабатывать при его вызове. Этот блок может содержать лямбда-выражение или анонимный метод.

Метод, который передается как параметр делегата, должен иметь такую же сигнатуру, что и объявление делегата. Экземпляр делегата может инкапсулировать статический метод или метод экземпляра.

> **NOTE**
> Несмотря на то, что делегат может использовать параметр out, не рекомендуется делать это для делегатов многоадресных событий, поскольку в этом случае невозможно определить, какой делегат будет вызван.

Начиная с C# 10 группы методов с одной перегрузкой имеют *естественный тип*. Это означает, что компилятор может вывести тип возвращаемого значения и типы параметров для типа делегата:

```
var read = Console.Read; // Just one overload; Func<int> inferred
var write = Console.Write; // ERROR: Multiple overloads, can't choose
```

Примеры

Ниже приведен простой пример объявления и использования делегата. Обратите внимание, что делегат `Del` и связанный с ним метод `MultiplyNumbers` имеют одинаковые сигнатуры.

```
// Declare a delegate
delegate void Del(int i, double j);

class MathClass
{
    static void Main()
    {
        MathClass m = new MathClass();

        // Delegate instantiation using "MultiplyNumbers"
        Del d = m.MultiplyNumbers;

        // Invoke the delegate object.
        Console.WriteLine("Invoking the delegate using 'MultiplyNumbers':");
        for (int i = 1; i <= 5; i++)
        {
            d(i, 2);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    // Declare the associated method.
    void MultiplyNumbers(int m, double n)
    {
        Console.Write(m * n + " ");
    }
}
/* Output:
    Invoking the delegate using 'MultiplyNumbers':
    2 4 6 8 10
*/
```

В следующем примере делегат, сопоставленный одновременно со статическим методом и методом экземпляра, возвращает информацию из каждого из них.

```
// Declare a delegate
delegate void Del();

class SampleClass
{
    public void InstanceMethod()
    {
        Console.WriteLine("A message from the instance method.");
    }

    static public void StaticMethod()
    {
        Console.WriteLine("A message from the static method.");
    }
}

class TestSampleClass
{
    static void Main()
    {
        var sc = new SampleClass();

        // Map the delegate to the instance method:
        Del d = sc.InstanceMethod;
        d();

        // Map to the static method:
        d = SampleClass.StaticMethod;
        d();
    }
}
/* Output:
    A message from the instance method.
    A message from the static method.
*/
```

См. также

- Руководство по программированию на C#
- Делегаты
- Практическое руководство. Объединение делегатов (многоадресные делегаты)
- События

Практическое руководство. Объединение делегатов (многоадресные делегаты) (руководство по программированию на C#)

18.12.2021 • 2 minutes to read

В этом примере показано создание многоадресных делегатов. Объекты делегатов обладают полезным свойством, благодаря которому одному экземпляру делегата с помощью оператора `+` можно присвоить несколько объектов. Многоадресный делегат содержит список присвоенных ему делегатов. При вызове многоадресного делегата поочередно вызываются представленные в его списке делегаты. Объединять можно только делегаты одного типа.

С помощью оператора `-` можно удалить делегат, входящий в состав многоадресного делегата.

Пример

```
using System;

// Define a custom delegate that has a string parameter and returns void.
delegate void CustomDel(string s);

class TestClass
{
    // Define two methods that have the same signature as CustomDel.
    static void Hello(string s)
    {
        Console.WriteLine($"  Hello, {s}!");
    }

    static void Goodbye(string s)
    {
        Console.WriteLine($"  Goodbye, {s}!");
    }

    static void Main()
    {
        // Declare instances of the custom delegate.
        CustomDel hiDel, byeDel, multiDel, multiMinusHiDel;

        // In this example, you can omit the custom delegate if you
        // want to and use Action<string> instead.
        //Action<string> hiDel, byeDel, multiDel, multiMinusHiDel;

        // Create the delegate object hiDel that references the
        // method Hello.
        hiDel = Hello;

        // Create the delegate object byeDel that references the
        // method Goodbye.
        byeDel = Goodbye;

        // The two delegates, hiDel and byeDel, are combined to
        // form multiDel.
        multiDel = hiDel + byeDel;

        // Remove hiDel from the multicast delegate, leaving byeDel,
        // which calls only the method Goodbye.
        multiMinusHiDel = multiDel - hiDel;

        Console.WriteLine("Invoking delegate hiDel:");
        hiDel("A");
        Console.WriteLine("Invoking delegate byeDel:");
        byeDel("B");
        Console.WriteLine("Invoking delegate multiDel:");
        multiDel("C");
        Console.WriteLine("Invoking delegate multiMinusHiDel:");
        multiMinusHiDel("D");
    }
}
/* Output:
Invoking delegate hiDel:
  Hello, A!
Invoking delegate byeDel:
  Goodbye, B!
Invoking delegate multiDel:
  Hello, C!
  Goodbye, C!
Invoking delegate multiMinusHiDel:
  Goodbye, D!
*/
```

См. также

- MulticastDelegate
- Руководство по программированию на C#
- События

Руководство по программированию на C#. Объявление, создание экземпляра и использование делегата

18.12.2021 • 4 minutes to read

В C# 1.0 и более поздних версий делегаты можно объявлять так, как показано в следующем примере.

```
// Declare a delegate.
delegate void Del(string str);

// Declare a method with the same signature as the delegate.
static void Notify(string name)
{
    Console.WriteLine($"Notification received for: {name}");
}
```

```
// Create an instance of the delegate.
Del del1 = new Del(Notify);
```

В C# 2.0 предоставлен более простой способ для записи предыдущего объявления, который показан в следующем примере.

```
// C# 2.0 provides a simpler way to declare an instance of Del.
Del del2 = Notify;
```

В C# 2.0 и более поздних версиях можно использовать анонимный метод для объявления и инициализации делегата, как показано в следующем примере.

```
// Instantiate Del by using an anonymous method.
Del del3 = delegate(string name)
    { Console.WriteLine($"Notification received for: {name}"); };
```

В C# 3.0 и более поздних версиях можно объявлять делегаты и создавать для них экземпляры с помощью лямбда-выражения, как показано в следующем примере.

```
// Instantiate Del by using a lambda expression.
Del del4 = name =>  { Console.WriteLine($"Notification received for: {name}"); };
```

Дополнительные сведения см. в разделе Лямбда-выражения.

Следующий пример демонстрирует объявление, создание экземпляра и использование делегата. Класс `BookDB` инкапсулирует базу данных книжного магазина, в которой хранится информация о книгах. Он предоставляет метод `ProcessPaperbackBooks`, который находит в базе данных все книги в мягкой обложке и вызывает делегат для каждой из них. Используется тип `delegate` с именем `ProcessBookCallback`. Класс `Test` использует этот класс для печати заголовков и средней цены книг в мягкой обложке.

Использование делегата помогает правильно разделить функции между базой данных книжного магазина и кодом клиента. Код клиента не имеет сведений о том, как хранятся книги и как код книжного

магазина находит книги в мягкой обложке. Код книжного магазина не имеет сведений о том, какая обработка выполняется для найденных книг в мягкой обложке.

Пример

```
// A set of classes for handling a bookstore:
namespace Bookstore
{
    using System.Collections;

    // Describes a book in the book list:
    public struct Book
    {
        public string Title;        // Title of the book.
        public string Author;       // Author of the book.
        public decimal Price;       // Price of the book.
        public bool Paperback;      // Is it paperback?

        public Book(string title, string author, decimal price, bool paperBack)
        {
            Title = title;
            Author = author;
            Price = price;
            Paperback = paperBack;
        }
    }

    // Declare a delegate type for processing a book:
    public delegate void ProcessBookCallback(Book book);

    // Maintains a book database.
    public class BookDB
    {
        // List of all books in the database:
        ArrayList list = new ArrayList();

        // Add a book to the database:
        public void AddBook(string title, string author, decimal price, bool paperBack)
        {
            list.Add(new Book(title, author, price, paperBack));
        }

        // Call a passed-in delegate on each paperback book to process it:
        public void ProcessPaperbackBooks(ProcessBookCallback processBook)
        {
            foreach (Book b in list)
            {
                if (b.Paperback)
                    // Calling the delegate:
                    processBook(b);
            }
        }
    }
}

// Using the Bookstore classes:
namespace BookTestClient
{
    using Bookstore;

    // Class to total and average prices of books:
    class PriceTotaller
    {
        int countBooks = 0;
        decimal priceBooks = 0.0m;

        internal void AddBookToTotal(Book book)
```

```
        internal void AddBookToTotal(Book book)
        {
            countBooks += 1;
            priceBooks += book.Price;
        }

        internal decimal AveragePrice()
        {
            return priceBooks / countBooks;
        }
    }

    // Class to test the book database:
    class Test
    {
        // Print the title of the book.
        static void PrintTitle(Book b)
        {
            Console.WriteLine($"   {b.Title}");
        }

        // Execution starts here.
        static void Main()
        {
            BookDB bookDB = new BookDB();

            // Initialize the database with some books:
            AddBooks(bookDB);

            // Print all the titles of paperbacks:
            Console.WriteLine("Paperback Book Titles:");

            // Create a new delegate object associated with the static
            // method Test.PrintTitle:
            bookDB.ProcessPaperbackBooks(PrintTitle);

            // Get the average price of a paperback by using
            // a PriceTotaller object:
            PriceTotaller totaller = new PriceTotaller();

            // Create a new delegate object associated with the nonstatic
            // method AddBookToTotal on the object totaller:
            bookDB.ProcessPaperbackBooks(totaller.AddBookToTotal);

            Console.WriteLine("Average Paperback Book Price: ${0:#.##}",
                    totaller.AveragePrice());
        }

        // Initialize the book database with some test books:
        static void AddBooks(BookDB bookDB)
        {
            bookDB.AddBook("The C Programming Language", "Brian W. Kernighan and Dennis M. Ritchie", 19.95m,
true);
            bookDB.AddBook("The Unicode Standard 2.0", "The Unicode Consortium", 39.95m, true);
            bookDB.AddBook("The MS-DOS Encyclopedia", "Ray Duncan", 129.95m, false);
            bookDB.AddBook("Dogbert's Clues for the Clueless", "Scott Adams", 12.00m, true);
        }
    }
}
/* Output:
Paperback Book Titles:
   The C Programming Language
   The Unicode Standard 2.0
   Dogbert's Clues for the Clueless
Average Paperback Book Price: $23.97
*/
```

Отказоустойчивость

- Объявление делегата.

  Следующая инструкция объявляет новый тип делегата.

  ```
  public delegate void ProcessBookCallback(Book book);
  ```

  Каждый тип делегата описывает количество аргументов и их типы, а также тип возвращаемого значения для всех методов, которые он может инкапсулировать. Каждый раз, когда требуется новый набор типов аргументов или новый тип возвращаемого значения, нужно объявить новый тип делегата.

- Создания экземпляра делегата.

  После объявления типа делегата нужно создать объект этого делегата и связать его с определенным методом. Продолжая предыдущий пример, вы можете передать метод `PrintTitle` в метод `ProcessPaperbackBooks`, как показано в следующем примере:

  ```
  bookDB.ProcessPaperbackBooks(PrintTitle);
  ```

  Будет создан новый объект делегата, связанный со статическим методом `Test.PrintTitle`. Аналогичным образом, в следующем примере передается нестатический метод `AddBookToTotal` для объекта `totaller`:

  ```
  bookDB.ProcessPaperbackBooks(totaller.AddBookToTotal);
  ```

  В обоих случаях новый объект делегата передается в метод `ProcessPaperbackBooks`.

  После создания делегата невозможно изменить метод, с которым он связан. Объекты делегатов являются неизменяемыми.

- Использование делегатов

  Обычно после создания объекта делегата он передается в другой код, который будет использовать этот делегат. Для вызова объекта делегата используется имя этого объекта, за которым следуют параметризованные аргументы, которые нужно передать делегату. Ниже показан пример использования делегата.

  ```
  processBook(b);
  ```

  Делегат можно вызвать синхронно, как показано в этом примере, или асинхронно при помощи методов `BeginInvoke` и `EndInvoke`.

См. также

- Руководство по программированию на C#
- События
- Делегаты

Массивы (Руководство по программированию на C#)

18.12.2021 • 2 minutes to read

В структуре данных массива можно хранить несколько переменных одного типа. Чтобы объявить массив, следует указать тип его элементов. Если требуется, чтобы массив мог хранить элементы любого типа, можно указать `object` в качестве его типа. В унифицированной системе типов C# все типы, стандартные и определяемые пользователем, ссылочные типы и типы значений напрямую или косвенно наследуются из Object.

```
type[] arrayName;
```

Пример

В следующих примерах создаются одномерные массивы, многомерные массивы и массивы массивов:

```
class TestArraysClass
{
    static void Main()
    {
        // Declare a single-dimensional array of 5 integers.
        int[] array1 = new int[5];

        // Declare and set array element values.
        int[] array2 = new int[] { 1, 3, 5, 7, 9 };

        // Alternative syntax.
        int[] array3 = { 1, 2, 3, 4, 5, 6 };

        // Declare a two dimensional array.
        int[,] multiDimensionalArray1 = new int[2, 3];

        // Declare and set array element values.
        int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };

        // Declare a jagged array.
        int[][] jaggedArray = new int[6][];

        // Set the values of the first array in the jagged array structure.
        jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
    }
}
```

Общие сведения о массивах

Массив имеет следующие свойства:

- Массив может быть одномерным, многомерным или массивом массивов.
- Количество измерений и длина каждого из измерений задаются, когда создается экземпляр массива. Эти значения нельзя изменить во время существования экземпляра.
- Используемые по умолчанию значения числовых элементов массива равны нулю, и элементам ссылки присвоено значение `null`.

- В массиве массивов элементы являются ссылочными типами и инициализируются значением `null`.
- Массивы индексируются от нуля: массив с `n` элементами индексируется от `0` до `n-1`.
- Элементы массива могут иметь любой тип, в том числе тип массива.
- Типы массивов — это ссылочные типы, производные от абстрактного базового типа Array. Все массивы реализуют IList и IEnumerable. Для итерации по массиву можно использовать оператор foreach. Одномерные массивы также реализуют IList<T> и IEnumerable<T>.

Поведение значения по умолчанию

- Для типов значений элементы массива инициализируются со значением по умолчанию — шаблоном с нулевыми битами; элементы будут иметь значение `0`.
- Все ссылочные типы (в том числе не допускающие значения NULL) имеют значения `null`.
- Для типов, допускающих значения NULL, параметр `HasValue` имеет значение `false`, а для элементов будет установлено значение `null`.

Массивы как объекты

В C# массивы представляют собой реальные объекты, а не просто адресуемые области непрерывной памяти, как в C и C++. Array — это абстрактный базовый тип для всех типов массивов. Вы можете использовать свойства и другие члены класса, входящие в Array. Например, с помощью свойства Length можно получить длину массива. В следующем коде значение длины массива `numbers` (`5`) присваивается переменной с именем `lengthOfNumbers`:

```
int[] numbers = { 1, 2, 3, 4, 5 };
int lengthOfNumbers = numbers.Length;
```

В классе Array представлено множество других полезных методов и свойств для сортировки, поиска и копирования массивов. В следующем примере свойство Rank используется для отображения количества измерений массива.

```
class TestArraysClass
{
    static void Main()
    {
        // Declare and initialize an array.
        int[,] theArray = new int[5, 10];
        System.Console.WriteLine("The array has {0} dimensions.", theArray.Rank);
    }
}
// Output: The array has 2 dimensions.
```

См. также раздел

- Как использовать одномерные массивы
- Как использовать многомерные массивы
- Как использовать массивы массивов
- Использование оператора foreach с массивами
- Передача массивов в качестве аргументов
- Неявно типизированные массивы
- Руководство по программированию на C#
- Коллекции

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

Одномерные массивы (Руководство по программированию на C#)

18.12.2021 • 2 minutes to read

Для создания одномерного массива используется оператор new и указывается тип элементов массива и число элементов. В следующем примере показано объявление массива, содержащего пять целых чисел:

```
int[] array = new int[5];
```

Этот массив содержит элементы с `array[0]` по `array[4]`. Элементы массива инициализируются до значения по умолчанию для типа элемента. Для целых чисел это `0`.

Массивы могут хранить любой указанный тип элемента. Например, в следующем примере приводится объявление массива строк:

```
string[] stringArray = new string[6];
```

Инициализация массива

Элементы массива можно инициализировать при объявлении. В этом случае не требуется спецификатор длины, поскольку он уже задан по числу элементов в списке инициализации. Пример:

```
int[] array1 = new int[] { 1, 3, 5, 7, 9 };
```

Ниже приведено объявление массива строк, где каждый элемент массива инициализируется с использованием названия дня:

```
string[] weekDays = new string[] { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

Если массив инициализируется при объявлении, можно не использовать выражение `new` и тип массива, как показано в следующем коде. Такой массив называется неявно типизированным:

```
int[] array2 = { 1, 3, 5, 7, 9 };
string[] weekDays2 = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

Переменную массива можно объявить без ее создания, но при присвоении нового массива этой переменной необходимо использовать оператор `new`. Пример:

```
int[] array3;
array3 = new int[] { 1, 3, 5, 7, 9 };   // OK
//array3 = {1, 3, 5, 7, 9};   // Error
```

Массивы типов значений и ссылочных типов

Рассмотрим следующее объявление массива:

```
SomeType[] array4 = new SomeType[10];
```

Результат этого оператора зависит от того, является ли `SomeType` типом значения или ссылочным типом. Если это тип значения, оператор создает массив из 10 элементов, каждый из которых имеет тип `SomeType`. Если `SomeType` является ссылочным типом, этот оператор создает массив из 10 элементов, каждый из которых инициализируется с использованием ссылки NULL. В обоих случаях элементы инициализируются до значения по умолчанию для типа элемента. См. дополнительные сведения о типах значений и ссылочных типах.

Получение данных из массива

Получить данные из массива можно с помощью индекса. Пример:

```
string[] weekDays2 = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };

Console.WriteLine(weekDays2[0]);
Console.WriteLine(weekDays2[1]);
Console.WriteLine(weekDays2[2]);
Console.WriteLine(weekDays2[3]);
Console.WriteLine(weekDays2[4]);
Console.WriteLine(weekDays2[5]);
Console.WriteLine(weekDays2[6]);

/*Output:
Sun
Mon
Tue
Wed
Thu
Fri
Sat
*/
```

См. также

- Array
- Массивы
- Многомерные массивы
- Массивы массивов

Многомерные массивы (Руководство по программированию на C#)

18.12.2021 • 2 minutes to read

Массивы могут иметь несколько измерений. Например, следующее объявление создает двухмерный массив из четырех строк и двух столбцов.

```
int[,] array = new int[4, 2];
```

Следующее объявление создает массив из трех измерений: 4, 2 и 3.

```
int[,,] array1 = new int[4, 2, 3];
```

Инициализация массива

Массив можно инициализировать при объявлении, как показано в следующем примере.

```
// Two-dimensional array.
int[,] array2D = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// The same array with dimensions specified.
int[,] array2Da = new int[4, 2] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// A similar array with string elements.
string[,] array2Db = new string[3, 2] { { "one", "two" }, { "three", "four" },
                                        { "five", "six" } };

// Three-dimensional array.
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4, 5, 6 } },
                                { { 7, 8, 9 }, { 10, 11, 12 } } };
// The same array with dimensions specified.
int[,,] array3Da = new int[2, 2, 3] { { { 1, 2, 3 }, { 4, 5, 6 } },
                                      { { 7, 8, 9 }, { 10, 11, 12 } } };

// Accessing array elements.
System.Console.WriteLine(array2D[0, 0]);
System.Console.WriteLine(array2D[0, 1]);
System.Console.WriteLine(array2D[1, 0]);
System.Console.WriteLine(array2D[1, 1]);
System.Console.WriteLine(array2D[3, 0]);
System.Console.WriteLine(array2Db[1, 0]);
System.Console.WriteLine(array3Da[1, 0, 1]);
System.Console.WriteLine(array3D[1, 1, 2]);

// Getting the total count of elements or the length of a given dimension.
var allLength = array3D.Length;
var total = 1;
for (int i = 0; i < array3D.Rank; i++)
{
    total *= array3D.GetLength(i);
}
System.Console.WriteLine("{0} equals {1}", allLength, total);

// Output:
// 1
// 2
// 3
// 4
// 7
// three
// 8
// 12
// 12 equals 12
```

Можно также инициализировать массив без указания ранга.

```
int[,] array4 = { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
```

Чтобы объявить переменную массива без инициализации, используйте оператор `new` для присвоения массива переменной. Использование оператора `new` показано в следующем примере.

```
int[,] array5;
array5 = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };   // OK
//array5 = {{1,2}, {3,4}, {5,6}, {7,8}};   // Error
```

В следующем примере присваивается значение конкретному элементу массива.

```
array5[2, 1] = 25;
```

Аналогичным образом, в следующем примере получается значение конкретного элемента массива,

которое присваивается переменной `elementValue`.

```
int elementValue = array5[2, 1];
```

В следующем примере кода элементы массива инициализируются с использованием значений по умолчанию (кроме массивов массивов).

```
int[,] array6 = new int[10, 10];
```

См. также

- Руководство по программированию на C#
- Массивы
- Одномерные массивы
- Массивы массивов

Массивы массивов (Руководство по программированию на C#)

18.12.2021 • 2 minutes to read

Массив массивов — это массив, элементы которого являются массивами и могут быть различных размеров. Массив массивов иногда называется нерегулярным массивом. В следующих примерах показано, как объявлять и инициализировать массивы массивов, а также получать доступ к ним.

Ниже объявляется одномерный массив из трех элементов, каждый из которых является одномерным массивом целых чисел:

```
int[][] jaggedArray = new int[3][];
```

Прежде чем использовать `jaggedArray`, его элементы необходимо инициализировать. Это можно сделать следующим образом:

```
jaggedArray[0] = new int[5];
jaggedArray[1] = new int[4];
jaggedArray[2] = new int[2];
```

Каждый элемент представляет собой одномерный массив целых чисел. Первый из них содержит 5 целых чисел, второй — 4, а третий — 2.

Кроме того, с помощью инициализаторов можно заполнять элементы массива значениями (при этом вам не потребуется знать размер массива). Пример:

```
jaggedArray[0] = new int[] { 1, 3, 5, 7, 9 };
jaggedArray[1] = new int[] { 0, 2, 4, 6 };
jaggedArray[2] = new int[] { 11, 22 };
```

Также массив можно инициализировать при объявлении, как показано ниже:

```
int[][] jaggedArray2 = new int[][]
{
new int[] { 1, 3, 5, 7, 9 },
new int[] { 0, 2, 4, 6 },
new int[] { 11, 22 }
};
```

Можно использовать следующую краткую форму. Обратите внимание, что при инициализации элементов нельзя опускать оператор `new`, поскольку механизм инициализации по умолчанию для них не предусмотрен:

```
int[][] jaggedArray3 =
{
    new int[] { 1, 3, 5, 7, 9 },
    new int[] { 0, 2, 4, 6 },
    new int[] { 11, 22 }
};
```

В массиве массивов элементы являются ссылочными типами и инициализируются значением `null`.

Доступ к отдельным элементам массива можно получить способами, показанными в следующих примерах:

```
// Assign 77 to the second element ([1]) of the first array ([0]):
jaggedArray3[0][1] = 77;

// Assign 88 to the second element ([1]) of the third array ([2]):
jaggedArray3[2][1] = 88;
```

Массивы массивов и многомерные массивы можно смешивать. Ниже показаны объявление и инициализация одномерного массива массивов, элементами которого являются двухмерные массивы разного размера. Дополнительные сведения см. в разделе Многомерные массивы.

```
int[][,] jaggedArray4 = new int[3][,]
{
    new int[,] { {1,3}, {5,7} },
    new int[,] { {0,2}, {4,6}, {8,10} },
    new int[,] { {11,22}, {99,88}, {0,9} }
};
```

В этом примере демонстрируется доступ к отдельным элементам, для чего отображается значение элемента `[1,0]` первого массива (`5`):

```
System.Console.Write("{0}", jaggedArray4[0][1, 0]);
```

Метод `Length` возвращает число массивов, содержащихся в массиве массивов. Допустим, предыдущий массив был объявлен с использованием следующей строки:

```
System.Console.WriteLine(jaggedArray4.Length);
```

возвращает значение 3.

Пример

В этом примере создается массив, элементы которого являются массивами. Все элементы массива имеют разный размер.

```
class ArrayTest
{
    static void Main()
    {
        // Declare the array of two elements.
        int[][] arr = new int[2][];

        // Initialize the elements.
        arr[0] = new int[5] { 1, 3, 5, 7, 9 };
        arr[1] = new int[4] { 2, 4, 6, 8 };

        // Display the array elements.
        for (int i = 0; i < arr.Length; i++)
        {
            System.Console.Write("Element({0}): ", i);

            for (int j = 0; j < arr[i].Length; j++)
            {
                System.Console.Write("{0}{1}", arr[i][j], j == (arr[i].Length - 1) ? "" : " ");
            }
            System.Console.WriteLine();
        }
        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    Element(0): 1 3 5 7 9
    Element(1): 2 4 6 8
*/
```

См. также

- Array
- Руководство по программированию на C#
- Массивы
- Одномерные массивы
- Многомерные массивы

Использование оператора foreach с массивами (Руководство по программированию на C#)

18.12.2021 • 2 minutes to read

Оператор foreach обеспечивает простой и понятный способ итерации по элементам массива или любой перечислимой коллекции.

Для одномерных массивов оператор `foreach` обрабатывает элементы в порядке возрастания индекса, начиная с индекса 0 и заканчивая индексом `Length - 1`:

```
int[] numbers = { 4, 5, 6, 1, 2, 3, -2, -1, 0 };
foreach (int i in numbers)
{
    System.Console.Write("{0} ", i);
}
// Output: 4 5 6 1 2 3 -2 -1 0
```

Для многомерных массивов элементов обрабатываются так, что сначала увеличиваются индексы крайнего правого измерения, а затем — следующего левого и т. д. влево:

```
int[,] numbers2D = new int[3, 2] { { 9, 99 }, { 3, 33 }, { 5, 55 } };
// Or use the short form:
// int[,] numbers2D = { { 9, 99 }, { 3, 33 }, { 5, 55 } };

foreach (int i in numbers2D)
{
    System.Console.Write("{0} ", i);
}
// Output: 9 99 3 33 5 55
```

Тем не менее для управления порядком обрабатываемых элементов в многомерных массивах можно использовать вложенный цикл for.

См. также

- Array
- Руководство по программированию на C#
- Массивы
- Одномерные массивы
- Многомерные массивы
- Массивы массивов

Передача массивов в качестве аргументов (руководство по программированию на C#)

18.12.2021 • 2 minutes to read

Массивы можно передавать в качестве аргументов в параметры метода. Поскольку массивы представляют собой ссылочные типы, метод может изменять значения элементов.

Передача одномерных массивов в качестве аргументов

Инициализированный одномерный массив можно передать в метод. Например, следующий оператор передает массив в метод печати.

```
int[] theArray = { 1, 3, 5, 7, 9 };
PrintArray(theArray);
```

В следующем примере кода показана разделяемая реализация метода печати.

```
void PrintArray(int[] arr)
{
    // Method code.
}
```

Новый массив можно инициализировать и передать за один шаг, как показано в следующем примере.

```
PrintArray(new int[] { 1, 3, 5, 7, 9 });
```

Пример

В следующем примере массив строк инициализируется и передается в качестве аргумента в метод `DisplayArray` для строк. Этот метод отображает элементы массива. Затем метод `ChangeArray` размещает элементы массива в обратном порядке, а метод `ChangeArrayElements` изменяет первые три элемента массива. После возврата каждого метода метод `DisplayArray` показывает, что передача массива по значению не препятствует изменению элементов массива.

```
using System;

class ArrayExample
{
    static void DisplayArray(string[] arr) => Console.WriteLine(string.Join(" ", arr));

    // Change the array by reversing its elements.
    static void ChangeArray(string[] arr) => Array.Reverse(arr);

    static void ChangeArrayElements(string[] arr)
    {
        // Change the value of the first three array elements.
        arr[0] = "Mon";
        arr[1] = "Wed";
        arr[2] = "Fri";
    }

    static void Main()
    {
        // Declare and initialize an array.
        string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
        // Display the array elements.
        DisplayArray(weekDays);
        Console.WriteLine();

        // Reverse the array.
        ChangeArray(weekDays);
        // Display the array again to verify that it stays reversed.
        Console.WriteLine("Array weekDays after the call to ChangeArray:");
        DisplayArray(weekDays);
        Console.WriteLine();

        // Assign new values to individual array elements.
        ChangeArrayElements(weekDays);
        // Display the array again to verify that it has changed.
        Console.WriteLine("Array weekDays after the call to ChangeArrayElements:");
        DisplayArray(weekDays);
    }
}
// The example displays the following output:
//         Sun Mon Tue Wed Thu Fri Sat
//
//        Array weekDays after the call to ChangeArray:
//        Sat Fri Thu Wed Tue Mon Sun
//
//        Array weekDays after the call to ChangeArrayElements:
//        Mon Wed Fri Wed Tue Mon Sun
```

Передача многомерных массивов в качестве аргументов

Инициализированный многомерный массив можно передать в метод так же, как и одномерный массив.

```
int[,] theArray = { { 1, 2 }, { 2, 3 }, { 3, 4 } };
Print2DArray(theArray);
```

В следующем коде показано разделяемое объявление метода печати, который принимает в качестве аргумента двухмерный массив.

```
void Print2DArray(int[,] arr)
{
    // Method code.
}
```

Новый массив можно инициализировать и передать за один шаг, как показано в следующем примере.

```
Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });
```

Пример

В следующем примере инициализируется двухмерный массив целых чисел, который передается в метод `Print2DArray`. Этот метод отображает элементы массива.

```
class ArrayClass2D
{
    static void Print2DArray(int[,] arr)
    {
        // Display the array elements.
        for (int i = 0; i < arr.GetLength(0); i++)
        {
            for (int j = 0; j < arr.GetLength(1); j++)
            {
                System.Console.WriteLine("Element({0},{1})={2}", i, j, arr[i, j]);
            }
        }
    }
    static void Main()
    {
        // Pass the array as an argument.
        Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    Element(0,0)=1
    Element(0,1)=2
    Element(1,0)=3
    Element(1,1)=4
    Element(2,0)=5
    Element(2,1)=6
    Element(3,0)=7
    Element(3,1)=8
*/
```

См. также

- Руководство по программированию на C#
- Массивы
- Одномерные массивы
- Многомерные массивы
- Массивы массивов

Неявно типизированные массивы (Руководство по программированию на C#)

18.12.2021 • 2 minutes to read

Вы можете создать неявно типизированный массив, тип экземпляра которого будет определяться на основе элементов, указанных в инициализаторе массива. В отношении таких массивов действуют правила для неявно типизированных переменных. Дополнительные сведения см. в статье Неявно типизированные локальные переменные (руководство по программированию на C#).

Неявно типизированные массивы обычно используются в выражениях запросов вместе с анонимными типами, а также инициализаторами объектов и коллекций.

В следующих примерах демонстрируется создание неявно типизированного массива:

```
class ImplicitlyTypedArraySample
{
    static void Main()
    {
        var a = new[] { 1, 10, 100, 1000 }; // int[]
        var b = new[] { "hello", null, "world" }; // string[]

        // single-dimension jagged array
        var c = new[]
        {
            new[]{1,2,3,4},
            new[]{5,6,7,8}
        };

        // jagged array of strings
        var d = new[]
        {
            new[]{"Luca", "Mads", "Luke", "Dinesh"},
            new[]{"Karen", "Suma", "Frances"}
        };
    }
}
```

Обратите внимание, что в предыдущем примере с неявно типизированными массивами в левой части выражения инициализации не используются квадратные скобки. Также обратите внимание, что массивы массивов инициализируются с помощью выражения `new []`, как и одномерные массивы.

Неявно типизированные массивы в инициализаторах объектов

При создании анонимного типа, содержащего массив, этот массив необходимо неявно типизировать в инициализаторе объекта типа. В следующем примере `contacts` представляет собой неявно типизированный массив анонимных типов, каждый из которых содержит массив с именем `PhoneNumbers`. Обратите внимание, что внутри инициализаторов объектов не используется ключевое слово `var`.

```
var contacts = new[]
{
    new {
        Name = " Eugene Zabokritski",
        PhoneNumbers = new[] { "206-555-0108", "425-555-0001" }
    },
    new {
        Name = " Hanying Feng",
        PhoneNumbers = new[] { "650-555-0199" }
    }
};
```

См. также

- Руководство по программированию на C#
- Неявно типизированные локальные переменные
- Массивы
- Анонимные типы
- Инициализаторы объектов и коллекций
- var
- LINQ в C#

Строки (Руководство по программированию на C#)

18.12.2021 • 11 minutes to read

Строка — это объект типа String, значением которого является текст. Внутри программы текст хранится в виде упорядоченной коллекции объектов Char только для чтения. В конце строки C# нет нуль-символов. Поэтому строка C# может содержать любое число внедренных нуль-символов ('\0'). Свойство Length строки соответствует числу содержащихся в ней объектов `Char`, но не числу символов Юникода. Для доступа к отдельным кодовым точкам Юникода в строке используйте объект StringInfo.

Сравнение строки и System.String

В C# ключевое слово `string` является псевдонимом для String. Таким образом, `String` и `string` являются эквивалентами. Однако рекомендуется использовать предоставленный псевдоним `string`, так как он работает даже без `using System;`. Класс `String` предоставляет множество методов для безопасного создания, обработки и сравнения строк. Кроме того, язык C# перегружает некоторые операторы для упрощения типичных операций со строками. Дополнительные сведения о ключевых словах см. в статье, посвященной строкам. Дополнительные сведения о типе и его методах см. здесь: String.

Объявление и инициализация строк

Вы можете объявлять и инициализировать строки различными способами, как показано в следующем примере:

```
// Declare without initializing.
string message1;

// Initialize to null.
string message2 = null;

// Initialize as an empty string.
// Use the Empty constant instead of the literal "".
string message3 = System.String.Empty;

// Initialize with a regular string literal.
string oldPath = "c:\\Program Files\\Microsoft Visual Studio 8.0";

// Initialize with a verbatim string literal.
string newPath = @"c:\Program Files\Microsoft Visual Studio 9.0";

// Use System.String if you prefer.
System.String greeting = "Hello World!";

// In local variables (i.e. within a method body)
// you can use implicit typing.
var temp = "I'm still a strongly-typed System.String!";

// Use a const string to prevent 'message4' from
// being used to store another string value.
const string message4 = "You can't get rid of me!";

// Use the String constructor only when creating
// a string from a char*, char[], or sbyte*. See
// System.String documentation for details.
char[] letters = { 'A', 'B', 'C' };
string alphabet = new string(letters);
```

Обратите внимание, что вы не используете оператор new для создания объекта строки, за исключением случаев инициализации строки с помощью массива символов.

Инициализируйте строку с константным значением Empty для создания нового объекта String, строка которого имеет нулевую длину. Представлением строкового литерала строки с нулевой длиной является "". Если вы инициализируете строки со значением Empty вместо NULL, вы снизите вероятность появления исключения NullReferenceException. Используйте статический метод IsNullOrEmpty(String), чтобы проверить значение строки, прежде чем пытаться получить к ней доступ.

Неизменность строковых объектов

Строковые объекты являются *неизменяемыми*: их нельзя изменить после создания. Может показаться, что все методы String и операторы C# изменяют строку, но в действительности они возвращают результаты в новый строковый объект. Когда содержимое `s1` и `s2` объединяется для формирования одной строки, две исходные строки не изменяются, как показано в следующем примере. Оператор `+=` создает новую строку, которая содержит объединенное содержимое. Этот новый объект присваивается переменной `s1`, а исходный объект, который был присвоен `s1`, освобождается для сборки мусора, так как ни одна переменная не ссылается на него.

```
string s1 = "A string is more ";
string s2 = "than the sum of its chars.";

// Concatenate s1 and s2. This actually creates a new
// string object and stores it in s1, releasing the
// reference to the original object.
s1 += s2;

System.Console.WriteLine(s1);
// Output: A string is more than the sum of its chars.
```

Так как "изменение" строки на самом деле является созданием новой строки, создавать ссылки на строки следует с осторожностью. Если вы создадите ссылку на строку, а затем "измените" исходную строку, ссылка будет по-прежнему указывать на исходный объект, а не на новый объект, который был создан при изменении строки. Это поведение проиллюстрировано в следующем коде:

```
string s1 = "Hello ";
string s2 = s1;
s1 += "World";

System.Console.WriteLine(s2);
//Output: Hello
```

Сведения о создании новых строк, основанных на таких изменениях, как операции поиска и замены исходной строки, см. в инструкциях по изменению содержимого строки.

Регулярные и буквальные строковые литералы

Используйте регулярные строковые литералы, когда вам нужно внедрить escape-символы, доступные в C#, как показано в следующем примере:

```
string columns = "Column 1\tColumn 2\tColumn 3";
//Output: Column 1        Column 2        Column 3

string rows = "Row 1\r\nRow 2\r\nRow 3";
/* Output:
  Row 1
  Row 2
  Row 3
*/

string title = "\"The \u00C6olean Harp\", by Samuel Taylor Coleridge";
//Output: "The Æolean Harp", by Samuel Taylor Coleridge
```

Буквальные строковые литералы используются для удобства и читабельности, если текст строки содержит символы обратной косой черты, например в путях к файлам. Так как буквальные строки сохраняют символы новой строки как часть текста строки, их можно использовать для инициализации многострочных строк. Используйте двойные кавычки, чтобы вставить кавычки в буквальной строке. В следующем примере показаны наиболее часто используемым буквальные строки:

```
string filePath = @"C:\Users\scoleridge\Documents\";
//Output: C:\Users\scoleridge\Documents\

string text = @"My pensive SARA ! thy soft cheek reclined
    Thus on mine arm, most soothing sweet it is
    To sit beside our Cot,...";
/* Output:
My pensive SARA ! thy soft cheek reclined
   Thus on mine arm, most soothing sweet it is
   To sit beside our Cot,...
*/

string quote = @"Her name was ""Sara.""";
//Output: Her name was "Sara."
```

Escape-последовательности строк

ESCAPE-**ПОСЛЕДОВАТЕЛЬНОСТЬ**	**ИМЯ СИМВОЛА**	**КОДИРОВКА ЮНИКОД**
\'	Одинарная кавычка	0x0027
\"	Двойная кавычка	0x0022
\\	Обратная косая черта	0x005C
\0	Null	0x0000
\a	Предупреждение	0x0007
\b	Backspace	0x0008
\f	Перевод страницы	0x000C
\n	Новая строка	0x000A
\r	Возврат каретки	0x000D
\t	Горизонтальная табуляция	0x0009
\v	Вертикальная табуляция	0x000B
\u	Escape-последовательность Юникода (UTF-16)	`\uHHHH` (диапазон: 0000–FFFF; пример: `\u00E7` = "ç")
\U	Escape-последовательность Юникода (UTF-32)	`\U00HHHHHH` (диапазон: 000000–10FFFF; пример: `\U0001F47D` = "👽")
\x	Escape-последовательность Юникода аналогична "\u", она отличается только длиной переменной	`\xH[H][H][H]` (диапазон: 0–FFFF; пример: `\x00E7` или `\x0E7` или `\xE7` = "ç")

WARNING

Если вы используете escape-последовательность `\x` с менее чем четырьмя шестнадцатеричными цифрами, то когда непосредственно следующие за ней символы также являются допустимыми шестнадцатеричными цифрами (т. е. 0–9, A–F и a–f), они будут интерпретированы как часть этой escape-последовательности. Например, `\xA1` дает результат "¡", являющийся кодовой точкой U+00A1. Однако если следующий символ — "A" или "a", тогда escape-последовательность будет интерпретироваться как `\xA1A` и даст результат "■", являющийся кодовой точкой U+0A1A. В таких случаях, чтобы избежать некорректной интерпретации, указывайте все четыре шестнадцатеричных знака (например, `\x00A1`).

NOTE

Во время компиляции буквальные строки преобразуются в обычные строки с теми же escape-последовательностями. Поэтому, если вы просматриваете буквальную строку в окне контрольных значений отладчика, вы увидите escape-символы, добавленные компилятором, а не буквальную версию из исходного кода. Например, буквальная строка `@"C:\files.txt"` будет отображаться в окне контрольных значений как "C:\\files.txt".

Строки формата

Строка формата — это строка, содержимое которой можно определить динамически во время выполнения. Строки формата создаются путем внедрения *интерполированных выражений* или заполнителей внутри фигурных скобок в строке. Весь код внутри фигурных скобок (`{...}`) будет преобразован в значение и выходные данные как отформатированная строка во время выполнения. Существует два способа создания строк формата: интерполяция строк и составное форматирование.

Интерполяция строк

В C# 6.0 и более поздних версий *интерполированные строки* определяются по специальному символу `$`. Они включают интерполированные выражения в фигурных скобках. См. дополнительные сведения в интерактивном руководстве по интерполяции строк в C#.

Используйте интерполяцию для повышения удобства чтения и обслуживаемости кода. Интерполяция строк позволяет достичь тех же результатов, что и использование метода `String.Format`, но более простым и понятным способом.

```
var jh = (firstName: "Jupiter", lastName: "Hammon", born: 1711, published: 1761);
Console.WriteLine($"{jh.firstName} {jh.lastName} was an African American poet born in {jh.born}.");
Console.WriteLine($"He was first published in {jh.published} at the age of {jh.published - jh.born}.");
Console.WriteLine($"He'd be over {Math.Round((2018d - jh.born) / 100d) * 100d} years old today.");

// Output:
// Jupiter Hammon was an African American poet born in 1711.
// He was first published in 1761 at the age of 50.
// He'd be over 300 years old today.
```

Начиная с C# 10, можно использовать интерполяцию строк для инициализации константной строки, если все выражения, используемые для заполнителей, также являются константными строками.

Составное форматирование

String.Format использует заполнители в фигурных скобках, чтобы создать строку формата. В этом примере результат аналогичен выходным данным, получаемым с помощью метода интерполяции строк, описанного выше.

```
var pw = (firstName: "Phillis", lastName: "Wheatley", born: 1753, published: 1773);
Console.WriteLine("{0} {1} was an African American poet born in {2}.", pw.firstName, pw.lastName, pw.born);
Console.WriteLine("She was first published in {0} at the age of {1}.", pw.published, pw.published -
pw.born);
Console.WriteLine("She'd be over {0} years old today.", Math.Round((2018d - pw.born) / 100d) * 100d);

// Output:
// Phillis Wheatley was an African American poet born in 1753.
// She was first published in 1773 at the age of 20.
// She'd be over 300 years old today.
```

См. дополнительные сведения о типах форматирования в .NET.

Подстроки

Подстрока — это последовательность символов, содержащихся в строке. Используйте метод Substring, чтобы создать новую строку из части исходной строки. Одно вхождение подстроки или несколько можно найти с помощью метода IndexOf. Используйте метод Replace, чтобы заменить все вхождения указанной подстроки новой строкой. Как и метод Substring, метод Replace фактически возвращает новую строку и не изменяет исходную строку. См. дополнительные сведения о поиске строк и изменении содержимого строк.

```
string s3 = "Visual C# Express";
System.Console.WriteLine(s3.Substring(7, 2));
// Output: "C#"

System.Console.WriteLine(s3.Replace("C#", "Basic"));
// Output: "Visual Basic Express"

// Index values are zero-based
int index = s3.IndexOf("C");
// index = 7
```

Доступ к отдельным символам

Используя нотацию массива со значением индекса, можно получить доступ только для чтения к отдельным символам, как показано в следующем примере:

```
string s5 = "Printing backwards";

for (int i = 0; i < s5.Length; i++)
{
    System.Console.Write(s5[s5.Length - i - 1]);
}
// Output: "sdrawkcab gnitnirP"
```

Если вам необходимо изменить отдельные символы в строке и функций методов String вам недостаточно, используйте объект StringBuilder, чтобы изменить отдельные символы "на месте", а затем создайте новую строку для сохранения результатов с помощью методов StringBuilder. В следующем примере предположим, что необходимо определенным образом изменить исходную строку, а затем сохранить результаты для дальнейшего использования:

```
string question = "hOW DOES mICROSOFT wORD DEAL WITH THE cAPS lOCK KEY?";
System.Text.StringBuilder sb = new System.Text.StringBuilder(question);

for (int j = 0; j < sb.Length; j++)
{
    if (System.Char.IsLower(sb[j]) == true)
        sb[j] = System.Char.ToUpper(sb[j]);
    else if (System.Char.IsUpper(sb[j]) == true)
        sb[j] = System.Char.ToLower(sb[j]);
}
// Store the new string.
string corrected = sb.ToString();
System.Console.WriteLine(corrected);
// Output: How does Microsoft Word deal with the Caps Lock key?
```

Строки NULL и пустые строки

Пустая строка — это экземпляр объекта System.String, который содержит нуль символов. Пустые строки часто используются в различных сценариях программирования для представления пустого текстового поля. Вы можете вызывать методы для пустых строк, так как они являются допустимыми объектами System.String. Пустые строки инициализируются следующим образом:

```
string s = String.Empty;
```

В отличие от пустых строк строка NULL не ссылается на экземпляр объекта System.String, поэтому любая попытка вызвать метод для строки NULL приводит к исключению NullReferenceException. Но вы можете использовать строки NULL в операциях объединения и сравнения с другими строками. В следующих примерах показаны случаи, в которых ссылка на строку NULL вызывает и не вызывает исключение:

```
static void Main()
{
    string str = "hello";
    string nullStr = null;
    string emptyStr = String.Empty;

    string tempStr = str + nullStr;
    // Output of the following line: hello
    Console.WriteLine(tempStr);

    bool b = (emptyStr == nullStr);
    // Output of the following line: False
    Console.WriteLine(b);

    // The following line creates a new empty string.
    string newStr = emptyStr + nullStr;

    // Null strings and empty strings behave differently. The following
    // two lines display 0.
    Console.WriteLine(emptyStr.Length);
    Console.WriteLine(newStr.Length);
    // The following line raises a NullReferenceException.
    //Console.WriteLine(nullStr.Length);

    // The null character can be displayed and counted, like other chars.
    string s1 = "\x0" + "abc";
    string s2 = "abc" + "\x0";
    // Output of the following line: * abc*
    Console.WriteLine("*" + s1 + "*");
    // Output of the following line: *abc *
    Console.WriteLine("*" + s2 + "*");
    // Output of the following line: 4
    Console.WriteLine(s2.Length);
}
```

Использование класса StringBuilder для быстрого создания строк

Операции со строками в .NET хорошо оптимизированы, и в большинстве случаев они не снижают производительность. Но в некоторых сценариях, например в сплошных циклах, которые выполняются сотни и тысячи раз, операции со строками могут повлиять на производительность. Класс StringBuilder создает строковый буфер, который ускоряет работу, если программа выполняет много операций над строками. Строка StringBuilder также позволяет заново присваивать отдельные символы, что не поддерживает встроенный строковый тип данных. Например, этот код изменяет содержимое строки без создания новой строки:

```
System.Text.StringBuilder sb = new System.Text.StringBuilder("Rat: the ideal pet");
sb[0] = 'C';
System.Console.WriteLine(sb.ToString());
System.Console.ReadLine();

//Outputs Cat: the ideal pet
```

В этом примере объект StringBuilder используется для создания строки из набора числовых типов:

```
using System;
using System.Text;

namespace CSRefStrings
{
    class TestStringBuilder
    {
        static void Main()
        {
            var sb = new StringBuilder();

            // Create a string composed of numbers 0 - 9
            for (int i = 0; i < 10; i++)
            {
                sb.Append(i.ToString());
            }
            Console.WriteLine(sb);  // displays 0123456789

            // Copy one character of the string (not possible with a System.String)
            sb[0] = sb[9];

            Console.WriteLine(sb);  // displays 9123456789
            Console.WriteLine();
        }
    }
}
```

Строки, методы расширения и LINQ

Так как тип String использует IEnumerable<T>, вы можете применять методы расширения, определенные для строк в классе Enumerable. Чтобы избежать визуального загромождения, эти методы исключены из IntelliSense для типа String, но все равно доступны. Можно также использовать выражения запроса LINQ в строках. Дополнительные сведения см. в документации по LINQ и строкам.

См. также

РАЗДЕЛ	ОПИСАНИЕ
Практическое руководство. Изменение содержимого строки	Описание методов преобразования строк и изменения их содержимого.
Сравнение строк	Рассказывает, как строки сравниваются по языку, региональным параметрам и порядку.
Сцепка нескольких строк	Демонстрирует разные способы соединения нескольких строк в одну.
Анализ строк с помощью метода String.Split	Содержит примеры кода, демонстрирующие использование метода `String.Split` для анализа строк.
Практическое руководство. Поиск по строкам	Описание использования поиска для нахождения в строках определенных шаблонов или текста.
Определение представления числового значения в строке	Объясняет, как безопасно проанализировать строку, чтобы проверить, содержит ли она допустимое числовое значение.

РАЗДЕЛ	ОПИСАНИЕ
Интерполяция строк	Описывает функцию интерполяции строк, которая предоставляет удобный синтаксис для форматирования строк.
Базовые операции со строками в .NET Framework	Содержит ссылки на статьи, в которых показаны базовые операции над строками с помощью методов System.String и System.Text.StringBuilder.
Анализ строк в .NET	Описывает, как преобразовать строковые представления базовых типов .NET в экземпляры соответствующих типов.
Синтаксический анализ строк даты и времени в .NET	Показывает, как преобразовать строку, например "01/24/2008", в объект System.DateTime.
Сравнение строк в .NET Framework	Объясняет, как сравнивать строки, и содержит примеры на языках C# и Visual Basic.
Using the StringBuilder class (Использование класса StringBuilder)	Описывает создание и изменение динамических строковых объектов с помощью класса StringBuilder.
LINQ и строки	Содержит инструкции по выполнению различных операций со строками с помощью запросов LINQ.
Руководство по программированию на C#	Содержит ссылки на статьи с конструкциями программирования на C#.

Руководство по программированию в C#. Определение представления числового значения в строке

18.12.2021 • 2 minutes to read

Чтобы определить, является ли строка допустимым представлением указанного числового типа, воспользуйтесь статическим методом `TryParse`, реализуемым всеми простыми числовыми типами, например DateTime и IPAddress. В следующем примере показано, как определить, является ли число 108 допустимым типом int.

```
int i = 0;
string s = "108";
bool result = int.TryParse(s, out i); //i now = 108
```

Если строка содержит нечисловые знаки либо числовое значение слишком велико или мало для указанного типа, `TryParse` возвращает значение "false" и задает выходному параметру значение "0". В противном случае возвращается значение "true", а выходному параметру задается числовое значение строки.

NOTE

Строка может содержать только числовые знаки и оставаться недопустимой для типа, где используется метод `TryParse`. Например, "256" не является допустимым значением для `byte`, однако оно допустимо для `int`. "98,6" не является допустимым значением для `int`, однако оно допустимо для `decimal`.

Пример

В следующем примере показано использование `TryParse` со строковыми представлениями значений `long`, `byte` и `decimal`.

```
string numString = "1287543"; //"1287543.0" will return false for a long
long number1 = 0;
bool canConvert = long.TryParse(numString, out number1);
if (canConvert == true)
  Console.WriteLine("number1 now = {0}", number1);
else
  Console.WriteLine("numString is not a valid long");

byte number2 = 0;
numString = "255"; // A value of 256 will return false
canConvert = byte.TryParse(numString, out number2);
if (canConvert == true)
  Console.WriteLine("number2 now = {0}", number2);
else
  Console.WriteLine("numString is not a valid byte");

decimal number3 = 0;
numString = "27.3"; //"27" is also a valid decimal
canConvert = decimal.TryParse(numString, out number3);
if (canConvert == true)
  Console.WriteLine("number3 now = {0}", number3);
else
  Console.WriteLine("number3 is not a valid decimal");
```

Отказоустойчивость

Простые числовые типы также реализуют статический метод `Parse`, который вызывает исключение, если строка не является допустимым числом. В целом оператор `TryParse` более эффективен, поскольку если число не является допустимым, он просто возвращает значение "false".

Безопасность .NET

Для проверки данных, введенных пользователем в такие элементы управления, как текстовые поля и поля со списком, всегда следует использовать метод `TryParse` или `Parse`.

См. также

- Преобразование массива байтов в значение типа int
- Практическое руководство. Преобразование строки в число
- Преобразование из шестнадцатеричных строк в числовые типы
- Анализ числовых строк
- Типы форматирования

Индексаторы (Руководство по программированию в C#)

18.12.2021 • 2 minutes to read

Индексаторы позволяют индексировать экземпляры класса или структуры точно так же, как и массивы. Индексированное значение можно задавать или получать без явного указания типа или экземпляра элемента. Индексаторы действуют как свойства, за исключением того, что их акцессоры принимают параметры.

В следующем примере определяется универсальный класс с простыми акцессорами get и set для назначения и получения значений. Класс `Program` создает экземпляр этого класса для хранения строк.

```
using System;

class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];

   // Define the indexer to allow client code to use [] notation.
   public T this[int i]
   {
      get { return arr[i]; }
      set { arr[i] = value; }
   }
}

class Program
{
   static void Main()
   {
      var stringCollection = new SampleCollection<string>();
      stringCollection[0] = "Hello, World";
      Console.WriteLine(stringCollection[0]);
   }
}
// The example displays the following output:
//       Hello, World.
```

NOTE

Дополнительные примеры см. в разделе Связанные разделы.

Определения текста выражений

Довольно часто акцессор get или set индексатора состоит из одной инструкции, которая просто возвращает или задает значение. Члены, воплощающие выражение, предоставляют упрощенный синтаксис для поддержки такого варианта использования. Начиная с версии C# 6, доступные только для чтения индексаторы можно реализовать в виде члена, воплощающего выражение, как показано в следующем примере.

```
using System;

class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];
   int nextIndex = 0;

   // Define the indexer to allow client code to use [] notation.
   public T this[int i] => arr[i];

   public void Add(T value)
   {
      if (nextIndex >= arr.Length)
         throw new IndexOutOfRangeException($"The collection can hold only {arr.Length} elements.");
      arr[nextIndex++] = value;
   }
}

class Program
{
   static void Main()
   {
      var stringCollection = new SampleCollection<string>();
      stringCollection.Add("Hello, World");
      System.Console.WriteLine(stringCollection[0]);
   }
}
// The example displays the following output:
//       Hello, World.
```

Обратите внимание, что `=>` представляет тело выражения, а ключевое слово `get` не используется.

Начиная с версии C# 7.0, методы доступа get и set можно реализовывать в виде членов с телом в виде выражения. В этом случае необходимо указывать оба ключевых слова (`get` и `set`). Пример:

```
using System;

class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];

   // Define the indexer to allow client code to use [] notation.
   public T this[int i]
   {
      get => arr[i];
      set => arr[i] = value;
   }
}

class Program
{
   static void Main()
   {
      var stringCollection = new SampleCollection<string>();
      stringCollection[0] = "Hello, World.";
      Console.WriteLine(stringCollection[0]);
   }
}
// The example displays the following output:
//       Hello, World.
```

Общие сведения об индексаторах

- Индексаторы позволяют индексировать объекты так же, как и массивы.
- Метод доступа `get` возвращает значение. Метод доступа `set` назначает значение.
- Ключевое слово this используется для определения индексаторов.
- Ключевое слово value используется для определения значения, присваиваемого методом доступа `set`.
- Индексаторы не нужно индексировать по целому значению; пользователь может определить конкретный механизм поиска на свое усмотрение.
- Индексаторы могут быть перегружены.
- Индексаторы могут иметь более одного формального параметра, например при доступе к двумерному массиву.

Связанные разделы

- Использование индексаторов
- Индексаторы в интерфейсах
- Сравнение свойств и индексаторов
- Ограничение доступности методов доступа

Спецификация языка C#

Дополнительные сведения см. в разделе Индексаторы в Спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Руководство по программированию на C#
- Свойства

Использование индексаторов. Руководство по программированию на C#

18.12.2021 • 5 minutes to read

Применение индексаторов упрощает работу с синтаксисом, позволяя создавать классы, структуры и интерфейсы, к которым клиентские приложения могут обращаться так же, как к массиву. В этом случае компилятор создает свойство `Item` (или свойство с другим именем, если присутствует IndexerNameAttribute) и соответствующие методы доступа. Индексаторы чаще всего реализуются в типах, предназначенных преимущественно для инкапсуляции внутренней коллекции или массива. Допустим, у вас есть класс `TempRecord`, представляющий журнал с 10 измерениями температуры по шкале Фаренгейта за период в 24 часа. Этот класс содержит массив `temps` типа `float[]` для хранения значений температуры. Реализация индексатора в этом классе позволит клиентам получать доступ к значениям температуры в экземпляре `TempRecord`, используя `float temp = tempRecord[4]` вместо `float temp = tempRecord.temps[4]`. Это позволяет не только упростить синтаксис клиентских приложений, но и облегчить понимание кода класса и его предназначения другими разработчиками.

Чтобы объявить индексатор для класса или структуры, используйте ключевое слово this, как в следующем примере:

```
// Indexer declaration
public int this[int index]
{
    // get and set accessors
}
```

IMPORTANT

При объявлении индексатора для объекта автоматически создается свойство с именем `Item`. Свойство `Item` не будет доступно непосредственно из выражения доступа к члену экземпляра. Кроме того, если вы добавите к объекту с индексатором собственное свойство `Item`, возникнет ошибка компилятора CS0102. Чтобы избежать этого, используйте IndexerNameAttribute, чтобы переименовать индексатор, как описывается ниже.

Примечания

Тип индексатора и типы его параметров должны иметь по крайней мере такой же уровень доступности, как и сам индексатор. Дополнительные сведения об уровнях доступа см. в разделе Модификаторы доступа.

Дополнительные сведения об использовании индексаторов с интерфейсом см. в разделе Индексаторы интерфейса.

Сигнатура индексатора определяет число и типы его формальных параметров. В ней не указываются тип индексатора или имена его формальных параметров. Если для одного класса объявляется несколько индексаторов, они должны иметь разные сигнатуры.

Значение индексатора не классифицируется как переменная и, соответственно, не может передаваться в качестве параметра ref или out.

Чтобы присвоить индексатору имя, которое можно использовать в других языках, используйте System.Runtime.CompilerServices.IndexerNameAttribute, как показано в этом примере:

```
// Indexer declaration
[System.Runtime.CompilerServices.IndexerName("TheItem")]
public int this[int index]
{
    // get and set accessors
}
```

Этот индексатор будет иметь имя `TheItem`, поскольку оно переопределено атрибутом имени индексатора. По умолчанию используется имя индексатора `Item`.

Пример 1

В следующем примере показано, как объявить частное поле массива `temps` и индексатор. Индексатор обеспечивает прямой доступ к экземпляру `tempRecord[i]`. Вместо использования индексатора можно объявить массив как элемент public и осуществлять доступ к его элементам напрямую (`tempRecord.temps[i]`).

```
public class TempRecord
{
    // Array of temperature values
    float[] temps = new float[10]
    {
        56.2F, 56.7F, 56.5F, 56.9F, 58.8F,
        61.3F, 65.9F, 62.1F, 59.2F, 57.5F
    };

    // To enable client code to validate input
    // when accessing your indexer.
    public int Length => temps.Length;

    // Indexer declaration.
    // If index is out of range, the temps array will throw the exception.
    public float this[int index]
    {
        get => temps[index];
        set => temps[index] = value;
    }
}
```

Обратите внимание, что при определении прав доступа индексатора, например в инструкции `Console.Write`, вызывается метод доступа get. Таким образом, если метод доступа `get` отсутствует, возникает ошибка времени компиляции.

```
using System;

class Program
{
    static void Main()
    {
        var tempRecord = new TempRecord();

        // Use the indexer's set accessor
        tempRecord[3] = 58.3F;
        tempRecord[5] = 60.1F;

        // Use the indexer's get accessor
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine($"Element #{i} = {tempRecord[i]}");
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
    /* Output:
        Element #0 = 56.2
        Element #1 = 56.7
        Element #2 = 56.5
        Element #3 = 58.3
        Element #4 = 58.8
        Element #5 = 60.1
        Element #6 = 65.9
        Element #7 = 62.1
        Element #8 = 59.2
        Element #9 = 57.5
    */
}
```

Индексирование с использованием других значений

В C# тип параметра индексатора не ограничивается целочисленными значениями. Например, в качестве индексатора могут использоваться строки. Такой индексатор можно реализовать путем поиска строки в коллекции с возвратом соответствующего значения. Поскольку методы доступа можно перегружать, строковые и целочисленные версии могут сосуществовать.

Пример 2

Этот пример объявляет класс, который хранит названия дней недели. Метод доступа `get` принимает название дня в виде строкового значения и возвращает соответствующее целое число. Например, для Sunday возвращается значение 0, для Monday — 1 и т. д.

```
using System;

// Using a string as an indexer value
class DayCollection
{
    string[] days = { "Sun", "Mon", "Tues", "Wed", "Thurs", "Fri", "Sat" };

    // Indexer with only a get accessor with the expression-bodied definition:
    public int this[string day] => FindDayIndex(day);

    private int FindDayIndex(string day)
    {
        for (int j = 0; j < days.Length; j++)
        {
            if (days[j] == day)
            {
                return j;
            }
        }

        throw new ArgumentOutOfRangeException(
            nameof(day),
            $"Day {day} is not supported.\nDay input must be in the form \"Sun\", \"Mon\", etc");
    }
}
```

Пример использования 2

```
using System;

class Program
{
    static void Main(string[] args)
    {
        var week = new DayCollection();
        Console.WriteLine(week["Fri"]);

        try
        {
            Console.WriteLine(week["Made-up day"]);
        }
        catch (ArgumentOutOfRangeException e)
        {
            Console.WriteLine($"Not supported input: {e.Message}");
        }
    }
    // Output:
    // 5
    // Not supported input: Day Made-up day is not supported.
    // Day input must be in the form "Sun", "Mon", etc (Parameter 'day')
}
```

Пример 3

В этом примере объявляется класс, в котором хранятся названия дней недели с использованием перечисления System.DayOfWeek. Метод доступа `get` принимает название дня (`DayOfWeek`) в виде строкового значения и возвращает соответствующее целое число. Например, для `DayOfWeek.Sunday` возвращается 0, для `DayOfWeek.Monday` — 1 и т. д.

```
using System;
using Day = System.DayOfWeek;

class DayOfWeekCollection
{
    Day[] days =
    {
        Day.Sunday, Day.Monday, Day.Tuesday, Day.Wednesday,
        Day.Thursday, Day.Friday, Day.Saturday
    };

    // Indexer with only a get accessor with the expression-bodied definition:
    public int this[Day day] => FindDayIndex(day);

    private int FindDayIndex(Day day)
    {
        for (int j = 0; j < days.Length; j++)
        {
            if (days[j] == day)
            {
                return j;
            }
        }
        throw new ArgumentOutOfRangeException(
            nameof(day),
            $"Day {day} is not supported.\nDay input must be a defined System.DayOfWeek value.");
    }
}
```

Пример использования 3

```
using System;

class Program
{
    static void Main()
    {
        var week = new DayOfWeekCollection();
        Console.WriteLine(week[DayOfWeek.Friday]);

        try
        {
            Console.WriteLine(week[(DayOfWeek)43]);
        }
        catch (ArgumentOutOfRangeException e)
        {
            Console.WriteLine($"Not supported input: {e.Message}");
        }
    }
    // Output:
    // 5
    // Not supported input: Day 43 is not supported.
    // Day input must be a defined System.DayOfWeek value. (Parameter 'day')
}
```

Отказоустойчивость

Повысить безопасность и надежность индексаторов можно двумя способами:

- Реализуйте стратегию обработки ошибок, предусматривающую действия в ситуациях, когда из клиентского кода передается недопустимое значение индекса. В первом примере из этого раздела класс TempRecord содержит свойство Length, с помощью которого клиентский код проверяет введенное значение, прежде чем передать его в индексатор. Кроме того, код обработки ошибок

можно поместить в сам индексатор. Не забудьте задокументировать исключения, которые будут вызываться в методе доступа индексатора, для других пользователей.

- Настройте максимально ограничивающие уровни доступа для методов доступа get и set. Особенно важно сделать это для метода доступа `set`. Дополнительные сведения см. в разделе Доступность методов доступа.

См. также

- Руководство по программированию на C#
- Индексаторы
- Свойства

Индексаторы в интерфейсах (Руководство по программированию в C#)

18.12.2021 • 2 minutes to read

Индексаторы можно объявлять для интерфейса. Методы доступа индексаторов интерфейса отличаются от методов доступа индексаторов класса следующим образом:

- Методы доступа интерфейса не используют модификаторы.
- Метод доступа интерфейса обычно не имеет тела.

Методы доступа нужны, чтобы указать, доступен ли индексатор для чтения и записи, только для чтения или только для записи. Вы можете предоставить реализацию для индексатора, определенного в интерфейсе, но это случается редко. Индексаторы обычно определяют API для доступа к полям данных, а поля данных не могут быть определены в интерфейсе.

Ниже показан пример метода доступа индексатора для интерфейса:

```
public interface ISomeInterface
{
    //...

    // Indexer declaration:
    string this[int index]
    {
        get;
        set;
    }
}
```

Сигнатура индексатора должна отличаться от сигнатур любых других индексаторов, объявленных в том же интерфейсе.

Пример

Следующий пример демонстрирует реализацию индексаторов интерфейса.

```
// Indexer on an interface:
public interface IIndexInterface
{
    // Indexer declaration:
    int this[int index]
    {
        get;
        set;
    }
}

// Implementing the interface.
class IndexerClass : IIndexInterface
{
    private int[] arr = new int[100];
    public int this[int index]   // indexer declaration
    {
        // The arr object will throw IndexOutOfRange exception.
        get => arr[index];
        set => arr[index] = value;
    }
}
```

```
IndexerClass test = new IndexerClass();
System.Random rand = new System.Random();
// Call the indexer to initialize its elements.
for (int i = 0; i < 10; i++)
{
    test[i] = rand.Next();
}
for (int i = 0; i < 10; i++)
{
    System.Console.WriteLine($"Element #{i} = {test[i]}");
}

/* Sample output:
    Element #0 = 360877544
    Element #1 = 327058047
    Element #2 = 1913480832
    Element #3 = 1519039937
    Element #4 = 601472233
    Element #5 = 323352310
    Element #6 = 1422639981
    Element #7 = 1797892494
    Element #8 = 875761049
    Element #9 = 393083859
*/
```

В приведенном выше примере можно использовать явную реализацию члена интерфейса с помощью полного имени члена интерфейса. Пример

```
string IIndexInterface.this[int index]
{
}
```

Тем не менее полное имя требуется только в целях устранения неоднозначности, если класс реализует более одного интерфейса с одинаковой сигнатурой индексатора. Например, если класс `Employee` реализует два интерфейса (`ICitizen` и `IEmployee`), оба из которых имеют одинаковую сигнатуру индексатора, требуется явная реализация члена интерфейса. Это значит, что потребуется следующее объявление индексатора:

```
string IEmployee.this[int index]
{
}
```

реализует индексатор в интерфейсе `IEmployee`, тогда как следующее объявление:

```
string ICitizen.this[int index]
{
}
```

реализует индексатор в интерфейсе `ICitizen`.

См. также

- Руководство по программированию на C#
- Индексаторы
- Свойства
- Интерфейсы

Сравнение свойств и индексаторов (Руководство по программированию в C#)

18.12.2021 • 2 minutes to read

Индексаторы подобны свойствам. К методам доступа индексаторов применяются те же правила, которые определены для методов доступа к свойствам, за исключением различий, показанных в следующей таблице.

СВОЙСТВО.	**ИНДЕКСАТОР**
Позволяет вызывать методы как открытые члены данных.	Обеспечивает доступ к элементам внутренней коллекции объекта с использованием нотации массива для самого объекта.
Доступ по простому имени.	Доступ посредством индекса.
Может быть статическим членом или членом экземпляра.	Должен быть членом экземпляра.
Метод доступа get свойства не имеет параметров.	Метод доступа `get` индексатора имеет тот же список формальных параметров, что и сам индексатор.
Метод доступа set свойства содержит неявный параметр `value`.	Метод доступа `set` индексатора имеет тот же список формальных параметров, что и сам индексатор, и также должен содержать параметр value.
Поддерживает сокращенный синтаксис с использованием автоматически реализуемых свойств.	Поддерживает элементы в виде выражения для индексаторов только для получения.

См. также

- Руководство по программированию на C#
- Индексаторы
- Свойства

События (Руководство по программированию в C#)

18.12.2021 • 2 minutes to read

События позволяют классу или объекту уведомлять другие классы или объекты о возникновении каких-либо ситуаций. Класс, отправляющий (или *порождающий*) событие, называется *издателем*, а классы, принимающие (или *обрабатывающие*) событие, называются *подписчиками*.

В типичном веб-приложении или приложении Windows Forms на C# вы подписываетесь на события, вызываемые элементами управления, такими как кнопки и списки. Вы можете использовать интегрированную среду разработки (IDE) Visual C#, чтобы просмотреть события, публикуемые элементом управления, и выбрать те из них, которые необходимо обрабатывать. IDE позволяет автоматически добавлять пустой метод обработчика событий и код для подписки на событие. Дополнительные сведения см. в разделе Практическое руководство. Подписка и отмена подписки на события.

Общие сведения о событиях

События имеют следующие свойства:

- Издатель определяет, когда возникает событие; подписчики определяют, какое действие выполняется в ответ на событие.
- У события может быть несколько подписчиков. Подписчик может обрабатывать несколько событий от нескольких издателей.
- События, не имеющие подписчиков, никогда не возникают.
- Обычно события используются для оповещения о действиях пользователя, например нажатиях кнопок или выборе пунктов меню в графических пользовательских интерфейсах.
- Если событие имеет несколько подписчиков, при возникновении события обработчики событий вызываются синхронно. Сведения об асинхронном вызове событий см. в разделе Асинхронный вызов синхронных методов.
- В библиотеке классов .NET события основываются на делегате EventHandler и базовом классе EventArgs.

Связанные разделы

Дополнительные сведения можно найти в разделе

- Практическое руководство. Подписка и отмена подписки на события
- Практическое руководство. Публикация событий, соответствующих рекомендациям .NET
- Практическое руководство. Создание событий базового класса в производных классах
- Практическое руководство. Реализация событий интерфейса
- Практическое руководство. Реализация пользовательских методов доступа к событиям

Спецификация языка C#

Дополнительные сведения см. в разделе События в Спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

Главы в популярных книгах

Делегаты, события и лямбда-выражения в справочном руководстве по C# 3.0, третье издание. Более 250 решений для программистов на C# 3.0

Делегаты и события в статье Изучение C# 3.0: основные понятия C# 3.0

См. также раздел

- EventHandler
- Руководство по программированию на C#
- Делегаты
- Создание обработчиков событий в Windows Forms

Руководство по программированию на C#. Подписка и отмена подписки на события

18.12.2021 • 2 minutes to read

Необходимость подписки на событие, опубликованное другим классом, может возникнуть, когда требуется написать пользовательский код, вызываемый при инициировании такого события. Например, можно подписаться на событие кнопки `click`, чтобы приложение выполняло некоторое действие при нажатии пользователем кнопки.

Подписка на события в интегрированной среде разработки Visual Studio

1. Если окно **Свойства** закрыто, в представлении **Конструктор** щелкните правой кнопкой мыши форму или элемент управления, для которого требуется создать обработчик событий, и выберите пункт **Свойства**.

2. Вверху окна **Свойства** щелкните значок **События**.

3. Дважды щелкните событие, которое требуется создать, например событие `Load`.

   Visual C# создаст пустой метод обработчика событий и добавит его в код. Код можно также добавить вручную в представлении **Код**. Например, приведенные ниже строки кода объявляют метод обработчика событий, который будет выполнен при вызове классом `Form` события `Load`.

   ```
   private void Form1_Load(object sender, System.EventArgs e)
   {
       // Add your form load event handling code here.
   }
   ```

   Строка кода, требуемая для подписки на событие, также создается автоматически в методе `InitializeComponent` в файле Form1.Designer.cs проекта. Она имеет следующий вид:

   ```
   this.Load += new System.EventHandler(this.Form1_Load);
   ```

Подписка на события программными средствами

1. Определите метод обработчика событий, сигнатура которого соответствует сигнатуре делегата для события. Например, если событие основано на типе делегата EventHandler, то следующий код представляет заглушку метода:

   ```
   void HandleCustomEvent(object sender, CustomEventArgs a)
   {
      // Do something useful here.
   }
   ```

2. Чтобы присоединить обработчик событий к событию, используйте оператор присваивания сложения (`+=`). В приведенном ниже примере предположим, что объект с именем `publisher` имеет событие с именем `RaiseCustomEvent`. Обратите внимание на то, что классу подписчика требуется ссылка на класс издателя, чтобы подписаться на его события.

   ```
   publisher.RaiseCustomEvent += HandleCustomEvent;
   ```

Чтобы указать обработчик событий, можно также воспользоваться лямбда-выражением:

```
public Form1()
{
    InitializeComponent();
    this.Click += (s,e) =>
        {
            MessageBox.Show(((MouseEventArgs)e).Location.ToString());
        };
}
```

Подписка на события с помощью анонимной функции

Если вам не нужно будет позже отменять подписку на событие, можно использовать оператор присваивания сложения (`+=`) для привязки анонимной функции как обработчика событий. В следующем примере предположим, что объект с именем `publisher` имеет событие с именем `RaiseCustomEvent` и что класс `CustomEventArgs` также был определен и содержит некие относящиеся к событию сведения. Обратите внимание на то, что классу подписчика требуется ссылка на `publisher`, чтобы подписаться на его события.

```
publisher.RaiseCustomEvent += (object o, CustomEventArgs e) =>
{
  string s = o.ToString() + " " + e.ToString();
  Console.WriteLine(s);
};
```

Отменить подписку на событие не так просто, если для подписки использовалась анонимная функция. Чтобы отменить подписку в этом случае, в коде, где была выполнена подписка на событие, сохраните анонимную функцию в переменной делегата, а затем добавьте делегат к событию. Мы рекомендуем не использовать анонимные функции для подписки на события, если в будущем вам нужно будет отменить подписку. Дополнительные сведения об анонимных функциях см. в статье Лямбда-выражения.

Отмена подписки

Чтобы предотвратить вызов обработчика событий при инициировании события, подписку на событие необходимо отменить. Во избежание утечки ресурсов отменять подписку на события следует до удаления объекта подписчика. До тех пор пока подписка на событие не отменена, делегат многоадресной рассылки, лежащий в основе события в публикующем объекте, будет ссылаться на делегат, инкапсулирующий обработчик событий подписчика. Если ссылка присутствует в публикующем объекте, объект подписчика не будет удален при сборке мусора.

Отмена подписки на событие

- Чтобы отменить подписку на событие, воспользуйтесь оператором присваивания вычитания (`-=`).

  ```
  publisher.RaiseCustomEvent -= HandleCustomEvent;
  ```

  Если подписка на событие отменена для всех подписчиков, экземпляр события в классе издателя получает значение `null`.

См. также

- События
- event
- Практическое руководство. Публикация событий, соответствующих рекомендациям .NET

- Операторы - и -=
- Операторы + и +=

Публикация событий, соответствующих рекомендациям .NET (руководство по программированию на C#)

18.12.2021 • 3 minutes to read

Следующая процедура демонстрирует добавление событий, которые соответствуют стандартному шаблону .NET для классов и структур. Все события в библиотеке классов .NET основаны на делегате EventHandler, который определен следующим образом:

```
public delegate void EventHandler(object sender, EventArgs e);
```

> **NOTE**
> В .NET Framework 2.0 представлена универсальная версия этого делегата, EventHandler<TEventArgs>. В следующих примерах демонстрируется использование обеих версий.

Хотя события в определяемых классах могут быть основаны на любом допустимом типе делегата, даже на делегатах, возвращающих значение, обычно рекомендуется основывать события на шаблоне .NET с помощью EventHandler, как показано в следующем примере.

Имя `EventHandler` может привести к путанице, так как на самом деле этот делегат не обрабатывает событие. Делегат EventHandler и универсальный делегат EventHandler<TEventArgs> являются типами делегатов. Метод или лямбда-выражение, сигнатура которого соответствует определению делегата, является *обработчиком событий* и будет вызываться при возникновении события.

Публикация событий, основанных на шаблоне EventHandler

1. (Пропустите этот шаг и перейдите к шагу 3a, если не нужно отправлять пользовательские данные с определенным событием.) Объявите класс для пользовательских данных в области, видимой для классов Publisher и Subscriber. Затем добавьте необходимые члены для хранения пользовательских данных о событиях. В этом примере возвращается простая строка.

   ```
   public class CustomEventArgs : EventArgs
   {
       public CustomEventArgs(string message)
       {
           Message = message;
       }

       public string Message { get; set; }
   }
   ```

2. (Пропустите этот шаг, если используется универсальная версия EventHandler<TEventArgs>.) Объявите делегат в своем классе публикации. Присвойте ему имя, которое заканчивается на `EventHandler`. Второй параметр указывает настраиваемый тип `EventArgs`.

   ```
   public delegate void CustomEventHandler(object sender, CustomEventArgs args);
   ```

3. Объявите событие в своем классе публикации, выполнив одно из следующих действий.

   a. Если у вас нет пользовательского класса EventArgs, тип события будет неуниверсальным делегатом EventHandler. Нет необходимости объявлять делегат, так как он уже объявлен в пространстве имен System, которое включается при создании проекта C#. Добавьте следующий код в класс Publisher.

   ```
   public event EventHandler RaiseCustomEvent;
   ```

   b. Если вы используете неуниверсальную версию EventHandler и имеется пользовательский класс, производный от EventArgs, объявите событие внутри класса публикации и используйте делегат из шага 2 в качестве типа.

   ```
   public event CustomEventHandler RaiseCustomEvent;
   ```

   c. Если используется универсальная версия, пользовательский делегат не требуется. Вместо этого в классе публикации укажите тип события как `EventHandler<CustomEventArgs>`, подставив имя своего класса в угловые скобки.

   ```
   public event EventHandler<CustomEventArgs> RaiseCustomEvent;
   ```

Пример

В следующем примере показана вышеописанная процедура с использованием пользовательского класса EventArgs и EventHandler<TEventArgs> в качестве типа событий.

```
using System;

namespace DotNetEvents
{
    // Define a class to hold custom event info
    public class CustomEventArgs : EventArgs
    {
        public CustomEventArgs(string message)
        {
            Message = message;
        }

        public string Message { get; set; }
    }

    // Class that publishes an event
    class Publisher
    {
        // Declare the event using EventHandler<T>
        public event EventHandler<CustomEventArgs> RaiseCustomEvent;

        public void DoSomething()
        {
            // Write some code that does something useful here
            // then raise the event. You can also raise an event
            // before you execute a block of code.
            OnRaiseCustomEvent(new CustomEventArgs("Event triggered"));
        }

        // Wrap event invocations inside a protected virtual method
        // to allow derived classes to override the event invocation behavior
        protected virtual void OnRaiseCustomEvent(CustomEventArgs e)
        {
```

```
        {
            // Make a temporary copy of the event to avoid possibility of
            // a race condition if the last subscriber unsubscribes
            // immediately after the null check and before the event is raised.
            EventHandler<CustomEventArgs> raiseEvent = RaiseCustomEvent;

            // Event will be null if there are no subscribers
            if (raiseEvent != null)
            {
                // Format the string to send inside the CustomEventArgs parameter
                e.Message += $" at {DateTime.Now}";

                // Call to raise the event.
                raiseEvent(this, e);
            }
        }
    }

    //Class that subscribes to an event
    class Subscriber
    {
        private readonly string _id;

        public Subscriber(string id, Publisher pub)
        {
            _id = id;

            // Subscribe to the event
            pub.RaiseCustomEvent += HandleCustomEvent;
        }

        // Define what actions to take when the event is raised.
        void HandleCustomEvent(object sender, CustomEventArgs e)
        {
            Console.WriteLine($"{_id} received this message: {e.Message}");
        }
    }

    class Program
    {
        static void Main()
        {
            var pub = new Publisher();
            var sub1 = new Subscriber("sub1", pub);
            var sub2 = new Subscriber("sub2", pub);

            // Call the method that raises the event.
            pub.DoSomething();

            // Keep the console window open
            Console.WriteLine("Press any key to continue...");
            Console.ReadLine();
        }
    }
}
```

См. также

- Delegate
- Руководство по программированию на C#
- События
- Делегаты

Руководство по программированию на C#. Порождение событий базового класса в производных классах

18.12.2021 • 3 minutes to read

В следующем простом примере показан стандартный способ объявления событий в базовом классе, позволяющий вызывать их из производных классов. Этот шаблон активно применяется в классах Windows Forms в библиотеке классов .NET.

При создании класса, который можно использовать в качестве базового для других классов, следует учитывать тот факт, что события представляют собой особый тип делегатов, который может вызываться только в том классе, который их объявил. Производные классы не могут создавать события, объявленные в базовом классе, напрямую. Несмотря на то, что в некоторых обстоятельствах вам может потребоваться событие, вызываемое только базовым классом, в большинстве случаев рекомендуется разрешать производному классу вызывать события базового класса. Для этого следует создать в базовом классе защищенный метод, предоставляющий оболочку для события. Вызывая или перезаписывая вызывающий метод, производные классы могут вызывать событие косвенно.

> **NOTE**
> Не объявляйте виртуальные события в базовом классе и не переопределяйте их в производном классе. Компилятор c# не обрабатывает их корректно, поэтому сложно сказать, будет ли подписчик производного события на самом деле подписываться на событие базового класса.

Пример

```
namespace BaseClassEvents
{
    // Special EventArgs class to hold info about Shapes.
    public class ShapeEventArgs : EventArgs
    {
        public ShapeEventArgs(double area)
        {
            NewArea = area;
        }

        public double NewArea { get; }
    }

    // Base class event publisher
    public abstract class Shape
    {
        protected double _area;

        public double Area
        {
            get => _area;
            set => _area = value;
        }

        // The event. Note that by using the generic EventHandler<T> event type
        // we do not need to declare a separate delegate type.
```

```
        public event EventHandler<ShapeEventArgs> ShapeChanged;

        public abstract void Draw();

        //The event-invoking method that derived classes can override.
        protected virtual void OnShapeChanged(ShapeEventArgs e)
        {
            // Safely raise the event for all subscribers
            ShapeChanged?.Invoke(this, e);
        }
    }

    public class Circle : Shape
    {
        private double _radius;

        public Circle(double radius)
        {
            _radius = radius;
            _area = 3.14 * _radius * _radius;
        }

        public void Update(double d)
        {
            _radius = d;
            _area = 3.14 * _radius * _radius;
            OnShapeChanged(new ShapeEventArgs(_area));
        }

        protected override void OnShapeChanged(ShapeEventArgs e)
        {
            // Do any circle-specific processing here.

            // Call the base class event invocation method.
            base.OnShapeChanged(e);
        }

        public override void Draw()
        {
            Console.WriteLine("Drawing a circle");
        }
    }

    public class Rectangle : Shape
    {
        private double _length;
        private double _width;

        public Rectangle(double length, double width)
        {
            _length = length;
            _width = width;
            _area = _length * _width;
        }

        public void Update(double length, double width)
        {
            _length = length;
            _width = width;
            _area = _length * _width;
            OnShapeChanged(new ShapeEventArgs(_area));
        }

        protected override void OnShapeChanged(ShapeEventArgs e)
        {
            // Do any rectangle-specific processing here.

            // Call the base class event invocation method.
            base.OnShapeChanged(e);
```

```
        }

        public override void Draw()
        {
            Console.WriteLine("Drawing a rectangle");
        }
    }

    // Represents the surface on which the shapes are drawn
    // Subscribes to shape events so that it knows
    // when to redraw a shape.
    public class ShapeContainer
    {
        private readonly List<Shape> _list;

        public ShapeContainer()
        {
            _list = new List<Shape>();
        }

        public void AddShape(Shape shape)
        {
            _list.Add(shape);

            // Subscribe to the base class event.
            shape.ShapeChanged += HandleShapeChanged;
        }

        // ...Other methods to draw, resize, etc.

        private void HandleShapeChanged(object sender, ShapeEventArgs e)
        {
            if (sender is Shape shape)
            {
                // Diagnostic message for demonstration purposes.
                Console.WriteLine($"Received event. Shape area is now {e.NewArea}");

                // Redraw the shape here.
                shape.Draw();
            }
        }
    }

    class Test
    {
        static void Main()
        {
            //Create the event publishers and subscriber
            var circle = new Circle(54);
            var rectangle = new Rectangle(12, 9);
            var container = new ShapeContainer();

            // Add the shapes to the container.
            container.AddShape(circle);
            container.AddShape(rectangle);

            // Cause some events to be raised.
            circle.Update(57);
            rectangle.Update(7, 7);

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to continue...");
            Console.ReadKey();
        }
    }
}
/* Output:
        Received event. Shape area is now 10201.86
        Drawing a circle
```

```
        Received event. Shape area is now 49
        Drawing a rectangle
*/
```

См. также

- Руководство по программированию на C#
- События
- Делегаты
- Модификаторы доступа
- Создание обработчиков событий в Windows Forms

Руководство по программированию на C#. Реализация событий интерфейса

18.12.2021 • 2 minutes to read

Интерфейс может объявлять события. Следующий пример демонстрирует реализацию событий интерфейса в классе. По сути правила остаются таким же, как при реализации любого метода или свойства интерфейса.

Реализация событий интерфейса в классе

Объявите событие в классе и вызовите его, когда потребуется.

```
namespace ImplementInterfaceEvents
{
    public interface IDrawingObject
    {
        event EventHandler ShapeChanged;
    }
    public class MyEventArgs : EventArgs
    {
        // class members
    }
    public class Shape : IDrawingObject
    {
        public event EventHandler ShapeChanged;
        void ChangeShape()
        {
            // Do something here before the event…

            OnShapeChanged(new MyEventArgs(/*arguments*/));

            // or do something here after the event.
        }
        protected virtual void OnShapeChanged(MyEventArgs e)
        {
            ShapeChanged?.Invoke(this, e);
        }
    }

}
```

Пример

Следующий пример демонстрирует обработку менее распространенной ситуации, в которой класс наследует от двух или более интерфейсов, каждый из которых имеет событие с одним и тем же именем. Вам придется предоставить явную реализацию интерфейса по крайней мере для одного из этих событий. Когда вы создаете для события явную реализацию интерфейса, нужно добавить еще и методы доступа `add` и `remove`. Обычно они предоставляются компилятором, но в этой ситуации компилятор не сможет предоставить их.

Предоставляя собственные методы доступа, вы можете указать одно или разные события вашего класса, которые соответствуют этим двум событиям. Например, если события должны инициироваться в разное время в соответствии со спецификациями интерфейса, можно связать каждое из них с отдельной реализацией в классе. В следующем примере подписчики определяют, какое из событий `OnDraw` они

получат, указывая ссылку на фигуру `IShape` или `IDrawingObject`.

```
namespace WrapTwoInterfaceEvents
{
    using System;

    public interface IDrawingObject
    {
        // Raise this event before drawing
        // the object.
        event EventHandler OnDraw;
    }
    public interface IShape
    {
        // Raise this event after drawing
        // the shape.
        event EventHandler OnDraw;
    }

    // Base class event publisher inherits two
    // interfaces, each with an OnDraw event
    public class Shape : IDrawingObject, IShape
    {
        // Create an event for each interface event
        event EventHandler PreDrawEvent;
        event EventHandler PostDrawEvent;

        object objectLock = new Object();

        // Explicit interface implementation required.
        // Associate IDrawingObject's event with
        // PreDrawEvent
        #region IDrawingObjectOnDraw
        event EventHandler IDrawingObject.OnDraw
        {
            add
            {
                lock (objectLock)
                {
                    PreDrawEvent += value;
                }
            }
            remove
            {
                lock (objectLock)
                {
                    PreDrawEvent -= value;
                }
            }
        }
        #endregion
        // Explicit interface implementation required.
        // Associate IShape's event with
        // PostDrawEvent
        event EventHandler IShape.OnDraw
        {
            add
            {
                lock (objectLock)
                {
                    PostDrawEvent += value;
                }
            }
            remove
            {
                lock (objectLock)
                {
                    PostDrawEvent -= value;
```

```
                }
            }
        }

        // For the sake of simplicity this one method
        // implements both interfaces.
        public void Draw()
        {
            // Raise IDrawingObject's event before the object is drawn.
            PreDrawEvent?.Invoke(this, EventArgs.Empty);

            Console.WriteLine("Drawing a shape.");

            // Raise IShape's event after the object is drawn.
            PostDrawEvent?.Invoke(this, EventArgs.Empty);
        }
    }
    public class Subscriber1
    {
        // References the shape object as an IDrawingObject
        public Subscriber1(Shape shape)
        {
            IDrawingObject d = (IDrawingObject)shape;
            d.OnDraw += d_OnDraw;
        }

        void d_OnDraw(object sender, EventArgs e)
        {
            Console.WriteLine("Sub1 receives the IDrawingObject event.");
        }
    }
    // References the shape object as an IShape
    public class Subscriber2
    {
        public Subscriber2(Shape shape)
        {
            IShape d = (IShape)shape;
            d.OnDraw += d_OnDraw;
        }

        void d_OnDraw(object sender, EventArgs e)
        {
            Console.WriteLine("Sub2 receives the IShape event.");
        }
    }

    public class Program
    {
        static void Main(string[] args)
        {
            Shape shape = new Shape();
            Subscriber1 sub = new Subscriber1(shape);
            Subscriber2 sub2 = new Subscriber2(shape);
            shape.Draw();

            // Keep the console window open in debug mode.
            System.Console.WriteLine("Press any key to exit.");
            System.Console.ReadKey();
        }
    }
}
/* Output:
    Sub1 receives the IDrawingObject event.
    Drawing a shape.
    Sub2 receives the IShape event.
*/
```

См. также

- Руководство по программированию на C#
- События
- Делегаты
- Явная реализация интерфейса
- Практическое руководство. Создание событий базового класса в производных классах

Руководство по программированию на C#. Реализация пользовательских методов доступа к событиям

18.12.2021 • 2 minutes to read

Событие представляет собой особый вид многоадресного делегата, который можно вызвать только из класса, где он объявлен. В клиентском коде создается подписка на событие. Для этого предоставляется ссылка на метод, который должен вызываться при возникновении этого события. Эти методы добавляются в список вызова делегата с помощью методов доступа к событиям, которые схожи с методами доступа к свойствам и отличаются только именами (`add` и `remove`). В большинстве случаев реализовывать настраиваемые методы доступа к событиям не требуется. Если в коде отсутствуют настраиваемые методы доступа к событиям, компилятор добавляет их автоматически. Тем не менее в некоторых случаях требуется реализовать настраиваемое поведение. Один из таких сценариев показан в руководстве по реализации событий интерфейса.

Пример

В следующем примере показано, как реализовать настраиваемые методы доступа add и remove для события. При необходимости вы можете заменять любой код в методах доступа, однако мы рекомендуем заблокировать событие, прежде чем добавлять или удалять новый метод обработчика событий.

```
event EventHandler IDrawingObject.OnDraw
{
    add
    {
        lock (objectLock)
        {
            PreDrawEvent += value;
        }
    }
    remove
    {
        lock (objectLock)
        {
            PreDrawEvent -= value;
        }
    }
}
```

См. также

- События
- event

Руководство по программированию на C#. Параметры универсального типа

18.12.2021 • 2 minutes to read

В определении универсального типа или метода параметр типа является заполнителем для определенного типа, который клиент задает при создании экземпляра переменной универсального типа. Универсальный класс, например `GenericList<T>` из раздела Общие сведения об универсальных шаблонах, нельзя использовать в исходном виде, поскольку он представляет собой не фактически тип, а, скорее, заготовку типа. Чтобы использовать класс `GenericList<T>`, в коде клиента необходимо объявить сконструированный тип и создать его экземпляр, указав аргумент типа в угловых скобках. Аргумент типа для этого конкретного класса может иметь любой тип, распознаваемый компилятором. Можно создать любое число экземпляров сконструированного типа, каждый из которых будет использовать разные аргументы типа, как показано ниже:

```
GenericList<float> list1 = new GenericList<float>();
GenericList<ExampleClass> list2 = new GenericList<ExampleClass>();
GenericList<ExampleStruct> list3 = new GenericList<ExampleStruct>();
```

В каждом из этих экземпляров `GenericList<T>` каждое вхождение `T` в классе во время выполнения будет заменяться аргументом типа. Посредством такой замены в этом случае создается три отдельных типобезопасных эффективных объекта, использующих одно определение класса. Дополнительные сведения о том, как в среде CLR реализуется эта замена, см. в разделе Универсальные типы во время выполнения.

Правила именования параметров типа

- **Присваивайте** параметрам универсального типа описательные имена, кроме случаев, когда достаточно одной буквы и описательное имя не имеет практической ценности.

  ```
  public interface ISessionChannel<TSession> { /*...*/ }
  public delegate TOutput Converter<TInput, TOutput>(TInput from);
  public class List<T> { /*...*/ }
  ```

- **Используйте** имя параметра типа T для типов, содержащих только однобуквенный параметр типа.

  ```
  public int IComparer<T>() { return 0; }
  public delegate bool Predicate<T>(T item);
  public struct Nullable<T> where T : struct { /*...*/ }
  ```

- **Используйте** префикс "T" для описательных имен параметров типа.

  ```
  public interface ISessionChannel<TSession>
  {
      TSession Session { get; }
  }
  ```

- **Указывайте** ограничения, связанные с параметром типа, в его имени. Например, параметр с

ограничением `ISession` может называться `TSession`.

Правило анализа кода CA1715 можно использовать, чтобы убедиться, что параметры типа, именуются соответствующим образом.

См. также

- System.Collections.Generic
- Руководство по программированию на C#
- Универсальные шаблоны
- Различия между шаблонами языка C++ и универсальными шаблонами языка C#

Ограничения параметров типа (руководство по программированию на C#)

18.12.2021 • 10 minutes to read

Ограничения сообщают компилятору о характеристиках, которые должен иметь аргумент типа. Без ограничений аргумент типа может быть любым типом. Компилятор может только предполагать членов System.Object, который является главным базовым классом для всех типов .NET. Дополнительные сведения см. в статье Зачем использовать ограничения. Если клиентский код использует тип, который не удовлетворяет ограничению, компилятор выдает ошибку. Ограничения задаются с помощью контекстного ключевого слова `where`. В следующей таблице описываются различные типы ограничений:

ОГРАНИЧЕНИЕ	ОПИСАНИЕ
`where T : struct`	Аргумент типа должен быть типом значения, не допускающим значения NULL. См. дополнительные сведения о типах значений, допускающих значение NULL. Поскольку все типы значений имеют доступный конструктор без параметров, ограничение `struct` подразумевает наличие ограничения `new()` и не может применяться в сочетании с ограничением `new()`. Ограничение `struct` нельзя использовать вместе с ограничением `unmanaged`.
`where T : class`	Аргумент типа должен быть ссылочным типом. Это ограничение также применяется к любому типу класса, интерфейса, делегата или массива. В контексте, допускающем значения NULL, в C# 8.0 или более поздней версии `T` должен быть ссылочным типом, не допускающим значения NULL.
`where T : class?`	Аргумент типа должен быть ссылочным типом, допускающим значения NULL или не допускающим значения NULL. Это ограничение также применяется к любому типу класса, интерфейса, делегата или массива.
`where T : notnull`	Аргумент типа должен быть типом, не допускающим значения NULL. Аргумент может быть ссылочным типом, не допускающим значения NULL в C# 8.0 или более поздней версии или типом значения, не допускающим значения NULL.
`where T : default`	Это ограничение устраняет неоднозначность, если необходимо указать неограниченный параметр типа, переопределяя метод или указывая явную реализацию интерфейса. Ограничение `default` подразумевает базовый метод без ограничения `class` или `struct`. Дополнительные сведения см. в разделе характеристик ограничения `default`.

ОГРАНИЧЕНИЕ	ОПИСАНИЕ
`where T : unmanaged`	Аргумент типа должен быть неуправляемым типом, не допускающим значения NULL. Ограничение `unmanaged` подразумевает ограничение `struct` и не может использоваться совместно с ограничением `struct` или `new()`.
`where T : new()`	Аргумент типа должен иметь общий конструктор без параметров. При одновременном использовании нескольких ограничений последним должно указываться ограничение `new()`. Ограничение `new()` не может использоваться с ограничениями `struct` и `unmanaged`.
`where T :` *<base class name>*	Аргумент типа должен иметь базовый класс или производный от него класс. В контексте, допускающем значение NULL, в C# 8.0 и более поздней версии `T` должен быть ссылочным типом, не допускающим значения NULL, производным от указанного базового класса.
`where T :` *<base class name>?*	Аргумент типа должен иметь базовый класс или производный от него класс. В контексте, допускающем значения NULL, в C# 8.0 и более поздней версии `T` может быть типом, допускающим значения NULL или не допускающим значения NULL, производным от указанного базового класса.
`where T :` *<interface name>*	Аргумент типа должен являться заданным интерфейсом или реализовывать его. Можно указать несколько ограничений интерфейса. Заданный в ограничении интерфейс также может быть универсальным. В контексте, допускающем значения NULL, в C# 8.0 и более поздней версии `T` должен быть типом, не допускающим значения NULL, который реализует указанный интерфейс.
`where T :` *<interface name>?*	Аргумент типа должен являться заданным интерфейсом или реализовывать его. Можно указать несколько ограничений интерфейса. Заданный в ограничении интерфейс также может быть универсальным. В контексте, допускающем значения NULL, в C# 8.0 `T` может быть ссылочным типом, допускающим значения NULL, ссылочным типом, не допускающим значения NULL, или типом значения. `T` не может быть типом значения, допускающим значения NULL.
`where T : U`	Аргумент типа, указанный для `T`, должен быть аргументом, указанным для `U`, или производным от него. Если в контексте, допускающем значения NULL, `U` является ссылочным типом, допускающим значения NULL, `T` должен быть ссылочным типом, не допускающим значения NULL. Если `U` является ссылочным типом, допускающим значения NULL, `T` может допускать или не допускать значения NULL.

Зачем использовать ограничения

Ограничения определяют возможности и ожидания параметра типа. Объявление этих ограничений означает, что можно использовать операции и вызовы методов ограничивающего типа. Если универсальный класс или метод использует любые другие операции с универсальными элементами, помимо простого присвоения или вызова методов, которые не поддерживает System.Object, вам необходимо будет применить ограничения к параметру типа. Например, на основе ограничения базового класса компилятор определяет, что в качестве аргументов типа будут использоваться только объекты указанного типа или производных от него. Имея такую гарантию, компилятор может вызывать методы указанного типа в универсальном классе. В следующем примере кода показаны функциональные возможности, которые можно добавить в класс `GenericList<T>` (см. раздел Введение в универсальные шаблоны), применив ограничение базового класса.

```
public class Employee
{
    public Employee(string name, int id) => (Name, ID) = (name, id);
    public string Name { get; set; }
    public int ID { get; set; }
}

public class GenericList<T> where T : Employee
{
    private class Node
    {
        public Node(T t) => (Next, Data) = (null, t);

        public Node Next { get; set; }
        public T Data { get; set; }
    }

    private Node head;

    public void AddHead(T t)
    {
        Node n = new Node(t) { Next = head };
        head = n;
    }

    public IEnumerator<T> GetEnumerator()
    {
        Node current = head;

        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }

    public T FindFirstOccurrence(string s)
    {
        Node current = head;
        T t = null;

        while (current != null)
        {
            //The constraint enables access to the Name property.
            if (current.Data.Name == s)
            {
                t = current.Data;
                break;
            }
            else
            {
                current = current.Next;
            }
        }
        return t;
    }
}
```

Это ограничение позволяет универсальному классу использовать свойство `Employee.Name`. Ограничение указывает, что все элементы типа `T` гарантированно являются либо объектом `Employee`, либо объектом, который наследует от `Employee`.

К одному параметру типа можно применять несколько ограничений, которые сами по себе могут быть универсальными типами, как показано ниже:

```
class EmployeeList<T> where T : Employee, IEmployee, System.IComparable<T>, new()
{
    // ...
}
```

При применении ограничения `where T : class` не рекомендуется использовать операторы `==` и `!=` для параметра типа, поскольку в этом случае будет проверяться только удостоверение ссылки, а не равенство значений. Такое поведение будет наблюдаться даже в том случае, если эти операторы будут перегружены в типе, используемом в качестве аргумента. Эта особенность показана в следующем коде, который будет возвращать значение false даже в том случае, если класс String перегружает оператор `==`.

```
public static void OpEqualsTest<T>(T s, T t) where T : class
{
    System.Console.WriteLine(s == t);
}

private static void TestStringEquality()
{
    string s1 = "target";
    System.Text.StringBuilder sb = new System.Text.StringBuilder("target");
    string s2 = sb.ToString();
    OpEqualsTest<string>(s1, s2);
}
```

Компилятору известно только то, что `T` является ссылочным типом во время компиляции, и он должен использовать операторы по умолчанию, которые действительны для всех ссылочных типов. Чтобы проверить равенство значений, рекомендуется применить ограничение `where T : IEquatable<T>` или `where T : IComparable<T>` и реализовать этот интерфейс в любом классе, который будет использоваться для создания универсального класса.

Ограничение нескольких параметров

Ограничения можно применить к нескольким параметрам. Кроме того, к одному параметру можно применять несколько ограничений, как показано в следующем примере:

```
class Base { }
class Test<T, U>
    where U : struct
    where T : Base, new()
{ }
```

Несвязанные параметры типа

Не имеющие ограничений параметры типа (например, T в общем классе `SampleClass<T>{}`) называются несвязанными. В отношении несвязанных параметров типа применяются следующие правила:

- Не допускается использование операторов `!=` и `==`, поскольку нет гарантии, что они будут поддерживаться конкретным аргументом типа.
- Их можно преобразовать в `System.Object` или явно преобразовать в любой тип интерфейса.
- Можно сравнить их со значением null. При сравнении несвязанного параметра с `null` для аргументов типа, являющихся типами значений, всегда возвращается значение false.

Параметры типа в качестве ограничений

Использование параметров универсального типа в качестве ограничений применимо, когда функция-

член со своим параметром типа должна ограничивать этот параметр параметром содержащего типа, как показано в следующем примере:

```
public class List<T>
{
    public void Add<U>(List<U> items) where U : T {/*...*/}
}
```

В предыдущем примере `T` является ограничением типа в контексте метода `Add` и несвязанным параметром типа в контексте класса `List`.

Параметры типа также можно использовать в определениях универсальных классов. Параметр типа необходимо объявлять в угловых скобках вместе с любыми другими параметрами типа:

```
//Type parameter V is used as a type constraint.
public class SampleClass<T, U, V> where T : V { }
```

Применение параметров типа в качестве ограничений для универсальных классов ограничено, поскольку в отношении таких параметров типа компилятор может предполагать только то, что они являются производными от `System.Object`. Параметры типа в качестве ограничений следует использовать в универсальных классах в тех случаях, когда необходимо обеспечить отношение наследования между двумя параметрами типа.

Ограничение `notnull`

Начиная с C# 8.0, можно использовать ограничение `notnull`, чтобы указать, что аргумент типа должен быть типом значения, не допускающим значения NULL, или ссылочным типом, не допускающим значения NULL. В отличие от большинства других ограничений, если аргумент типа нарушает ограничение `notnull`, компилятор генерирует предупреждение вместо ошибки.

Ограничение `notnull` действует только при использовании в контексте, допускающем значения NULL. При добавлении ограничения `notnull` в очевидный контекст, допускающий значения NULL, компилятор не создает никаких предупреждений или ошибок в случае нарушений ограничения.

Ограничение `class`

Начиная с C# 8.0, ограничение `class` в контексте, допускающем значения NULL, указывает, что аргумент типа должен быть ссылочным типом, не допускающим значения NULL. Если в контексте, допускающем значения NULL, аргумент типа является ссылочным типом, допускающим значения NULL, компилятор выдаст предупреждение.

Ограничение `default`

Добавление ссылочных типов, допускающих значения NULL, усложняет использование `T?` для универсального типа или метода. В коде C# версий, предшествующих 8, `T?` можно использовать, только если к `T` применено ограничение `struct`. В этом контексте `T?` ссылается на тип Nullable<T> для `T`. Начиная с C# 8 `T?` можно использовать с ограничением `struct` или `class`, но должно присутствовать только одно из них. Если используется ограничение `class`, `T?` ссылается на ссылочный тип, допускающий значения NULL, для `T`. Начиная с C# 9 `T?` можно использовать, если нет ограничений. В этом случае `T?` интерпретируется так же, как для типов значений и ссылочных типов в коде C# 8. Но если `T` — экземпляр Nullable<T>, `T?` соответствует `T`. Другими словами, это ограничение не станет `T??`.

Так как `T?` теперь можно использовать без ограничения `class` или `struct`, в переопределениях или явных реализациях интерфейса могут возникать неоднозначности. В обоих случаях переопределение не включает в себя ограничения, но наследует их от базового класса. Если базовый класс не применяет ограничение `class` или `struct`, производные классы должны каким-либо образом указывать переопределение, применяемое к базовому методу без ограничения. Это происходит, когда производный метод применяет ограничение `default`. Ограничение `default` *не* уточняет ни ограничение `class`, ни `struct`.

Неуправляемое ограничение

Начиная с C# 7.3, можно использовать ограничение `unmanaged`, чтобы указать, что параметр типа должен быть неуправляемым типом, не допускающим значения NULL. Ограничение `unmanaged` позволяет создавать многократно используемые подпрограммы для работы с типами, которые могут обрабатываться как блоки памяти, как показано в следующем примере:

```
unsafe public static byte[] ToByteArray<T>(this T argument) where T : unmanaged
{
    var size = sizeof(T);
    var result = new Byte[size];
    Byte* p = (byte*)&argument;
    for (var i = 0; i < size; i++)
        result[i] = *p++;
    return result;
}
```

В примере выше метод необходимо компилировать в контексте `unsafe`, так как он использует оператор `sizeof` для типа, не известного как встроенный тип. Без ограничения `unmanaged` оператор `sizeof` недоступен.

Ограничение `unmanaged` подразумевает ограничение `struct` и не может использоваться совместно с ним. Поскольку ограничение `struct` подразумевает ограничение `new()`, ограничение `unmanaged` также не может использоваться с ограничением `new()`.

Ограничения делегата

Кроме того, начиная с C# 7.3 можно использовать System.Delegate или System.MulticastDelegate как ограничение базового класса. В среде CLR это ограничение всегда было разрешено, но в языке C# оно было запрещено. Ограничение `System.Delegate` позволяет написать код, который работает с делегатами типобезопасным образом. Следующий код определяет метод расширения, который объединяет два делегата при условии, что они одного типа:

```
public static TDelegate TypeSafeCombine<TDelegate>(this TDelegate source, TDelegate target)
    where TDelegate : System.Delegate
    => Delegate.Combine(source, target) as TDelegate;
```

Приведенный выше метод можно использовать для объединения делегатов, которые относятся к одному типу:

```
Action first = () => Console.WriteLine("this");
Action second = () => Console.WriteLine("that");

var combined = first.TypeSafeCombine(second);
combined();

Func<bool> test = () => true;
// Combine signature ensures combined delegates must
// have the same type.
//var badCombined = first.TypeSafeCombine(test);
```

Если раскомментировать последнюю строку, она не будет компилироваться. `first` и `test` являются типами делегатов, но это разные типы делегатов.

Ограничения перечисления

Начиная с C# 7.3 можно указать тип System.Enum в качестве ограничения базового класса. В среде CLR это ограничение всегда было разрешено, но в языке C# оно было запрещено. Универсальные шаблоны с `System.Enum` предоставляют типобезопасное программирование для кэширования результатов использования статических методов в `System.Enum`. В следующем примере выполняется поиск всех допустимых значений для типа перечисления и создается словарь, который сопоставляет эти значения с их строковым представлением.

```
public static Dictionary<int, string> EnumNamedValues<T>() where T : System.Enum
{
    var result = new Dictionary<int, string>();
    var values = Enum.GetValues(typeof(T));

    foreach (int item in values)
        result.Add(item, Enum.GetName(typeof(T), item));
    return result;
}
```

`Enum.GetValues` и `Enum.GetName` используют отражение, которое влияет на производительность. Вы можете не повторять вызовы, требующие отражения, а вызвать `EnumNamedValues` для создания коллекции, которая кэшируется и используется повторно.

Вы можете использовать его, как показано в следующем примере, для создания перечисления и словаря имен и значений:

```
enum Rainbow
{
    Red,
    Orange,
    Yellow,
    Green,
    Blue,
    Indigo,
    Violet
}
```

```
var map = EnumNamedValues<Rainbow>();

foreach (var pair in map)
    Console.WriteLine($"{pair.Key}:\t{pair.Value}");
```

См. также

- System.Collections.Generic
- Руководство по программированию на C#
- Введение в универсальные шаблоны
- Универсальные классы
- Ограничение new

Универсальные классы (Руководство по программированию на C#)

18.12.2021 • 3 minutes to read

Универсальные классы инкапсулируют операции, которые не относятся к конкретному типу данных. Универсальные классы чаще всего используются для работы с коллекциями, такими как связанные списки, хэш-таблицы, стеки, очереди, деревья и т. д. Такие операции, как добавление и удаление элементов коллекции, по существу выполняются одинаково, независимо от типа хранимых данных.

В большинстве случаев для этого используются классы коллекций. Рекомендуется выбирать те из них, которые представлены в библиотеке классов платформы .NET. Дополнительные сведения об использовании этих классов см. в разделе Универсальные коллекции в .NET.

Как правило, при создании универсального класса сначала определяется конкретный класс, после чего его типы поочередно заменяются параметрами типов до тех пор, пока не будет достигнут необходимый баланс между степенью обобщения и удобством работы. При создании собственных универсальных классов необходимо учитывать следующие важные моменты:

- Типы, которые требуется обобщить с использованием параметров типа.

  Как правило, чем больше типов параметризовано, тем более гибким и универсальным становится ваш код. Тем не менее слишком высокая степень обобщения может отрицательно сказаться на понятности создаваемого вами кода для других разработчиков.

- Ограничения (если требуются), которые будут применяться к параметрам типа (см. раздел Ограничения параметров типа).

  Рекомендуется применять максимально возможный объем ограничений, при котором вы по-прежнему сможете работать с необходимыми типами. Например, если универсальный класс будет использоваться только для работы со ссылочными типами, примените ограничение класса. Это позволит исключить случайное использование класса с типами значений и позволит использовать оператор `as` в отношении `T`, а также проверять наличие значений null.

- Требуется ли разбивать универсальные функции между базовыми классами и подклассами.

  Поскольку универсальные классы могут выступать в качестве базовых классов, здесь необходимо учитывать те же принципы разработки, что и для классов, не являющихся универсальными. См. описание правил наследования от универсальных базовых классов далее в этом разделе.

- Требуется ли реализовывать один или несколько универсальных интерфейсов.

  Например, при разработке класса, который будет использоваться для создания элементов коллекции на основе универсальных шаблонов, может потребоваться реализовать интерфейс IComparable<T>, где `T` — это тип вашего класса.

Пример простого универсального класса можно найти в разделе Введение в универсальные шаблоны.

Правила в отношении параметров типа и ограничений влияют на поведение универсального класса, особенно в контексте наследования и доступа к элементам. Прежде чем продолжить, необходимо ознакомиться с некоторыми терминами и понятиями. Ссылка на универсальный класс `Node<T>,` в клиентском коде может задаваться с использованием аргумента типа или путем создания закрытого сконструированного типа (`Node<int>`). Кроме того, можно не задавать параметр типа (например, при указании универсального базового класса), чтобы определить открытый сконструированный тип (

`Node<T>`). Универсальные классы могут наследоваться от конкретных, а также закрытых или открытых сконструированных базовых классов:

```
class BaseNode { }
class BaseNodeGeneric<T> { }

// concrete type
class NodeConcrete<T> : BaseNode { }

//closed constructed type
class NodeClosed<T> : BaseNodeGeneric<int> { }

//open constructed type
class NodeOpen<T> : BaseNodeGeneric<T> { }
```

Классы, не являющиеся универсальными, то есть конкретные классы, могут наследоваться от закрытых сконструированных базовых классов. Наследование от аналогичных открытых классов или от параметров типа невозможно, поскольку во время выполнения клиентский код не может предоставить аргумент типа, необходимый для создания экземпляра базового класса.

```
//No error
class Node1 : BaseNodeGeneric<int> { }

//Generates an error
//class Node2 : BaseNodeGeneric<T> {}

//Generates an error
//class Node3 : T {}
```

Универсальные классы, наследуемые от открытых сконструированных типов, должны предоставлять аргументы типа для любых параметров типа базового класса, которые не используются совместно с наследующим классом. Это продемонстрировано в следующем коде:

```
class BaseNodeMultiple<T, U> { }

//No error
class Node4<T> : BaseNodeMultiple<T, int> { }

//No error
class Node5<T, U> : BaseNodeMultiple<T, U> { }

//Generates an error
//class Node6<T> : BaseNodeMultiple<T, U> {}
```

Универсальные классы, наследуемые от открытых сконструированных типов, должны задавать множество ограничений, которые явно или косвенно включают в себя все ограничения базового типа:

```
class NodeItem<T> where T : System.IComparable<T>, new() { }
class SpecialNodeItem<T> : NodeItem<T> where T : System.IComparable<T>, new() { }
```

Универсальные типы могут использовать несколько параметров типа и ограничений, как показано ниже:

```
class SuperKeyType<K, V, U>
    where U : System.IComparable<U>
    where V : new()
{ }
```

Открытые и закрытые сконструированные типы можно использовать в качестве параметров метода:

```
void Swap<T>(List<T> list1, List<T> list2)
{
    //code to swap items
}

void Swap(List<int> list1, List<int> list2)
{
    //code to swap items
}
```

Если универсальный класс реализует интерфейс, все экземпляры такого класса можно привести к этому интерфейсу.

Универсальные классы инвариантны. Другими словами, если входной параметр задает `List<BaseClass>`, при попытке предоставить `List<DerivedClass>` возникает ошибка времени компиляции.

См. также

- System.Collections.Generic
- Руководство по программированию на C#
- Универсальные шаблоны
- Сохранение состояния перечислителей
- Загадка по наследованию, часть 1

Универсальные интерфейсы. (Руководство по программированию на C#)

18.12.2021 • 4 minutes to read

Часто требуется определить интерфейсы для универсальных классов коллекций или для универсальных классов, представляющих элементы коллекции. Для универсальных классов предпочтительнее использовать универсальные интерфейсы, такие как IComparable<T>, вместо IComparable, поскольку это позволяет избежать выполнения операция упаковки-преобразования и распаковки-преобразования для типов значений. В библиотеке классов .NET в пространстве имен System.Collections.Generic определяется несколько универсальных интерфейсов для работы с классами коллекций.

Если интерфейс задан в качестве ограничения для параметра типа, можно использовать только типы, реализующие такой интерфейс. В следующем примере кода демонстрируется класс `SortedList<T>`, который является производным от класса `GenericList<T>`. Дополнительные сведения см. в разделе Введение в универсальные шаблоны. `SortedList<T>` добавляет ограничение `where T : IComparable<T>`. Это позволяет методу `BubbleSort` в `SortedList<T>` использовать универсальный метод CompareTo в отношении элементов списка. В этом примере элементы списка относятся к простому классу `Person`, который реализует `IComparable<Person>`.

```
//Type parameter T in angle brackets.
public class GenericList<T> : System.Collections.Generic.IEnumerable<T>
{
    protected Node head;
    protected Node current = null;

    // Nested class is also generic on T
    protected class Node
    {
        public Node next;
        private T data;  //T as private member datatype

        public Node(T t)  //T used in non-generic constructor
        {
            next = null;
            data = t;
        }

        public Node Next
        {
            get { return next; }
            set { next = value; }
        }

        public T Data  //T as return type of property
        {
            get { return data; }
            set { data = value; }
        }
    }

    public GenericList()  //constructor
    {
        head = null;
    }

    public void AddHead(T t)  //T as method parameter type
    {
```

```
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }

    // Implementation of the iterator
    public System.Collections.Generic.IEnumerator<T> GetEnumerator()
    {
        Node current = head;
        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }

    // IEnumerable<T> inherits from IEnumerable, therefore this class
    // must implement both the generic and non-generic versions of
    // GetEnumerator. In most cases, the non-generic method can
    // simply call the generic method.
    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}

public class SortedList<T> : GenericList<T> where T : System.IComparable<T>
{
    // A simple, unoptimized sort algorithm that
    // orders list elements from lowest to highest:

    public void BubbleSort()
    {
        if (null == head || null == head.Next)
        {
            return;
        }
        bool swapped;

        do
        {
            Node previous = null;
            Node current = head;
            swapped = false;

            while (current.next != null)
            {
                //  Because we need to call this method, the SortedList
                //  class is constrained on IComparable<T>
                if (current.Data.CompareTo(current.next.Data) > 0)
                {
                    Node tmp = current.next;
                    current.next = current.next.next;
                    tmp.next = current;

                    if (previous == null)
                    {
                        head = tmp;
                    }
                    else
                    {
                        previous.next = tmp;
                    }
                    previous = tmp;
                    swapped = true;
                }
                else
                {
                    previous = current;
```

```
                    current = current.next;
                }
            }
        } while (swapped);
    }
}

// A simple class that implements IComparable<T> using itself as the
// type argument. This is a common design pattern in objects that
// are stored in generic lists.
public class Person : System.IComparable<Person>
{
    string name;
    int age;

    public Person(string s, int i)
    {
        name = s;
        age = i;
    }

    // This will cause list elements to be sorted on age values.
    public int CompareTo(Person p)
    {
        return age - p.age;
    }

    public override string ToString()
    {
        return name + ":" + age;
    }

    // Must implement Equals.
    public bool Equals(Person p)
    {
        return (this.age == p.age);
    }
}

public class Program
{
    public static void Main()
    {
        //Declare and instantiate a new generic SortedList class.
        //Person is the type argument.
        SortedList<Person> list = new SortedList<Person>();

        //Create name and age values to initialize Person objects.
        string[] names = new string[]
        {
            "Franscoise",
            "Bill",
            "Li",
            "Sandra",
            "Gunnar",
            "Alok",
            "Hiroyuki",
            "Maria",
            "Alessandro",
            "Raul"
        };

        int[] ages = new int[] { 45, 19, 28, 23, 18, 9, 108, 72, 30, 35 };

        //Populate the list.
        for (int x = 0; x < 10; x++)
        {
            list.AddHead(new Person(names[x], ages[x]));
        }
```

```
        //Print out unsorted list.
        foreach (Person p in list)
        {
            System.Console.WriteLine(p.ToString());
        }
        System.Console.WriteLine("Done with unsorted list");

        //Sort the list.
        list.BubbleSort();

        //Print out sorted list.
        foreach (Person p in list)
        {
            System.Console.WriteLine(p.ToString());
        }
        System.Console.WriteLine("Done with sorted list");
    }
}
```

В качестве ограничений для одного типа можно задать несколько интерфейсов, как показано ниже:

```
class Stack<T> where T : System.IComparable<T>, IEnumerable<T>
{
}
```

Интерфейс может определять несколько параметров типа, как показано ниже:

```
interface IDictionary<K, V>
{
}
```

Правила наследования, действующие в отношении классов, также применяются к интерфейсам:

```
interface IMonth<T> { }

interface IJanuary     : IMonth<int> { }  //No error
interface IFebruary<T> : IMonth<int> { }  //No error
interface IMarch<T>    : IMonth<T> { }    //No error
//interface IApril<T>  : IMonth<T, U> {}  //Error
```

Универсальные интерфейсы могут наследоваться от неуниверсальных интерфейсов, если универсальный интерфейс является ковариантным (то есть использует в качестве возвращаемого значения только свой параметр типа). В библиотеке классов .NET IEnumerable<T> наследуется от IEnumerable, поскольку IEnumerable<T> использует `T` только в возвращаемом значении GetEnumerator и в методе считывания свойства Current.

Конкретные классы могут реализовывать закрытые сконструированные интерфейсы, как показано ниже:

```
interface IBaseInterface<T> { }

class SampleClass : IBaseInterface<string> { }
```

Универсальные классы могут реализовывать универсальные интерфейсы или закрытые сконструированные интерфейсы при условии, что в списке параметров класса заданы все аргументы, требуемые интерфейсом, как показано ниже:

```
interface IBaseInterface1<T> { }
interface IBaseInterface2<T, U> { }

class SampleClass1<T> : IBaseInterface1<T> { }          //No error
class SampleClass2<T> : IBaseInterface2<T, string> { }  //No error
```

Правила, управляющие перегрузкой методов, одинаковы для методов с универсальными классами, структурами или интерфейсами. Дополнительные сведения см. в разделе Универсальные методы.

См. также

- Руководство по программированию на C#
- Введение в универсальные шаблоны
- interface
- Универсальные шаблоны

Универсальные методы (Руководство по программированию на C#)

18.12.2021 • 2 minutes to read

Универсальным называется метод, объявленный с использованием параметров типа, как показано ниже:

```
static void Swap<T>(ref T lhs, ref T rhs)
{
    T temp;
    temp = lhs;
    lhs = rhs;
    rhs = temp;
}
```

В следующем примере кода показан один из способов вызова метода с использованием `int` в качестве аргумента типа:

```
public static void TestSwap()
{
    int a = 1;
    int b = 2;

    Swap<int>(ref a, ref b);
    System.Console.WriteLine(a + " " + b);
}
```

Если опустить аргумент типа, компилятор определит его. Следующий вызов `Swap` эквивалентен предыдущему:

```
Swap(ref a, ref b);
```

Для статических методов и методов экземпляра применяются одинаковые правила определения типа. Компилятор может определить параметры типа на основе переданных ему аргументов метода. Определение параметров типа невозможно только для ограничения или возвращаемого значения. Соответственно, механизм определения типа не работает с методами, не имеющими параметров. Определение типа происходит во время компиляции до того, как компилятор попытается разрешить сигнатуры перегруженных методов. Компилятор применяет логику определения типа ко всем универсальным методам с одинаковым именем. На этапе разрешения перегрузки компилятор включает только те универсальные методы, для которых был успешно определен тип.

В универсальном классе методы, не являющиеся универсальными, могут получать доступ к параметрам типа на уровне класса следующим образом:

```
class SampleClass<T>
{
    void Swap(ref T lhs, ref T rhs) { }
}
```

Если определить универсальный метод, принимающий те же параметры типа, что и содержащий его класс, возникнет предупреждение компилятора CS0693. Это связано с тем, что в области действия

метода предоставленный для внутреннего типа `T` аргумент скрывает аргумент, предоставленный для внешнего типа `T`. Для большей гибкости можно вызывать метод универсального класса с использованием аргументов типа, отличающихся от предоставленных при создании экземпляра класса. В этом случае следует предоставить другой идентификатор для параметра типа метода, как показано в `GenericList2<T>` в следующем примере.

```
class GenericList<T>
{
    // CS0693
    void SampleMethod<T>() { }
}

class GenericList2<T>
{
    //No warning
    void SampleMethod<U>() { }
}
```

С помощью ограничений можно реализовать специализированные операции в отношении параметров типа в методах. Эта версия `Swap<T>` теперь называется `SwapIfGreater<T>` и может использоваться только с типами, реализующими IComparable<T>.

```
void SwapIfGreater<T>(ref T lhs, ref T rhs) where T : System.IComparable<T>
{
    T temp;
    if (lhs.CompareTo(rhs) > 0)
    {
        temp = lhs;
        lhs = rhs;
        rhs = temp;
    }
}
```

Универсальные методы могут быть перегружены в нескольких параметрах типа. Например, все представленные ниже методы могут располагаться в одном классе:

```
void DoWork() { }
void DoWork<T>() { }
void DoWork<T, U>() { }
```

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#.

См. также

- System.Collections.Generic
- Руководство по программированию на C#
- Введение в универсальные шаблоны
- Методы

Универсальные методы и массивы (Руководство по программированию на C#)

18.12.2021 • 2 minutes to read

В C# 2.0 и более поздних версиях одномерные массивы с нулевой нижней границей автоматически реализуют IList<T>. Это позволяет создавать универсальные методы, которые могут использовать один и тот же код для выполнения итераций в массивах и других типах коллекций. Этот способ используется преимущественно для чтения данных в коллекциях. Интерфейс IList<T> нельзя использовать для добавления или удаления элементов массива. При попытке вызвать метод IList<T>, например RemoveAt, для массива в этом контексте возникнет исключение.

В следующем примере кода показано, как отдельный универсальный метод, принимающий входной параметр IList<T>, можно использовать для выполнения итераций в списке и массиве (в данном случае в массиве целых чисел).

```
class Program
{
    static void Main()
    {
        int[] arr = { 0, 1, 2, 3, 4 };
        List<int> list = new List<int>();

        for (int x = 5; x < 10; x++)
        {
            list.Add(x);
        }

        ProcessItems<int>(arr);
        ProcessItems<int>(list);
    }

    static void ProcessItems<T>(IList<T> coll)
    {
        // IsReadOnly returns True for the array and False for the List.
        System.Console.WriteLine
            ("IsReadOnly returns {0} for this collection.",
            coll.IsReadOnly);

        // The following statement causes a run-time exception for the
        // array, but not for the List.
        //coll.RemoveAt(4);

        foreach (T item in coll)
        {
            System.Console.Write(item.ToString() + " ");
        }
        System.Console.WriteLine();
    }
}
```

См. также

- System.Collections.Generic
- Руководство по программированию на C#
- Универсальные шаблоны

- Массивы
- Универсальные шаблоны

Универсальные делегаты. (Руководство по программированию на C#)

18.12.2021 • 2 minutes to read

Делегат может определять собственные параметры типа. В коде, который ссылается на универсальный делегат, можно указать аргумент типа для создания закрытого сконструированного типа. Это будет аналогично созданию экземпляра универсального класса или вызову универсального метода, как показано в следующем примере:

```
public delegate void Del<T>(T item);
public static void Notify(int i) { }

Del<int> m1 = new Del<int>(Notify);
```

В C# версии 2.0 представлена новая функция группового преобразования методов, которая применяется как к конкретным, так и к универсальным типам делегатов, и позволяет записать приведенную выше строку с использованием упрощенного синтаксиса:

```
Del<int> m2 = Notify;
```

Делегаты, определенные в универсальном классе, могут использовать параметры класса универсального типа таким же образом, как это делают методы класса.

```
class Stack<T>
{
    T[] items;
    int index;

    public delegate void StackDelegate(T[] items);
}
```

В коде, который ссылается на делегат, необходимо указать аргумент типа содержащего класса, как показано ниже:

```
private static void DoWork(float[] items) { }

public static void TestStack()
{
    Stack<float> s = new Stack<float>();
    Stack<float>.StackDelegate d = DoWork;
}
```

Универсальные делегаты особенно полезны при определении событий, основанных на типовых шаблонах разработки, поскольку аргумент отправителя может быть строго типизирован и больше не требует приведения к Object и из него.

```
delegate void StackEventHandler<T, U>(T sender, U eventArgs);

class Stack<T>
{
    public class StackEventArgs : System.EventArgs { }
    public event StackEventHandler<Stack<T>, StackEventArgs> stackEvent;

    protected virtual void OnStackChanged(StackEventArgs a)
    {
        stackEvent(this, a);
    }
}

class SampleClass
{
    public void HandleStackChange<T>(Stack<T> stack, Stack<T>.StackEventArgs args) { }
}

public static void Test()
{
    Stack<double> s = new Stack<double>();
    SampleClass o = new SampleClass();
    s.stackEvent += o.HandleStackChange;
}
```

См. также

- System.Collections.Generic
- Руководство по программированию на C#
- Введение в универсальные шаблоны
- Универсальные методы
- Универсальные классы
- Универсальные интерфейсы
- Делегаты
- Универсальные шаблоны

Различия между шаблонами языка C++ и универсальными шаблонами языка C# (Руководство по программированию в C#)

18.12.2021 • 2 minutes to read

Универсальные шаблоны C# и шаблоны C++ — это возможности языков программирования, обеспечивающие поддержку параметризированных типов. Однако между ними существует немало различий. На уровне синтаксиса универсальные шаблоны C# представляют собой более простой способ работы с параметризованными типами без сложностей, связанных с шаблонами C++. Кроме того, C# не пытается предоставить все функции, предоставляемые шаблонами C++. На уровне реализации основное различие заключается в том, что замена универсальных типов C# осуществляется во время выполнения, а значит для объектов, экземпляры которых при этом создаются, сохраняются данные об универсальных типах. Дополнительные сведения см. в разделе Универсальные типы во время выполнения.

Основные различия между универсальными шаблонами C# и шаблонами C++ состоят в следующем:

- Универсальные шаблоны C# не дают такую же гибкость, как шаблоны C++. Например, в универсальном классе C# нельзя вызывать арифметические операторы, но можно вызывать операторы, определенные пользователем.
- C# не позволяет использовать параметры шаблонов, не являющиеся типами, такие как `template C<int i> {}`.
- C# не поддерживает явную специализацию, т. е. индивидуальную реализацию шаблона для конкретного типа.
- C# не поддерживает частичную специализацию, т. е. индивидуальную реализацию для подмножества аргументов типа.
- C# не позволяет использовать параметр типа в качестве базового класса для универсального типа.
- C# не позволяет параметрам типов иметь типы по умолчанию.
- В C# параметр универсального типа сам по себе не может быть универсальным, однако в качестве универсальных типов можно использовать сконструированные типы. C++ не позволяет использовать параметры шаблонов.
- C++ позволяет использовать код, который может быть недопустимым для всех параметров типа в шаблоне, а затем проверяет, используется ли в качестве параметра типа определенный тип. C# требует, чтобы код в классе был написан так, чтобы он работал с любым типом, соответствующим ограничениям. Например, в C++ можно написать функцию, которая применяет арифметические операторы `+` и `-` к объектам параметра типа, в связи с чем при создании экземпляра шаблона, тип которого не поддерживает эти операторы, возникает ошибка. C# этого не позволяет; допускаются только те языковые конструкции, которые можно вывести из ограничений.

См. также

- Руководство по программированию на C#
- Введение в универсальные шаблоны
- Шаблоны

Универсальные типы во время выполнения (Руководство по программированию в C#)

18.12.2021 • 2 minutes to read

Универсальный тип или метод, компилируемый на языке MSIL, будет содержать метаданные, определяющие наличие в нем параметров типа. Способ использования MSIL для универсального типа зависит от того, является ли предоставленный параметр типа ссылочным типом или типом значения.

Если в качестве параметра при первом построении универсального типа использовался тип значения, во время выполнения создается специальный универсальный тип, для которого в соответствующих местах кода MSIL заменяются предоставленные параметры. Специальные универсальные типы создаются один раз для каждого уникального типа значения, используемого в качестве параметра.

Допустим, в коде программы объявляется стек, состоящий из целых чисел:

```
Stack<int> stack;
```

В этой точке среда выполнения создает специальную версию класса Stack<T>, параметр которого будет соответствующим образом заменяться целым числом. После этого каждый раз, когда в коде программы используется стек целых чисел, среда выполнения обращается к созданному специальному классу Stack<T>. В следующем примере создаются два экземпляра стека целых чисел, которые совместно используют один экземпляр кода `Stack<int>`:

```
Stack<int> stackOne = new Stack<int>();
Stack<int> stackTwo = new Stack<int>();
```

Допустим, в другой точке кода создается другой класс Stack<T>, в качестве параметра которого используется другой тип значения, например `long`, или определяемая пользователем структура. В результате среда выполнения создает другую версию универсального типа и заменяет `long` в соответствующих местах кода MSIL. Преобразования больше не требуются, поскольку каждый специальный универсальный класс содержит собственный тип значения.

Для ссылочных типов универсальные шаблоны применяются несколько иным образом. При первом создании универсального типа с любым ссылочным типом среда выполнения создает в коде MSIL специальный универсальный тип, параметры которого заменяются ссылками на объекты. После этого каждый раз при создании экземпляра такого сконструированного типа с использованием в качестве параметра ссылочного типа (независимо от того, какой это тип) среда выполнения использует ранее созданную специальную версию универсального типа. Это возможно, поскольку все ссылки имеют одинаковый размер.

Допустим, у вас есть два ссылочных типа (классы `Customer` и `Order`) и вы создаете стек типов `Customer`:

```
class Customer { }
class Order { }
```

```
Stack<Customer> customers;
```

В этой точке среда выполнения создает специальную версию класса Stack<T>, содержащую ссылки на

объекты, которые не хранят данные и будут заполнены позднее. Допустим, в следующей строке кода создается стек другого ссылочного типа с именем `Order`:

```
Stack<Order> orders = new Stack<Order>();
```

В отличие от типов значений, в этом случае не создается другая специальная версия класса Stack<T> для типа `Order`. Вместо этого создается экземпляр специальной версии класса Stack<T> и задается переменная `orders`, ссылающаяся на него. Предположим, далее встречается строка кода, в которой создается стек типа `Customer`:

```
customers = new Stack<Customer>();
```

Как и в случае предшествующего использования класса Stack<T>, созданного с помощью типа `Order`, создается другой специальный экземпляр класса Stack<T>. Содержащиеся в нем указатели будут ссылаться на область памяти с размером типа `Customer`. Поскольку число ссылок в разных программах может значительно различаться, реализация C# на основе универсальных шаблонов позволяет значительно сократить объем кода за счет того, что компилятор создает для универсальных классов ссылочных типов только один специальный класс.

Кроме того, при создании экземпляра универсального класса C# с использованием параметра типа значения или ссылочного типа копия может выполнять запросы к нему во время выполнения, и при этом могут быть установлены его фактический тип и параметр типа.

См. также раздел

- System.Collections.Generic
- Руководство по программированию на C#
- Введение в универсальные шаблоны
- Универсальные шаблоны

Универсальные типы и отражение (Руководство по программированию в C#)

18.12.2021 • 2 minutes to read

Поскольку среда CLR имеет доступ к данным универсальных типов во время выполнения, вы можете использовать отражение для получения сведений об универсальных типах точно так же, как и для неуниверсальных типов. Дополнительные сведения см. в разделе Универсальные типы во время выполнения.

В .NET Framework 2.0 в класс Type было добавлено несколько новых элементов для поддержки данных времени выполнения для универсальных типов. Дополнительные сведения о том, как использовать эти методы и свойства, см. в документации по этим классам. Пространство имен System.Reflection.Emit также содержит новые члены, которые поддерживают универсальные типы. См. практическое руководство по Определение универсального типа с порождаемым отражением

Список неизменяемых условий для терминов, используемых в отражении универсальных типов, см. в примечаниях к описанию свойства IsGenericType.

ИМЯ ЧЛЕНА SYSTEM.TYPE	**ОПИСАНИЕ**
IsGenericType	Возвращает значение true, если тип является универсальным.
GetGenericArguments	Возвращает массив объектов `Type`, которые представляют аргументы типа, предоставленные для создаваемого типа, или параметры определения универсального типа.
GetGenericTypeDefinition	Возвращает базовое определение универсального типа для текущего создаваемого типа.
GetGenericParameterConstraints	Возвращает массив объектов `Type`, которые представляют ограничения, накладываемые на параметр текущего универсального типа.
ContainsGenericParameters	Возвращает значение true, если тип или любые включающие его типы или методы содержат параметры типа, для которых конкретные типы не предоставлены.
GenericParameterAttributes	Получает сочетание флагов `GenericParameterAttributes`, описывающих особые ограничения текущего параметра универсального типа.
GenericParameterPosition	Для объекта `Type`, представляющего параметр типа, получает позицию параметра типа в списке параметров определения универсального типа или метода, который объявил параметр типа.

ИМЯ ЧЛЕНА SYSTEM.TYPE	ОПИСАНИЕ
IsGenericParameter	Получает значение, позволяющее определить, представляет ли текущий объект `Type` параметр типа для определения универсального типа или метода.
IsGenericTypeDefinition	Возвращает значение, определяющее, представляет ли текущий объект Type определение универсального типа, на основе которого можно конструировать другие универсальные типы. Возвращает значение true, если тип представляет определение универсального типа.
DeclaringMethod	Возвращает универсальный метод, который определяет параметр текущего универсального типа, или значение null, если параметр типа не был определен универсальным методом.
MakeGenericType	Замещает элементы массива типов для параметров определения текущего универсального типа и возвращает объект Type, представляющий сконструированный результирующий тип.

Кроме того, члены класса MethodInfo обеспечивают данные времени выполнения для универсальных методов. Список неизменяемых условий для терминов, используемых для отражения универсальных методов, см. в примечаниях к описанию свойства IsGenericMethod.

ИМЯ ЧЛЕНА SYSTEM.REFLECTION.MEMBERINFO	ОПИСАНИЕ
IsGenericMethod	Возвращает значение true, если метод является универсальным.
GetGenericArguments	Возвращает массив объектов Type, которые представляют аргументы создаваемого универсального метода, относящиеся к типу, или параметры типа определения универсального метода.
GetGenericMethodDefinition	Возвращает базовое определение универсального метода для текущего создаваемого метода.
ContainsGenericParameters	Возвращает значение true, если метод или любые включающие его типы содержат параметры типа, для которых конкретные типы не предоставлены.
IsGenericMethodDefinition	Возвращает значение true, если текущий метод MethodInfo представляет определение универсального метода.
MakeGenericMethod	Заменяет параметры типа элементами массива типов для определения текущего универсального метода и возвращает объект MethodInfo, представляющий итоговый сконструированный метод.

См. также

- Руководство по программированию на C#
- Универсальные шаблоны

- Отражение и универсальные типы
- Универсальные шаблоны

Универсальные шаблоны и атрибуты (Руководство по программированию в C#)

18.12.2021 • 2 minutes to read

Атрибуты можно применять к универсальным типам так же, как и к типам, не являющимся универсальными. Дополнительные сведения о применении атрибутов см. в разделе Атрибуты.

Настраиваемые атрибуты могут ссылаться только на открытые универсальные типы (универсальные типы, для которых не предоставлены аргументы типа) и закрытые сконструированные универсальные типы (типы, для всех параметров типа которых предоставлены аргументы).

Такой настраиваемый атрибут используется в следующих примерах:

```
class CustomAttribute : System.Attribute
{
    public System.Object info;
}
```

Атрибут может ссылаться на открытый универсальный тип:

```
public class GenericClass1<T> { }

[CustomAttribute(info = typeof(GenericClass1<>))]
class ClassA { }
```

Укажите несколько параметров типа, используя соответствующее количество запятых. В этом примере `GenericClass2` имеет два параметра типа:

```
public class GenericClass2<T, U> { }

[CustomAttribute(info = typeof(GenericClass2<,>))]
class ClassB { }
```

Атрибут может ссылаться на закрытый сконструированный универсальный тип:

```
public class GenericClass3<T, U, V> { }

[CustomAttribute(info = typeof(GenericClass3<int, double, string>))]
class ClassC { }
```

Если атрибут ссылается на параметр универсального типа, возникнет ошибка времени компиляции:

```
//[CustomAttribute(info = typeof(GenericClass3<int, T, string>))]  //Error
class ClassD<T> { }
```

Универсальный тип не может наследоваться от Attribute:

```
//public class CustomAtt<T> : System.Attribute {}  //Error
```

Для получения сведений об универсальном типе или параметре типа во время выполнения можно использовать методы из System.Reflection. Дополнительные сведения см. в разделе Универсальные типы и отражение

См. также

- Руководство по программированию на C#
- Универсальные шаблоны
- Атрибуты

Файловая система и реестр (руководство по программированию на C#)

18.12.2021 • 2 minutes to read

Перечисленные здесь статьи демонстрируют, как выполнять различные базовые операции с файлами, папками и реестром с помощью C# и .NET.

Содержание раздела

ЗАГОЛОВОК	ОПИСАНИЕ
Практическое руководство. Перебор дерева папок	Демонстрирует ручной перебор дерева каталогов.
Практическое руководство. Получение сведений о файлах, папках и дисках	Демонстрирует, как получить сведения о файлах, папках и дисках, например время создания и размер.
Практическое руководство. Создание файла или папки	Демонстрирует создание нового файла и новой папки.
Практическое руководство. Копирование, удаление и перемещение файлов и папок (руководство по программированию в C#)	Демонстрирует копирование, удаление и перемещение файлов и папок.
Практическое руководство. Предоставление диалогового окна "Ход выполнения" для операций с файлами	Объясняет, как отобразить стандартное диалоговое окно Windows для хода выполнения определенных файловых операций.
Практическое руководство. Запись в текстовый файл	Демонстрирует запись в текстовый файл.
Практическое руководство. Чтение из текстового файла	Демонстрирует чтение из текстового файла.
Практическое руководство. Построчное чтение текстового файла	Демонстрирует извлечение текста из файла по одной строке.
Практическое руководство. Создание раздела в реестре	Демонстрирует запись раздела в системный реестр.

Связанные разделы

- Файловый и потоковый ввод-вывод
- Практическое руководство. Копирование, удаление и перемещение файлов и папок (руководство по программированию в C#)
- Руководство по программированию на C#
- System.IO

Руководство по программированию на C#. Выполнение итерации по дереву папок

18.12.2021 • 6 minutes to read

Под фразой "обход дерева каталогов" подразумевается доступ к каждому файлу в каждом вложенном подкаталоге в заданной корневой папке на любую глубину. Необязательно открывать каждый файл. Можно просто извлечь имя файла или подкаталога в виде значения `string` или получить дополнительную информацию в форме объекта System.IO.FileInfo или System.IO.DirectoryInfo.

> **NOTE**
> В Windows термины "каталог" и "папка" являются взаимозаменяемыми. В большинстве документации и в текстах пользовательского интерфейса используется термин "папка", но в библиотеке классов .NET используется термин "каталог".

В простейшем случае, когда точно известно, что имеются права доступа ко всем каталогам в указанном корне, можно использовать флаг `System.IO.SearchOption.AllDirectories`. Этот флаг возвращает все вложенные подкаталоги, соответствующие заданному шаблону. В приведенном ниже примере показано, как использовать этот флаг.

```
root.GetDirectories("*.*", System.IO.SearchOption.AllDirectories);
```

Слабая сторона такого подхода заключается в том, что если какой-либо из подкаталогов в указанном корне вызовет DirectoryNotFoundException или UnauthorizedAccessException, то весь метод не будет выполнен и каталоги не будут возвращены. Это же относится и к использованию метода GetFiles. Если необходимо обработать эти исключения в определенных вложенных папках, необходимо вручную пройти по дереву каталога, как показано в приведенных ниже примерах.

При проходе вручную по дереву каталога можно обработать сначала файлы (*обход в прямом порядке*) или подкаталоги (*обход в обратном порядке*). При выполнении обхода в прямом порядке вы проходите файлы непосредственно в самой папке, а затем проходите все дерево в текущей папке. Обход в обратном порядке — это другой способ обхода всего дерева ниже перед получением файлов текущей папки. В приведенных ниже примерах выполняется обход в прямом порядке, но его можно легко изменить на обход в обратном порядке.

Еще одним параметром является использование рекурсии или обхода на основе стека. В приведенных ниже примерах показаны оба подхода.

Если необходимо выполнить различные операции с файлами и папками, то можно смоделировать эти примеры путем рефакторинга операции на отдельные функции, которые можно вызвать с помощью одного делегата.

NOTE

Файловые системы NTFS могут содержать *точки повторного анализа* в форме *точек соединения*, *символических ссылок* и *жестких связей*. Методы платформы .NET, такие как GetFiles и GetDirectories, не возвращают подкаталоги ниже точки повторного анализа. Такое поведение предотвращает риск входа в бесконечный цикл, когда две точки повторного анализа ссылаются друг на друга. Как правило, следует быть предельно осторожными при работе с точками повторного анализа, чтобы избежать случайного изменения или удаления файлов. Если требуется строгий контроль над точками повторного анализа, используйте вызов неуправляемого кода или машинный код для прямого вызова подходящих методов файловой системы Win32.

Примеры

В приведенном ниже примере показан проход по дереву каталога с помощью рекурсии. Рекурсивный подход является элегантным, но он потенциально может вызвать исключение переполнения стека, если дерево каталога большое и имеет большой уровень вложения.

Обрабатываемые исключения и действия, выполняемые с каждым файлом или папкой, предоставляются только в качестве примеров. Этот код следует изменить в соответствии с конкретными требованиями. Дополнительные сведения см. в комментариях в коде.

```
public class RecursiveFileSearch
{
    static System.Collections.Specialized.StringCollection log = new
System.Collections.Specialized.StringCollection();

    static void Main()
    {
        // Start with drives if you have to search the entire computer.
        string[] drives = System.Environment.GetLogicalDrives();

        foreach (string dr in drives)
        {
            System.IO.DriveInfo di = new System.IO.DriveInfo(dr);

            // Here we skip the drive if it is not ready to be read. This
            // is not necessarily the appropriate action in all scenarios.
            if (!di.IsReady)
            {
                Console.WriteLine("The drive {0} could not be read", di.Name);
                continue;
            }
            System.IO.DirectoryInfo rootDir = di.RootDirectory;
            WalkDirectoryTree(rootDir);
        }

        // Write out all the files that could not be processed.
        Console.WriteLine("Files with restricted access:");
        foreach (string s in log)
        {
            Console.WriteLine(s);
        }
        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key");
        Console.ReadKey();
    }

    static void WalkDirectoryTree(System.IO.DirectoryInfo root)
    {
        System.IO.FileInfo[] files = null;
        System.IO.DirectoryInfo[] subDirs = null;

        // First, process all the files directly under this folder
```

```
        try
        {
            files = root.GetFiles("*.*");
        }
        // This is thrown if even one of the files requires permissions greater
        // than the application provides.
        catch (UnauthorizedAccessException e)
        {
            // This code just writes out the message and continues to recurse.
            // You may decide to do something different here. For example, you
            // can try to elevate your privileges and access the file again.
            log.Add(e.Message);
        }

        catch (System.IO.DirectoryNotFoundException e)
        {
            Console.WriteLine(e.Message);
        }

        if (files != null)
        {
            foreach (System.IO.FileInfo fi in files)
            {
                // In this example, we only access the existing FileInfo object. If we
                // want to open, delete or modify the file, then
                // a try-catch block is required here to handle the case
                // where the file has been deleted since the call to TraverseTree().
                Console.WriteLine(fi.FullName);
            }

            // Now find all the subdirectories under this directory.
            subDirs = root.GetDirectories();

            foreach (System.IO.DirectoryInfo dirInfo in subDirs)
            {
                // Resursive call for each subdirectory.
                WalkDirectoryTree(dirInfo);
            }
        }
    }
}
```

В приведенном ниже примере показан обход файлов и папок в дереве папок без использования рекурсии. Этот метод использует универсальный тип коллекции Stack<T>, который представляет стек типа "последним пришел — первым вышел" (LIFO).

Обрабатываемые исключения и действия, выполняемые с каждым файлом или папкой, предоставляются только в качестве примеров. Этот код следует изменить в соответствии с конкретными требованиями. Дополнительные сведения см. в комментариях в коде.

```
public class StackBasedIteration
{
    static void Main(string[] args)
    {
        // Specify the starting folder on the command line, or in
        // Visual Studio in the Project > Properties > Debug pane.
        TraverseTree(args[0]);

        Console.WriteLine("Press any key");
        Console.ReadKey();
    }

    public static void TraverseTree(string root)
    {
        // Data structure to hold names of subfolders to be
        // examined for files.
```

```
        Stack<string> dirs = new Stack<string>(20);

        if (!System.IO.Directory.Exists(root))
        {
            throw new ArgumentException();
        }
        dirs.Push(root);

        while (dirs.Count > 0)
        {
            string currentDir = dirs.Pop();
            string[] subDirs;
            try
            {
                subDirs = System.IO.Directory.GetDirectories(currentDir);
            }
            // An UnauthorizedAccessException exception will be thrown if we do not have
            // discovery permission on a folder or file. It may or may not be acceptable
            // to ignore the exception and continue enumerating the remaining files and
            // folders. It is also possible (but unlikely) that a DirectoryNotFound exception
            // will be raised. This will happen if currentDir has been deleted by
            // another application or thread after our call to Directory.Exists. The
            // choice of which exceptions to catch depends entirely on the specific task
            // you are intending to perform and also on how much you know with certainty
            // about the systems on which this code will run.
            catch (UnauthorizedAccessException e)
            {
                Console.WriteLine(e.Message);
                continue;
            }
            catch (System.IO.DirectoryNotFoundException e)
            {
                Console.WriteLine(e.Message);
                continue;
            }

            string[] files = null;
            try
            {
                files = System.IO.Directory.GetFiles(currentDir);
            }

            catch (UnauthorizedAccessException e)
            {

                Console.WriteLine(e.Message);
                continue;
            }

            catch (System.IO.DirectoryNotFoundException e)
            {
                Console.WriteLine(e.Message);
                continue;
            }
            // Perform the required action on each file here.
            // Modify this block to perform your required task.
            foreach (string file in files)
            {
                try
                {
                    // Perform whatever action is required in your scenario.
                    System.IO.FileInfo fi = new System.IO.FileInfo(file);
                    Console.WriteLine("{0}: {1}, {2}", fi.Name, fi.Length, fi.CreationTime);
                }
                catch (System.IO.FileNotFoundException e)
                {
                    // If file was deleted by a separate application
                    //  or thread since the call to TraverseTree()
                    // then just continue.
```

```
                    Console.WriteLine(e.Message);
                    continue;
                }
            }

            // Push the subdirectories onto the stack for traversal.
            // This could also be done before handing the files.
            foreach (string str in subDirs)
                dirs.Push(str);
        }
    }
}
```

Обычно проверка каждой папки на наличие у приложения разрешений на ее открытие занимает слишком много времени. Поэтому в примере кода эта часть операции просто включена в блок `try/catch`. Блок `catch` можно изменить так, чтобы при отказе в доступе к папке предпринималась попытка повысить права, а затем повторная попытка получить доступ. Как правило, следует перехватывать только те исключения, которые можно обработать, не оставляя приложение в неопределенном состоянии.

Если необходимо сохранить содержимое дерева каталога либо в памяти, либо на диске, наилучшим вариантом будет сохранение только свойства FullName (типа `string`) для каждого файла. Затем можно использовать эту строку для создания объекта FileInfo или DirectoryInfo по мере необходимости или открыть любой файл, для которого требуется дополнительная обработка.

Отказоустойчивость

Надежный код обхода файлов должен учитывать сложности файловой системы. Дополнительные сведения о файловой системе Windows см. в статье Обзор NTFS.

См. также

- System.IO
- LINQ и каталоги файлов
- Файловая система и реестр (руководство по программированию на C#)

Руководство по программированию на C#. Получение сведений о файлах, папках и дисках

18.12.2021 • 2 minutes to read

На платформе .NET доступ к сведениям о файловой системе можно получить, используя следующие классы.

- System.IO.FileInfo
- System.IO.DirectoryInfo
- System.IO.DriveInfo
- System.IO.Directory
- System.IO.File

Классы FileInfo и DirectoryInfo представляют файл или каталог и содержат свойства, представляющие многие атрибуты файла, поддерживаемые файловой системой NTFS. Они также содержат методы для открытия, закрытия, перемещения и удаления файлов и папок. Экземпляры этих классов можно создать, передав в конструктор строку, представляющую имя файла, папки или диска.

```
System.IO.DriveInfo di = new System.IO.DriveInfo(@"C:\");
```

Имена файлов, папок или дисков можно также получить с помощью вызова DirectoryInfo.GetDirectories, DirectoryInfo.GetFiles и DriveInfo.RootDirectory.

Классы System.IO.Directory и System.IO.File предоставляют статические методы для получения сведений о каталогах и файлах.

Пример

В следующем примере показаны различные способы доступа к сведениям о файлах и папках.

```
class FileSysInfo
{
    static void Main()
    {
        // You can also use System.Environment.GetLogicalDrives to
        // obtain names of all logical drives on the computer.
        System.IO.DriveInfo di = new System.IO.DriveInfo(@"C:\");
        Console.WriteLine(di.TotalFreeSpace);
        Console.WriteLine(di.VolumeLabel);

        // Get the root directory and print out some information about it.
        System.IO.DirectoryInfo dirInfo = di.RootDirectory;
        Console.WriteLine(dirInfo.Attributes.ToString());

        // Get the files in the directory and print out some information about them.
        System.IO.FileInfo[] fileNames = dirInfo.GetFiles("*.*");

        foreach (System.IO.FileInfo fi in fileNames)
        {
            Console.WriteLine("{0}: {1}: {2}", fi.Name, fi.LastAccessTime, fi.Length);
        }
```

```
        // Get the subdirectories directly that is under the root.
        // See "How to: Iterate Through a Directory Tree" for an example of how to
        // iterate through an entire tree.
        System.IO.DirectoryInfo[] dirInfos = dirInfo.GetDirectories("*.*");

        foreach (System.IO.DirectoryInfo d in dirInfos)
        {
            Console.WriteLine(d.Name);
        }

        // The Directory and File classes provide several static methods
        // for accessing files and directories.

        // Get the current application directory.
        string currentDirName = System.IO.Directory.GetCurrentDirectory();
        Console.WriteLine(currentDirName);

        // Get an array of file names as strings rather than FileInfo objects.
        // Use this method when storage space is an issue, and when you might
        // hold on to the file name reference for a while before you try to access
        // the file.
        string[] files = System.IO.Directory.GetFiles(currentDirName, "*.txt");

        foreach (string s in files)
        {
            // Create the FileInfo object only when needed to ensure
            // the information is as current as possible.
            System.IO.FileInfo fi = null;
            try
            {
                 fi = new System.IO.FileInfo(s);
            }
            catch (System.IO.FileNotFoundException e)
            {
                // To inform the user and continue is
                // sufficient for this demonstration.
                // Your application may require different behavior.
                Console.WriteLine(e.Message);
                continue;
            }
            Console.WriteLine("{0} : {1}",fi.Name, fi.Directory);
        }

        // Change the directory. In this case, first check to see
        // whether it already exists, and create it if it does not.
        // If this is not appropriate for your application, you can
        // handle the System.IO.IOException that will be raised if the
        // directory cannot be found.
        if (!System.IO.Directory.Exists(@"C:\Users\Public\TestFolder\"))
        {
            System.IO.Directory.CreateDirectory(@"C:\Users\Public\TestFolder\");
        }

        System.IO.Directory.SetCurrentDirectory(@"C:\Users\Public\TestFolder\");

        currentDirName = System.IO.Directory.GetCurrentDirectory();
        Console.WriteLine(currentDirName);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

Отказоустойчивость

При обработке заданных пользователем строк, определяющих пути, необходимо также обрабатывать исключения для следующих условий:

- Неверное имя файла. Например, оно содержит недопустимые символы или состоит из одних пробелов.
- Имя файла имеет значение NULL.
- Имя файла больше максимальной длины, определенной системой.
- Имя файла содержит двоеточие (:).

Если у приложения недостаточно прав для чтения указанного файла, метод `Exists` возвратит значение `false` независимо от существования указанного пути. Исключений этот метод не вызывает.

См. также

- System.IO
- Руководство по программированию на C#
- Файловая система и реестр (руководство по программированию на C#)

Руководство по программированию на C#. Создание файла или папки

18.12.2021 • 3 minutes to read

Вы можете программно создать на компьютере папку, вложенную папку и файл во вложенной папке, а затем записать данные в этот файл.

Пример

```
public class CreateFileOrFolder
{
    static void Main()
    {
        // Specify a name for your top-level folder.
        string folderName = @"c:\Top-Level Folder";

        // To create a string that specifies the path to a subfolder under your
        // top-level folder, add a name for the subfolder to folderName.
        string pathString = System.IO.Path.Combine(folderName, "SubFolder");

        // You can write out the path name directly instead of using the Combine
        // method. Combine just makes the process easier.
        string pathString2 = @"c:\Top-Level Folder\SubFolder2";

        // You can extend the depth of your path if you want to.
        //pathString = System.IO.Path.Combine(pathString, "SubSubFolder");

        // Create the subfolder. You can verify in File Explorer that you have this
        // structure in the C: drive.
        //    Local Disk (C:)
        //        Top-Level Folder
        //            SubFolder
        System.IO.Directory.CreateDirectory(pathString);

        // Create a file name for the file you want to create.
        string fileName = System.IO.Path.GetRandomFileName();

        // This example uses a random string for the name, but you also can specify
        // a particular name.
        //string fileName = "MyNewFile.txt";

        // Use Combine again to add the file name to the path.
        pathString = System.IO.Path.Combine(pathString, fileName);

        // Verify the path that you have constructed.
        Console.WriteLine("Path to my file: {0}\n", pathString);

        // Check that the file doesn't already exist. If it doesn't exist, create
        // the file and write integers 0 - 99 to it.
        // DANGER: System.IO.File.Create will overwrite the file if it already exists.
        // This could happen even with random file names, although it is unlikely.
        if (!System.IO.File.Exists(pathString))
        {
            using (System.IO.FileStream fs = System.IO.File.Create(pathString))
            {
                for (byte i = 0; i < 100; i++)
                {
                    fs.WriteByte(i);
                }
            }
```

```
            }
        }
        else
        {
            Console.WriteLine("File \"{0}\" already exists.", fileName);
            return;
        }

        // Read and display the data from your file.
        try
        {
            byte[] readBuffer = System.IO.File.ReadAllBytes(pathString);
            foreach (byte b in readBuffer)
            {
                Console.Write(b + " ");
            }
            Console.WriteLine();
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine(e.Message);
        }

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
    // Sample output:

    // Path to my file: c:\Top-Level Folder\SubFolder\ttxvauxe.vv0

    //0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
    //30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56
    // 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 8
    //3 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
}
```

Если папка уже существует, CreateDirectory не выполняет никаких действий и исключение не возникает. Но File.Create заменяет существующий файл новым. Для того чтобы этого избежать, в примере используется оператор `if` - `else`.

Изменив пример указанным ниже образом, вы можете задать различные результаты в зависимости от того, существует ли файл с определенным именем. Если файл не существует, он создается. Если файл существует, код добавляет в него данные.

- Укажите конкретное имя файла.

```
// Comment out the following line.
//string fileName = System.IO.Path.GetRandomFileName();

// Replace that line with the following assignment.
string fileName = "MyNewFile.txt";
```

- В следующем коде замените оператор `if` - `else` на `using`.

```
using (System.IO.FileStream fs = new System.IO.FileStream(pathString, FileMode.Append))
{
    for (byte i = 0; i < 100; i++)
    {
        fs.WriteByte(i);
    }
}
```

Выполните код в примере несколько раз, чтобы убедиться, что каждый раз данные добавляются в файл.

Другие значения `FileMode`, которые можно использовать для проверки, см. в разделе FileMode.

При следующих условиях возможно возникновение исключения:

- Имя папки недопустимо, Например, оно содержит недопустимые символы или состоит из одних пробелов (класс ArgumentException). Используйте класс Path, чтобы создавать допустимые пути.
- Родительская папка создаваемой папки доступна только для чтения (класс IOException).
- Имя папки — `null` (класс ArgumentNullException).
- Имя папки слишком длинное (класс PathTooLongException).
- Имя папки состоит из одного двоеточия ":" (класс PathTooLongException).

Безопасность .NET

Экземпляр класса SecurityException может быть порожден как исключение в ситуации частичного доверия.

Если у вас нет разрешения на создание папки, код в приведенном примере породит как исключение экземпляр класса UnauthorizedAccessException.

См. также

- System.IO
- Руководство по программированию на C#
- Файловая система и реестр (руководство по программированию на C#)

Руководство по программированию на C#. Копирование, удаление, перемещение файлов и папок

18.12.2021 • 3 minutes to read

В следующих примерах показано, как синхронно копировать, перемещать и удалять файлы и папки с помощью классов System.IO.File, System.IO.Directory, System.IO.FileInfo, и System.IO.DirectoryInfo из пространства имен System.IO. В этих примерах не используется индикатор хода выполнения или какой-либо иной пользовательский интерфейс. Если нужно использовать стандартное диалоговое окна хода выполнения, см. практическое руководство Отображение диалогового окна хода выполнения для операций с файлами.

Используйте System.IO.FileSystemWatcher для предоставления событий, которые позволяют вычислять ход выполнения при работе с несколькими файлами. Другим подходом является использование вызова неуправляемого кода для вызова в Windows Shell методов, относящихся к обработке файлов. Сведения о способах асинхронного выполнения таких операций над файлами см. в разделе Асинхронный файловый ввод-вывод.

Примеры

В следующем примере показано, как копировать файлы и каталоги.

```
// Simple synchronous file copy operations with no user interface.
// To run this sample, first create the following directories and files:
// C:\Users\Public\TestFolder
// C:\Users\Public\TestFolder\test.txt
// C:\Users\Public\TestFolder\SubDir\test.txt
public class SimpleFileCopy
{
    static void Main()
    {
        string fileName = "test.txt";
        string sourcePath = @"C:\Users\Public\TestFolder";
        string targetPath =  @"C:\Users\Public\TestFolder\SubDir";

        // Use Path class to manipulate file and directory paths.
        string sourceFile = System.IO.Path.Combine(sourcePath, fileName);
        string destFile = System.IO.Path.Combine(targetPath, fileName);

        // To copy a folder's contents to a new location:
        // Create a new target folder.
        // If the directory already exists, this method does not create a new directory.
        System.IO.Directory.CreateDirectory(targetPath);

        // To copy a file to another location and
        // overwrite the destination file if it already exists.
        System.IO.File.Copy(sourceFile, destFile, true);

        // To copy all the files in one directory to another directory.
        // Get the files in the source folder. (To recursively iterate through
        // all subfolders under the current directory, see
        // "How to: Iterate Through a Directory Tree.")
        // Note: Check for target path was performed previously
        //       in this code example.
        if (System.IO.Directory.Exists(sourcePath))
        {
            string[] files = System.IO.Directory.GetFiles(sourcePath);

            // Copy the files and overwrite destination files if they already exist.
            foreach (string s in files)
            {
                // Use static Path methods to extract only the file name from the path.
                fileName = System.IO.Path.GetFileName(s);
                destFile = System.IO.Path.Combine(targetPath, fileName);
                System.IO.File.Copy(s, destFile, true);
            }
        }
        else
        {
            Console.WriteLine("Source path does not exist!");
        }

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

В следующем примере показано, как перемещать файлы и каталоги.

```
// Simple synchronous file move operations with no user interface.
public class SimpleFileMove
{
    static void Main()
    {
        string sourceFile = @"C:\Users\Public\public\test.txt";
        string destinationFile = @"C:\Users\Public\private\test.txt";

        // To move a file or folder to a new location:
        System.IO.File.Move(sourceFile, destinationFile);

        // To move an entire directory. To programmatically modify or combine
        // path strings, use the System.IO.Path class.
        System.IO.Directory.Move(@"C:\Users\Public\public\test\", @"C:\Users\Public\private");
    }
}
```

В следующем примере показано, как удалять файлы и каталоги.

```
// Simple synchronous file deletion operations with no user interface.
// To run this sample, create the following files on your drive:
// C:\Users\Public\DeleteTest\test1.txt
// C:\Users\Public\DeleteTest\test2.txt
// C:\Users\Public\DeleteTest\SubDir\test2.txt

public class SimpleFileDelete
{
    static void Main()
    {
        // Delete a file by using File class static method...
        if(System.IO.File.Exists(@"C:\Users\Public\DeleteTest\test.txt"))
        {
            // Use a try block to catch IOExceptions, to
            // handle the case of the file already being
            // opened by another process.
            try
            {
                System.IO.File.Delete(@"C:\Users\Public\DeleteTest\test.txt");
            }
            catch (System.IO.IOException e)
            {
                Console.WriteLine(e.Message);
                return;
            }
        }

        // ...or by using FileInfo instance method.
        System.IO.FileInfo fi = new System.IO.FileInfo(@"C:\Users\Public\DeleteTest\test2.txt");
        try
        {
            fi.Delete();
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine(e.Message);
        }

        // Delete a directory. Must be writable or empty.
        try
        {
            System.IO.Directory.Delete(@"C:\Users\Public\DeleteTest");
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine(e.Message);
        }
```

```
        // Delete a directory and all subdirectories with Directory static method...
        if(System.IO.Directory.Exists(@"C:\Users\Public\DeleteTest"))
        {
            try
            {
                System.IO.Directory.Delete(@"C:\Users\Public\DeleteTest", true);
            }

            catch (System.IO.IOException e)
            {
                Console.WriteLine(e.Message);
            }
        }

        // ...or with DirectoryInfo instance method.
        System.IO.DirectoryInfo di = new System.IO.DirectoryInfo(@"C:\Users\Public\public");
        // Delete this dir and all subdirs.
        try
        {
            di.Delete(true);
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine(e.Message);
        }
    }
}
```

См. также

- System.IO
- Руководство по программированию на C#
- Файловая система и реестр (руководство по программированию на C#)
- Практическое руководство. Предоставление диалогового окна "Ход выполнения" для операций с файлами
- Файловый и потоковый ввод-вывод
- Распространенные задачи ввода-вывода

Руководство по программированию на C#. Отображение диалогового окна хода выполнения для операций с файлами

18.12.2021 • 2 minutes to read

Вы можете предоставить стандартное диалоговое окно, в котором будет отображаться ход выполнения операций с файлами в Windows, при использовании метода CopyFile(String, String, UIOption) из пространства имен Microsoft.VisualBasic.

> **NOTE**
> Отображаемые на компьютере имена или расположения некоторых элементов пользовательского интерфейса Visual Studio могут отличаться от указанных в следующих инструкциях. Это зависит от имеющегося выпуска Visual Studio и используемых параметров. Дополнительные сведения см. в разделе Персонализация среды IDE.

Добавление ссылки в Visual Studio

1. В строке меню выберите **Проект**, **Добавить ссылку**.

   Откроется диалоговое окно **Диспетчер ссылок**.

2. В области **Сборки** выберите **Платформа**, если этот вариант еще не выбран.

3. В списке имен установите флажок **Microsoft.VisualBasic**, а затем нажмите кнопку **ОК**, чтобы закрыть диалоговое окно.

Пример

Следующий код копирует каталог, указанный `sourcePath`, в каталог, указанный `destinationPath`. Этот код также предоставляет стандартное диалоговое окно, в котором отображается примерное время, оставшееся до завершения операции.

```
// The following using directive requires a project reference to Microsoft.VisualBasic.
using Microsoft.VisualBasic.FileIO;

class FileProgress
{
    static void Main()
    {
        // Specify the path to a folder that you want to copy. If the folder is small,
        // you won't have time to see the progress dialog box.
        string sourcePath = @"C:\Windows\symbols\";
        // Choose a destination for the copied files.
        string destinationPath = @"C:\TestFolder";

        FileSystem.CopyDirectory(sourcePath, destinationPath,
            UIOption.AllDialogs);
    }
}
```

См. также

- Файловая система и реестр (руководство по программированию на C#)

Руководство по программированию на C#. Запись в текстовый файл

18.12.2021 • 2 minutes to read

В этой статье есть несколько примеров, демонстрирующих различные способы записи текста в файл. В первых двух примерах используются удобные статические методы класса System.IO.File для записи каждого элемента любого объекта `IEnumerable<string>` и `string` в текстовый файл. В третьем примере показано, как добавить текст в файл, если необходимо обрабатывать отдельно каждую строку по мере ее записи в файл. В первых трех примерах происходит перезапись всего существующего содержимого файла. В последнем примере показано, как добавить текст в существующий файл.

Все эти примеры записывают строковые литералы в файлы. Чтобы отформатировать записываемый в файл текст, воспользуйтесь методом Format или функцией интерполяции строк C#.

Запись коллекции строк в файл

```
using System.IO;
using System.Threading.Tasks;

class WriteAllLines
{
    public static async Task ExampleAsync()
    {
        string[] lines =
        {
            "First line", "Second line", "Third line"
        };

        await File.WriteAllLinesAsync("WriteLines.txt", lines);
    }
}
```

Предыдущий пример исходного кода:

- Создает экземпляр массива строк с тремя значениями.
- Ожидает вызов File.WriteAllLinesAsync, который:
  - Асинхронно создает имя файла *WriteLines.txt*. Если целевой файл уже существует, он будет перезаписан.
  - Записывает заданные строки в файл.
  - Закрывает файл, выполняет автоматическую очистку и удаление по мере необходимости.

Запись одной строки в файл

```
using System.IO;
using System.Threading.Tasks;

class WriteAllText
{
    public static async Task ExampleAsync()
    {
        string text =
            "A class is the most powerful data type in C#. Like a structure, " +
            "a class defines the data and behavior of the data type. ";

        await File.WriteAllTextAsync("WriteText.txt", text);
    }
}
```

Предыдущий пример исходного кода:

- Создает экземпляр строки с заданным назначенным строковым литералом.
- Ожидает вызов File.WriteAllTextAsync, который:
  - Асинхронно создает имя файла *WriteText.txt*. Если целевой файл уже существует, он будет перезаписан.
  - Записывает заданный текст в файл.
  - Закрывает файл, выполняет автоматическую очистку и удаление по мере необходимости.

Запись выбранных строк из массива в файл

```
using System.IO;
using System.Threading.Tasks;

class StreamWriterOne
{
    public static async Task ExampleAsync()
    {
        string[] lines = { "First line", "Second line", "Third line" };
        using StreamWriter file = new("WriteLines2.txt");

        foreach (string line in lines)
        {
            if (!line.Contains("Second"))
            {
                await file.WriteLineAsync(line);
            }
        }
    }
}
```

Предыдущий пример исходного кода:

- Создает экземпляр массива строк с тремя значениями.
- Создает экземпляр StreamWriter с путем к файлу *WriteLines2.txt* как с помощью объявления.
- Выполняет итерацию по всем строкам.
- Условно ожидает вызов StreamWriter.WriteLineAsync(String), который записывает строку в файл, если строка не содержит `"Second"`.

Добавление текста в существующий файл

```
using System.IO;
using System.Threading.Tasks;

class StreamWriterTwo
{
    public static async Task ExampleAsync()
    {
        using StreamWriter file = new("WriteLines2.txt", append: true);
        await file.WriteLineAsync("Fourth line");
    }
}
```

Предыдущий пример исходного кода:

- Создает экземпляр массива строк с тремя значениями.
- Создает экземпляр StreamWriter с путем к файлу *WriteLines2.txt* как с помощью объявления, передавая `true` для добавления.
- Ожидает вызов StreamWriter.WriteLineAsync(String), который записывает строку в файл как добавленную строку.

Исключения

При следующих условиях возможно возникновение исключения:

- InvalidOperationException: Файл существует и является файлом только для чтения.
- PathTooLongException: Имя пути имеет слишком большую длину.
- IOException: Диск может быть переполнен.

Существуют дополнительные условия, которые могут привести к возникновению исключений при работе с файловой системой, поэтому программировать следует с осторожностью.

См. также

- Руководство по программированию на C#
- Файловая система и реестр (руководство по программированию на C#)

Руководство по программированию на C#. Чтение из текстового файла

18.12.2021 • 2 minutes to read

В этом примере считывается содержимое текстового файла с помощью статических методов ReadAllText и ReadAllLines из класса System.IO.File.

См. пример использования StreamReader в руководстве по построчному чтению текстового файла.

> **NOTE**
> Файлы, которые используются в этом примере, созданы в руководстве по записи в текстовый файл.

Пример

```
class ReadFromFile
{
    static void Main()
    {
        // The files used in this example are created in the topic
        // How to: Write to a Text File. You can change the path and
        // file name to substitute text files of your own.

        // Example #1
        // Read the file as one string.
        string text = System.IO.File.ReadAllText(@"C:\Users\Public\TestFolder\WriteText.txt");

        // Display the file contents to the console. Variable text is a string.
        System.Console.WriteLine("Contents of WriteText.txt = {0}", text);

        // Example #2
        // Read each line of the file into a string array. Each element
        // of the array is one line of the file.
        string[] lines = System.IO.File.ReadAllLines(@"C:\Users\Public\TestFolder\WriteLines2.txt");

        // Display the file contents by using a foreach loop.
        System.Console.WriteLine("Contents of WriteLines2.txt = ");
        foreach (string line in lines)
        {
            // Use a tab to indent each line of the file.
            Console.WriteLine("\t" + line);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
```

Компиляция кода

Скопируйте код и вставьте его в консольное приложение C#.

Если вы не используете текстовые файлы из руководства по записи в текстовый файл, замените аргумент `ReadAllText` и `ReadAllLines` соответствующими путем и именем файла на компьютере.

Отказоустойчивость

При следующих условиях возможно возникновение исключения:

- Файл не существует или не существует в указанном месте. Проверьте правильность написания имени файла и путь к нему.

Безопасность .NET

Не следует полагаться на имя файла, чтобы определить содержимое файла. Например, файл `myFile.cs` может вовсе не быть исходным файлом C#.

См. также

- System.IO
- Руководство по программированию на C#
- Файловая система и реестр (руководство по программированию на C#)

Руководство по программированию на C#. Построчное чтение текстового файла

18.12.2021 • 2 minutes to read

В этом примере производится построчное чтение содержимого текстового файла в строку с помощью метода `ReadLines` класса `File`. Каждая строка текста сохраняется в строке `line` и отображается на экране.

Пример

```
int counter = 0;

// Read the file and display it line by line.
foreach (string line in System.IO.File.ReadLines(@"c:\test.txt"))
{
    System.Console.WriteLine(line);
    counter++;
}

System.Console.WriteLine("There were {0} lines.", counter);
// Suspend the screen.
System.Console.ReadLine();
```

Компиляция кода

Скопируйте код и вставьте его в метод `Main` консольного приложения.

Замените `"c:\test.txt"` фактическим именем файла.

Отказоустойчивость

При следующих условиях возможно возникновение исключения:

- Возможно, файл не существует.

Безопасность .NET

По имени файла не всегда можно с уверенностью судить о его содержимом. Например, файл с именем `myFile.cs` может вовсе не быть исходным файлом C#.

См. также

- System.IO
- Руководство по программированию на C#
- Файловая система и реестр (руководство по программированию на C#)

Руководство по программированию на C#. Создание раздела в реестре

18.12.2021 • 2 minutes to read

Код в этом примере добавляет в раздел Names реестра текущего пользователя пару значений — Name и Isabella.

Пример

```
Microsoft.Win32.RegistryKey key;
key = Microsoft.Win32.Registry.CurrentUser.CreateSubKey("Names");
key.SetValue("Name", "Isabella");
key.Close();
```

Компиляция кода

- Скопируйте код и вставьте его в метод `Main` консольного приложения.
- Замените параметр `Names` на имя ключа, который находится прямо в узле HKEY_CURRENT_USER реестра.
- Замените параметр `Name` на имя значения, которое находится прямо в узле Names.

Отказоустойчивость

Проверьте структуру реестра и найдите подходящее место для ключа. Для этого можно, например, открыть ключ программного обеспечения текущего пользователя и создать ключ с названием вашей компании. Затем добавьте в ключ компании значения реестра.

При следующих условиях может возникнуть исключение:

- Пустое имя ключа.
- У пользователя нет разрешения на создание разделов реестра.
- Имя ключа превышает ограничение в 255 символов.
- Раздел является закрытым.
- Раздел реестра доступен только для чтения.

Безопасность .NET

Безопаснее записывать данные в папку пользователя (`Microsoft.Win32.Registry.CurrentUser`), чем на локальный компьютер (`Microsoft.Win32.Registry.LocalMachine`).

Создавая значение реестра, необходимо решить, что делать, если это значение уже существует. Другой процесс (возможно, вредоносный) мог уже создать это значение и получить к нему доступ. Данные, добавленные в значение реестра, становятся доступными для другого процесса. Чтобы этого избежать, используйте `Overload:Microsoft.Win32.RegistryKey.GetValue` метод. Он возвращает значение NULL, если раздел еще не существует.

Небезопасно хранить секретные данные (например, пароли) в реестре как обычный текст, даже если раздел реестра защищен с помощью списков управления доступом (ACL).

См. также

- System.IO
- Руководство по программированию на C#
- Файловая система и реестр (руководство по программированию на C#)
- Чтение, запись и удаление данных реестра с помощью C#

Взаимодействие (Руководство по программированию в C#)

18.12.2021 • 2 minutes to read

Возможность взаимодействия позволяет использовать уже созданный неуправляемый код, экономя средства на разработку. Код, который выполняется под управлением среды CLR, называется *управляемым кодом*, а код, который выполняется вне среды CLR, называется *неуправляемым кодом*. COM, COM +, компоненты C++, компоненты ActiveX и Microsoft Windows API являются примерами неуправляемого кода.

.NET обеспечивает взаимодействие с неуправляемым кодом посредством служб вызова неуправляемого кода (PInvoke), пространства имен System.Runtime.InteropServices, COM-взаимодействия и взаимодействия с C++.

В этом разделе

Общие сведения о взаимодействии
Описывает способы взаимодействия между управляемым кодом C# и неуправляемым кодом.

Практическое руководство. Доступ к объектам взаимодействия Office с помощью функций языка C#
Описывает возможности, представленные в Visual C#, которые упрощают программирование для Office.

Практическое руководство. Использование индексированных свойств в программировании COM-взаимодействия
Описывает использование индексированных свойств для доступа к свойствам COM, которые имеют параметры.

Практическое руководство. Использование вызова неуправляемого кода для воспроизведения WAV-файла
Описывает, как использовать платформу вызова служб, чтобы воспроизвести звуковой WAV-файл в операционной системе Windows.

Пошаговое руководство. Программирование для Office
Описывает процесс создания книги Excel и документа Word со ссылкой на эту книгу.

Пример COM-класса
Предлагает пример предоставления класса C# в качестве COM-объекта.

Спецификация языка C#

Дополнительные сведения см. в разделе Основные понятия в Спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Marshal.ReleaseComObject
- Руководство по программированию на C#
- Взаимодействие с неуправляемым кодом
- Пошаговое руководство: Программирование для Office

Общие сведения о взаимодействии. (Руководство по программированию в C#)

18.12.2021 • 2 minutes to read

В этом разделе описываются способы включения взаимодействия между управляемым кодом C# и неуправляемым кодом.

Вызов неуправляемого кода

Вызов неуправляемого кода — это служба, позволяющая управляемому коду вызывать неуправляемые функции, реализованные в библиотеках динамической компоновки (DLL), например функции библиотек Microsoft Windows API. Он обнаруживает и вызывает экспортированную функцию и при необходимости маршалирует ее аргументы (целые числа, строки, массивы, структуры и так далее) через границы взаимодействия.

Дополнительные сведения см. руководстве по использованию неуправляемых функций DLL и руководстве по использованию вызова неуправляемого кода для воспроизведения звукового файла.

> **NOTE**
>
> Среда CLR управляет доступом к системным ресурсам. Вызов неуправляемого кода, который находится вне среды CLR, обходит этот механизм защиты и таким образом представляет угрозу безопасности. Например, неуправляемый код может обращаться к ресурсам в неуправляемом коде напрямую, минуя механизмы обеспечения безопасности среды CLR. Дополнительные сведения см. в разделе Безопасность в .NET.

взаимодействие C++

Взаимодействие C++, также известное как IJW, можно использовать для создания оболочки для собственного класса C++ для использования в коде, созданном на C# или другом языке .NET. Для этого необходимо написать код C++, создающий оболочку для собственного DLL- или COM-компонента. В отличие от других языков .NET, Visual C++ включает поддержку взаимодействия, что позволяет размещать управляемый и неуправляемый код в одном приложении и даже в одном файле. Затем вы выполняете сборку кода C++ с помощью параметра компилятора **/clr** для создания управляемой сборки. И, наконец, необходимо добавить ссылку на сборку в проекте C# и использовать объекты с оболочкой так же, как другие управляемые классы.

Предоставление доступа к COM-компонентам кода C#

Вы можете использовать COM-компоненты в проекте C#. Общая процедура следующая.

1. Найдите нужный COM-компонент и зарегистрируйте его. Используйте regsvr32.exe для регистрации или отмены регистрации библиотеки DLL модели COM.

2. Добавьте в проект ссылку на COM-компонент или библиотеку типов.

   При добавлении ссылки Visual Studio использует Tlbimp.exe (программу импорта библиотек типов), которая принимает на вход библиотеку типов и выдает на выходе сборку взаимодействия .NET. Сборка, также называемая вызываемой оболочкой времени выполнения (RCW), содержит управляемые классы и интерфейсы, которые служат оболочкой для классов и интерфейсов COM в библиотеке типов. Visual Studio добавляет в проект ссылку на созданную сборку.

3. Создайте экземпляр класса, определенного в вызываемой оболочке времени выполнения. Он, в свою очередь, создает экземпляр COM-объекта.

4. Используйте объект так же, как другие управляемые объекты. Когда объект удаляется сборщиком мусора, экземпляр COM-объекта также удаляется из памяти.

Дополнительные сведения см. в статье Предоставление клиентам .NET Framework доступа к COM-компонентам.

Предоставление C# клиентам COM

Клиенты COM могут потреблять типы C#, которые были правильно предоставлены. Ниже приведены основные шаги для предоставления типов C#.

1. Добавьте атрибуты взаимодействия в проект C#.

   Сборку COM можно сделать видимой, изменив свойства проекта Visual C#. Дополнительные сведения см. в разделе Диалоговое окно "Сведения о сборке".

2. Создайте библиотеку типов COM и зарегистрируйте ее для использования моделью COM.

   Можно изменить свойства проекта Visual C# для автоматической регистрации сборки C# для COM-взаимодействия. Visual Studio использует Regasm.exe (средство регистрации сборок) с параметром командной строки `/tlb`. Средство принимает на вход управляемую сборку и создает библиотеку типов. Эта библиотека типов описывает типы `public` в сборке и добавляет записи реестра, чтобы клиенты COM могли создавать управляемые классы.

Дополнительные сведения см. в разделе Предоставление COM-клиентам доступа к компонентам .NET Framework и Пример COM-класса.

См. также

- Улучшение производительности взаимодействия
- Общие сведения о взаимодействии между COM и .NET
- Общие сведения о взаимодействии между COM и Visual Basic
- Marshaling between Managed and Unmanaged Code (Маршалинг между управляемым и неуправляемым кодом)
- Взаимодействие с неуправляемым кодом
- Руководство по программированию на C#

Руководство по программированию на C#. Получение доступа к объектам взаимодействия

18.12.2021 • 11 minutes to read

В C# предусмотрены функции, которые упрощают доступ к объектам API Office. К новым функциям относятся именованные и необязательные аргументы, новый тип `dynamic`, а также возможность передавать аргументы ссылочным параметрам в методах COM, как если бы они были параметрами значений.

В этом разделе с использованием новых функций будет написан код, который создает и отображает лист Microsoft Office Excel. После этого будет написан код для добавления документа Office Word, который содержит значок, ссылающийся на лист Excel.

Для выполнения данного пошагового руководства на компьютере должны быть установлены Microsoft Office Excel 2007 и Microsoft Office Word 2007 или более поздние версии продуктов.

> **NOTE**
> Отображаемые на компьютере имена или расположения некоторых элементов пользовательского интерфейса Visual Studio могут отличаться от указанных в следующих инструкциях. Это зависит от имеющегося выпуска Visual Studio и используемых параметров. Дополнительные сведения см. в разделе Персонализация среды IDE.

Создание нового проекта консольного приложения

1. Запустите Visual Studio.
2. В меню **Файл** выберите пункт **Создать**, а затем команду **Проект**. Откроется диалоговое окно **Новый проект** .
3. В области **Установленные шаблоны** разверните узел **Visual C#** и выберите **Windows**.
4. В верхней части диалогового окна **Новый проект** в качестве целевой платформы необходимо выбрать **.NET Framework 4** (или более позднюю версию).
5. В области **Шаблоны** щелкните **Консольное приложение**.
6. Введите имя проекта в поле **Имя**.
7. Нажмите кнопку **ОК**.

   В **обозревателе решений** появится новый проект.

Добавление ссылок

1. В **обозревателе решений** щелкните имя проекта правой кнопкой мыши и выберите пункт **Добавить ссылку**. Откроется диалоговое окно **Добавление ссылки**.
2. На странице **Сборки** в списке **Имя компонента** выберите **Microsoft.Office.Interop.Word**, а затем, удерживая нажатой клавишу CTRL, выберите **Microsoft.Office.Interop.Excel**. Если сборки отсутствуют, может потребоваться проверить, что они установлены и отображаются. См. практическое руководство по установке основных сборок взаимодействия Microsoft Office.

3. Нажмите кнопку **OK**.

Добавление необходимых директив using

1. В **обозревателе решений** щелкните правой кнопкой мыши файл *Program.cs* и выберите пункт **Просмотреть код**.

2. В начало файла кода добавьте следующие директивы `using`:

```
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

Создание списка банковских счетов

1. Вставьте следующее определение класса в файл **Program.cs** в класс `Program`.

```
public class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

2. Чтобы создать список `bankAccounts`, содержащий два счета, добавьте в метод `Main` следующий код.

```
// Create a list of accounts.
var bankAccounts = new List<Account> {
    new Account {
                  ID = 345678,
                  Balance = 541.27
                },
    new Account {
                  ID = 1230221,
                  Balance = -127.44
                }
};
```

Объявление метода, экспортирующего сведения о счетах в Excel

1. Чтобы настроить лист Excel, добавьте в класс `Program` следующий метод.

   У метода Add есть необязательный параметр для указания конкретного шаблона. Для необязательных параметров, впервые появившихся в C# 4, можно опускать аргумент, если нужно использовать значение параметра по умолчанию. Поскольку в следующем коде никакой аргумент не передается, в методе `Add` используется шаблон по умолчанию и создается новая книга. В эквивалентном операторе в более ранних версиях C# необходимо было использовать аргумент-местозаполнитель `ExcelApp.Workbooks.Add(Type.Missing)`.

```
static void DisplayInExcel(IEnumerable<Account> accounts)
{
    var excelApp = new Excel.Application();
    // Make the object visible.
    excelApp.Visible = true;

    // Create a new, empty workbook and add it to the collection returned
    // by property Workbooks. The new workbook becomes the active workbook.
    // Add has an optional parameter for specifying a particular template.
    // Because no argument is sent in this example, Add creates a new workbook.
    excelApp.Workbooks.Add();

    // This example uses a single workSheet. The explicit type casting is
    // removed in a later procedure.
    Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet;
}
```

2. Добавьте в конец метода `DisplayInExcel` следующий код. Этот код вставляет значения в первые два столбца первой строки листа.

```
// Establish column headings in cells A1 and B1.
workSheet.Cells[1, "A"] = "ID Number";
workSheet.Cells[1, "B"] = "Current Balance";
```

3. Добавьте в конец метода `DisplayInExcel` следующий код. Цикл `foreach` помещает сведения из списка счетов в первые два столбца последовательных строк листа.

```
var row = 1;
foreach (var acct in accounts)
{
    row++;
    workSheet.Cells[row, "A"] = acct.ID;
    workSheet.Cells[row, "B"] = acct.Balance;
}
```

4. Добавьте в конец метода `DisplayInExcel` следующий код, чтобы ширина столбца изменялась в соответствии с содержимым.

```
workSheet.Columns[1].AutoFit();
workSheet.Columns[2].AutoFit();
```

   В более ранних версиях C# требовалось явное приведение типов для этих операций, поскольку `ExcelApp.Columns[1]` возвращает тип `Object`, а метод `AutoFit` является методом Excel Range. Приведение показано в следующих строках.

```
((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

   C# 4 и более поздних версий преобразует возвращаемое значение `Object` в `dynamic` автоматически, если ссылка на сборку задана с помощью параметра компилятора EmbedInteropTypes или, что эквивалентно, если свойство Excel **Внедрить типы взаимодействия** имеет значение true. True является значением по умолчанию для этого свойства.

Запуск проекта

1. Добавьте в конец метода `Main` следующую строку.

```
// Display the list in an Excel spreadsheet.
DisplayInExcel(bankAccounts);
```

2. Нажмите клавиши CTRL+F5.

   Появится книга Excel, содержащая данные о двух счетах.

Добавление документа Word

1. Чтобы продемонстрировать дополнительные способы, совершенствующие программирование для Office в C# 4 и более поздних версиях, следующий код открывает приложение Word и создает значок со ссылкой на лист Excel.

   Вставьте метод `CreateIconInWordDoc`, указанный далее в этом шаге, в класс `Program`. `CreateIconInWordDoc` использует именованные и необязательные аргументы, чтобы упростить вызовы методов Add и PasteSpecial. В этих вызовах используются две новые возможности, появившиеся в C# 4, которые упрощают вызовы к методам COM, имеющим ссылочные параметры. Во-первых, аргументы можно передать в ссылочные параметры, как если бы они были параметрами значений. Это значит, что значения можно передавать напрямую без создания переменной для каждого ссылочного параметра. Компилятор создает временные переменные для хранения значений аргументов и уничтожает эти переменные после завершения вызываемого метода. Во-вторых, ключевое слово `ref` в списке аргументов можно опустить.

   У метода `Add` имеется четыре ссылочных параметра, все из которых являются необязательными. В C# 4.0 или более поздних версиях вы можете опустить аргументы для любого или для всех параметров, если требуется использовать значения по умолчанию. В C# 3.0 и более ранних версиях необходимо было указывать аргумент для каждого из параметров, и этот аргумент должен был быть переменной, поскольку параметры являются ссылочными.

   Метод `PasteSpecial` вставляет содержимое буфера обмена. У метода имеется семь ссылочных параметров, все из которых являются необязательными. Следующий код задает аргументы для двух из них: `Link` для создания ссылки на исходное содержимое буфера и `DisplayAsIcon` для отображения ссылки в виде значка. В C# 4.0 и более поздних версиях можно использовать именованные аргументы для этих двух параметров и опустить остальные аргументы. Хотя эти параметры являются ссылочными, использовать ключевое слово `ref` или создавать переменные для передачи аргументов не требуется. Значения можно передать напрямую. В C# 3.0 и более ранних версиях необходимо было передавать аргумент в виде переменной для каждого ссылочного параметра.

```
static void CreateIconInWordDoc()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;

    // The Add method has four reference parameters, all of which are
    // optional. Visual C# allows you to omit arguments for them if
    // the default values are what you want.
    wordApp.Documents.Add();

    // PasteSpecial has seven reference parameters, all of which are
    // optional. This example uses named arguments to specify values
    // for two of the parameters. Although these are reference
    // parameters, you do not need to use the ref keyword, or to create
    // variables to send in as arguments. You can send the values directly.
    wordApp.Selection.PasteSpecial( Link: true, DisplayAsIcon: true);
}
```

В C# 3.0 и более ранних версиях приходилось использовать следующий более сложный синтаксис.

```
static void CreateIconInWordDoc2008()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;

    // The Add method has four parameters, all of which are optional.
    // In Visual C# 2008 and earlier versions, an argument has to be sent
    // for every parameter. Because the parameters are reference
    // parameters of type object, you have to create an object variable
    // for the arguments that represents 'no value'.

    object useDefaultValue = Type.Missing;

    wordApp.Documents.Add(ref useDefaultValue, ref useDefaultValue,
        ref useDefaultValue, ref useDefaultValue);

    // PasteSpecial has seven reference parameters, all of which are
    // optional. In this example, only two of the parameters require
    // specified values, but in Visual C# 2008 an argument must be sent
    // for each parameter. Because the parameters are reference parameters,
    // you have to contruct variables for the arguments.
    object link = true;
    object displayAsIcon = true;

    wordApp.Selection.PasteSpecial( ref useDefaultValue,
                                    ref link,
                                    ref useDefaultValue,
                                    ref displayAsIcon,
                                    ref useDefaultValue,
                                    ref useDefaultValue,
                                    ref useDefaultValue);
}
```

2. Добавьте в конец метода `Main` следующую инструкцию.

```
// Create a Word document that contains an icon that links to
// the spreadsheet.
CreateIconInWordDoc();
```

3. Добавьте в конец метода `DisplayInExcel` следующую инструкцию. Метод `Copy` добавляет лист в буфер обмена.

```
// Put the spreadsheet contents on the clipboard. The Copy method has one
// optional parameter for specifying a destination. Because no argument
// is sent, the destination is the Clipboard.
workSheet.Range["A1:B3"].Copy();
```

4. Нажмите клавиши CTRL+F5.

   Появится документ Word, содержащий значок. Дважды щелкните значок, чтобы отобразить лист на переднем плане.

Задание свойства "Внедрить типы взаимодействия"

1. При вызове типа COM, который не требует во время выполнения основной сборки взаимодействия (PIA), можно использовать дополнительные усовершенствования. Избавление от зависимостей от PIA приводит к независимости версий и делает развертывание более простым. Дополнительные сведения о преимуществах программирования без основных сборок взаимодействия см. в руководстве по внедрению типов из управляемых сборок.

   Кроме того, писать программы стало проще, поскольку типы, принимаемые и возвращаемые методами COM, можно представить с помощью типа `dynamic` вместо типа `Object`. Переменные типа `dynamic` не вычисляются до времени выполнения, что позволяет обходиться без явного приведения. Дополнительные сведения см. в разделе Использование типа dynamic.

   В C# 4 внедрение сведений о типах вместо использования сборок PIA является поведением по умолчанию. За счет этого несколько приведенных выше примеров становятся проще, поскольку явное приведение не требуется. Например, объявление `worksheet` в методе `DisplayInExcel` записывается как `Excel._Worksheet workSheet = excelApp.ActiveSheet`, а не как `Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet`. Для вызовов `AutoFit` в рамках одного метода также требовалось бы явное приведение без поведения по умолчанию, поскольку `ExcelApp.Columns[1]` возвращает тип `Object`, а `AutoFit` является методом Excel. Приведение показано в следующем примере.

```
((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

2. Чтобы изменить поведение по умолчанию и использовать сборки PIA вместо внедрения сведений о типе, разверните узел **Ссылки** в **обозревателе решений** и выберите **Microsoft.Office.Interop.Excel** или **Microsoft.Office.Interop.Word**.

3. Если окно **Свойства** не отображается, нажмите клавишу F4.

4. В списке свойств найдите свойство **Внедрить типы взаимодействия** и измените его значение на **False**. Также можно выполнить компиляцию с помощью командной строки с использованием параметра компилятора References вместо EmbedInteropTypes.

Дополнительное форматирование таблицы

1. Замените два вызова `AutoFit` в методе `DisplayInExcel` следующей инструкцией.

```
// Call to AutoFormat in Visual C# 2010.
workSheet.Range["A1", "B3"].AutoFormat(
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

   У метода AutoFormat имеется семь параметров значений, и все они являются необязательными.

Именованные и необязательные аргументы позволяют задать аргументы для всех параметров, их части или ни для одного параметра. В приведенной выше инструкции аргумент задается только для одного из параметров, `Format`. Поскольку `Format` является первым параметром в списке, имя параметра указывать не требуется. Однако инструкция может быть проще для понимания, если указать имя параметра, как показано в следующем фрагменте кода.

```
// Call to AutoFormat in Visual C# 2010.
workSheet.Range["A1", "B3"].AutoFormat(Format:
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

2. Нажмите сочетание клавиш CTRL + F5, чтобы увидеть результат. Другие форматы представлены в перечислении XlRangeAutoFormat.

3. Сравните инструкцию в шаге 1 со следующим кодом, в котором показаны аргументы, необходимые в C# 3.0 или более ранних версиях.

```
// The AutoFormat method has seven optional value parameters. The
// following call specifies a value for the first parameter, and uses
// the default values for the other six.

// Call to AutoFormat in Visual C# 2008. This code is not part of the
// current solution.
excelApp.get_Range("A1", "B4").AutoFormat(Excel.XlRangeAutoFormat.xlRangeAutoFormatTable3,
    Type.Missing, Type.Missing, Type.Missing, Type.Missing, Type.Missing,
    Type.Missing);
```

Пример

Ниже приведен полный пример кода.

```
using System;
using System.Collections.Generic;
using System.Linq;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;

namespace OfficeProgramminWalkthruComplete
{
    class Walkthrough
    {
        static void Main(string[] args)
        {
            // Create a list of accounts.
            var bankAccounts = new List<Account>
            {
                new Account {
                              ID = 345678,
                              Balance = 541.27
                            },
                new Account {
                              ID = 1230221,
                              Balance = -127.44
                            }
            };

            // Display the list in an Excel spreadsheet.
            DisplayInExcel(bankAccounts);

            // Create a Word document that contains an icon that links to
            // the spreadsheet.
            CreateIconInWordDoc();
```

```
        }

        static void DisplayInExcel(IEnumerable<Account> accounts)
        {
            var excelApp = new Excel.Application();
            // Make the object visible.
            excelApp.Visible = true;

            // Create a new, empty workbook and add it to the collection returned
            // by property Workbooks. The new workbook becomes the active workbook.
            // Add has an optional parameter for specifying a praticular template.
            // Because no argument is sent in this example, Add creates a new workbook.
            excelApp.Workbooks.Add();

            // This example uses a single workSheet.
            Excel._Worksheet workSheet = excelApp.ActiveSheet;

            // Earlier versions of C# require explicit casting.
            //Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet;

            // Establish column headings in cells A1 and B1.
            workSheet.Cells[1, "A"] = "ID Number";
            workSheet.Cells[1, "B"] = "Current Balance";

            var row = 1;
            foreach (var acct in accounts)
            {
                row++;
                workSheet.Cells[row, "A"] = acct.ID;
                workSheet.Cells[row, "B"] = acct.Balance;
            }

            workSheet.Columns[1].AutoFit();
            workSheet.Columns[2].AutoFit();

            // Call to AutoFormat in Visual C#. This statement replaces the
            // two calls to AutoFit.
            workSheet.Range["A1", "B3"].AutoFormat(
                Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);

            // Put the spreadsheet contents on the clipboard. The Copy method has one
            // optional parameter for specifying a destination. Because no argument
            // is sent, the destination is the Clipboard.
            workSheet.Range["A1:B3"].Copy();
        }

        static void CreateIconInWordDoc()
        {
            var wordApp = new Word.Application();
            wordApp.Visible = true;

            // The Add method has four reference parameters, all of which are
            // optional. Visual C# allows you to omit arguments for them if
            // the default values are what you want.
            wordApp.Documents.Add();

            // PasteSpecial has seven reference parameters, all of which are
            // optional. This example uses named arguments to specify values
            // for two of the parameters. Although these are reference
            // parameters, you do not need to use the ref keyword, or to create
            // variables to send in as arguments. You can send the values directly.
            wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);
        }
    }

    public class Account
    {
        public int ID { get; set; }
        public double Balance { get; set; }
```

```
    }
}
```

См. также

- Type.Missing
- dynamic
- Использование типа dynamic
- Именованные и необязательные аргументы
- Использование именованных и необязательных аргументов в программировании приложений Office

Руководство по программированию на C#. Использование индексированных свойств в программировании COM-взаимодействия

18.12.2021 • 2 minutes to read

Индексированные свойства делают использование свойств COM с параметрами при программировании на C# более удобным. Индексированные свойства используются совместно с другими компонентами, представленными в Visual C#, например именованными и необязательными аргументами, новым типом (dynamic) и внедренными сведениями о типах для расширения возможностей программирования для Microsoft Office.

В более ранних версиях C# методы доступны как свойства только при условии, что у метода `get` нет параметров, а у метода `set` есть только один параметр значения. Однако не все свойства COM удовлетворяют этим ограничениям. Например, свойство Range[] Excel имеет метод доступа `get`, которому требуется параметр для имени диапазона. Раньше из-за отсутствия возможности прямого обращения к свойству `Range` приходилось использовать вместо этого метод `get_Range`, как показано в следующем примере.

```
// Visual C# 2008 and earlier.
var excelApp = new Excel.Application();
// . . .
Excel.Range targetRange = excelApp.get_Range("A1", Type.Missing);
```

Индексированные свойства позволяют вместо этого использовать следующий код:

```
// Visual C# 2010.
var excelApp = new Excel.Application();
// . . .
Excel.Range targetRange = excelApp.Range["A1"];
```

NOTE

В предыдущем примере также используются необязательные аргументы, которые позволяют опустить `Type.Missing`.

Аналогичным образом, чтобы задать значение свойства `Value` объекта Range в C# 3.0 и более ранних версиях, требуются два аргумента. Один предоставляет аргумент для необязательного параметра, указывающего тип значения диапазона. Другой предоставляет значение для свойства `Value`. Эти методы показаны в следующих примерах. В примерах ячейке A1 задается значение `Name`.

```
// Visual C# 2008.
targetRange.set_Value(Type.Missing, "Name");
// Or
targetRange.Value2 = "Name";
```

Индексированные свойства позволяют вместо этого использовать следующий код.

```
// Visual C# 2010.
targetRange.Value = "Name";
```

Разработчики не могут создавать собственные индексированные свойства. Эта возможность поддерживает только использование имеющихся индексированных свойств.

Пример

Ниже приведен полный пример кода. См. сведения о создании проекта, обращающегося к Office API, в руководстве по получению доступа к объектам взаимодействия Office с помощью функций C#.

```
// You must add a reference to Microsoft.Office.Interop.Excel to run
// this example.
using System;
using Excel = Microsoft.Office.Interop.Excel;

namespace IndexedProperties
{
    class Program
    {
        static void Main(string[] args)
        {
            CSharp2010();
            //CSharp2008();
        }

        static void CSharp2010()
        {
            var excelApp = new Excel.Application();
            excelApp.Workbooks.Add();
            excelApp.Visible = true;

            Excel.Range targetRange = excelApp.Range["A1"];
            targetRange.Value = "Name";
        }

        static void CSharp2008()
        {
            var excelApp = new Excel.Application();
            excelApp.Workbooks.Add(Type.Missing);
            excelApp.Visible = true;

            Excel.Range targetRange = excelApp.get_Range("A1", Type.Missing);
            targetRange.set_Value(Type.Missing, "Name");
            // Or
            //targetRange.Value2 = "Name";
        }
    }
}
```

См. также

- Именованные и необязательные аргументы
- dynamic
- Использование типа dynamic
- Использование именованных и необязательных аргументов в программировании приложений Office
- Практическое руководство. Доступ к объектам взаимодействия Office с помощью функций языка C#
- Пошаговое руководство: Программирование для Office

Руководство по программированию на C#. Вызов неуправляемого кода для воспроизведения WAV-файла

18.12.2021 • 2 minutes to read

В следующем примере кода C# показано, как использовать службы вызова неуправляемого кода для воспроизведения звуковых WAV-файлов в ОС Windows.

Пример

В этом примере DllImportAttribute используется для импорта точки входа метода `PlaySound` файла `winmm.dll` как `Form1 PlaySound()`. Пример включает простую форму Windows с кнопкой. При нажатии кнопки открывается стандартное диалоговое окно Windows OpenFileDialog, позволяющее выбрать файл для воспроизведения. Выбранный WAV-файл воспроизводится с помощью метода `PlaySound()` из метода библиотеки *winmm.dll*. Дополнительные сведения об этом методе см. в разделе Использование функции PlaySound со звуковыми WAV-файлами. Найдите и выберите файл с расширением WAV и нажмите кнопку **Открыть**, чтобы воспроизвести этот WAV-файл, используя вызов неуправляемого кода. В текстовом поле отображается полный путь к выбранному файлу.

За счет указанных ниже настроек в диалоговом окне **Открыть файлы** отображаются только файлы с расширением WAV:

```
dialog1.Filter = "Wav Files (*.wav)|*.wav";
```

```
using System.Windows.Forms;
using System.Runtime.InteropServices;

namespace WinSound
{
    public partial class Form1 : Form
    {
        private TextBox textBox1;
        private Button button1;

        public Form1()  // Constructor.
        {
            InitializeComponent();
        }

        [DllImport("winmm.DLL", EntryPoint = "PlaySound", SetLastError = true, CharSet = CharSet.Unicode,
ThrowOnUnmappableChar = true)]
        private static extern bool PlaySound(string szSound, System.IntPtr hMod, PlaySoundFlags flags);

        [System.Flags]
        public enum PlaySoundFlags : int
        {
            SND_SYNC = 0x0000,
            SND_ASYNC = 0x0001,
            SND_NODEFAULT = 0x0002,
            SND_LOOP = 0x0008,
            SND_NOSTOP = 0x0010,
            SND_NOWAIT = 0x00002000,
            SND_FILENAME = 0x00020000,
            SND_RESOURCE = 0x00040004
        }

        private void button1_Click(object sender, System.EventArgs e)
        {
            var dialog1 = new OpenFileDialog();

            dialog1.Title = "Browse to find sound file to play";
            dialog1.InitialDirectory = @"c:\";
            dialog1.Filter = "Wav Files (*.wav)|*.wav";
            dialog1.FilterIndex = 2;
            dialog1.RestoreDirectory = true;

            if (dialog1.ShowDialog() == DialogResult.OK)
            {
                textBox1.Text = dialog1.FileName;
                PlaySound(dialog1.FileName, new System.IntPtr(), PlaySoundFlags.SND_SYNC);
            }
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            // Including this empty method in the sample because in the IDE,
            // when users click on the form, generates code that looks for a default method
            // with this name. We add it here to prevent confusion for those using the samples.
        }
    }
}
```

Компиляция кода

1. Создайте проект приложения Windows Forms на C# в Visual Studio и назовите его **WinSound**.
2. Скопируйте указанный выше код и вставьте его поверх содержимого файла *Form1.cs*.
3. Скопируйте следующий код и вставьте его в файл *Form1.Designer.cs* в метод `InitializeComponent()`

после существующего кода.

```
this.button1 = new System.Windows.Forms.Button();
this.textBox1 = new System.Windows.Forms.TextBox();
this.SuspendLayout();
//
// button1
//
this.button1.Location = new System.Drawing.Point(192, 40);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(88, 24);
this.button1.TabIndex = 0;
this.button1.Text = "Browse";
this.button1.Click += new System.EventHandler(this.button1_Click);
//
// textBox1
//
this.textBox1.Location = new System.Drawing.Point(8, 40);
this.textBox1.Name = "textBox1";
this.textBox1.Size = new System.Drawing.Size(168, 20);
this.textBox1.TabIndex = 1;
this.textBox1.Text = "FIle path";
//
// Form1
//
this.AutoScaleDimensions = new System.Drawing.SizeF(5, 13);
this.ClientSize = new System.Drawing.Size(292, 266);
this.Controls.Add(this.textBox1);
this.Controls.Add(this.button1);
this.Name = "Form1";
this.Text = "Platform Invoke WinSound C#";
this.ResumeLayout(false);
this.PerformLayout();
```

4. Скомпилируйте и запустите код.

См. также

- Руководство по программированию на C#
- Общие сведения о взаимодействии
- Подробный обзор вызова неуправляемого кода
- Маршалинг данных при вызове неуправляемого кода

Пошаговое руководство. Программирование для Office (C# и Visual Basic)

18.12.2021 • 10 minutes to read

Visual Studio предлагает новые функции C# и Visual Basic, позволяющие улучшить программирование для Microsoft Office. В число полезных функций C# входят именованные и необязательные аргументы и возвращаемые значения типа `dynamic`. В программировании COM можно опустить ключевое слово `ref` и получить доступ к индексированным свойствам. Список функций Visual Basic включает автоматически реализуемые свойства, инструкции в лямбда-выражениях и инициализаторы коллекций.

Оба языка поддерживают внедрение сведений о типах, что позволяет развертывать сборки, взаимодействующие с компонентами COM, без предварительного развертывания на компьютере основных сборок взаимодействия (PIA). Дополнительные сведения см. в разделе Пошаговое руководство: внедрению типов из управляемых сборок.

В данном пошаговом руководстве эти возможности показаны в контексте программирования для Microsoft Office, но многие из них могут оказаться полезными и в других ситуациях. В этом пошаговом руководстве вы создадите книгу Excel с помощью надстройки Excel, а затем документ Word со ссылкой на эту книгу. Наконец, вы узнаете, как включать и отключать зависимость PIA.

Предварительные требования

Для выполнения данного пошагового руководства на компьютере должны быть установлены Microsoft Office Excel и Microsoft Office Word.

> **NOTE**
> Отображаемые на компьютере имена или расположения некоторых элементов пользовательского интерфейса Visual Studio могут отличаться от указанных в следующих инструкциях. Это зависит от имеющегося выпуска Visual Studio и используемых параметров. Дополнительные сведения см. в разделе Персонализация среды IDE.

Настройка надстройки Excel

1. Запустите Visual Studio.
2. В меню **Файл** выберите пункт **Создать**, а затем команду **Проект**.
3. В области **Установленные шаблоны** разверните узел **Visual Basic** или **Visual C#** , а затем узел **Office** и выберите версию Office.
4. В области **Шаблоны** выберите **Надстройка Excel <version>** .
5. Убедитесь, что в верхней части области **Шаблоны** в поле **Требуемая версия** .NET Framework отображается **.NET Framework 4** или более поздняя версия.
6. Если нужно, в поле **Имя** введите имя проекта.
7. Нажмите кнопку **ОК**.
8. В **обозревателе решений** появится новый проект.

Добавление ссылок

1. В **обозревателе решений** щелкните имя проекта правой кнопкой мыши и выберите пункт

Добавить ссылку. Откроется диалоговое окно **Добавление ссылки**.

2. На вкладке **Сборки** в списке **Имя компонента** выберите Microsoft.Office.Interop.Excel, версия `<version>.0.0.0` (расшифровку номеров версий продуктов Office см. в разделе Версии Майкрософт), а затем, удерживая нажатой клавишу CTRL, выберите Microsoft.Office.Interop.Word, `version <version>.0.0.0`. Если сборки отсутствуют, может потребоваться проверить, что они установлены и отображаются (см. раздел Практическое руководство. Установка основных сборок взаимодействия Microsoft Office).

3. Нажмите кнопку **ОК**.

Добавление необходимых операторов Imports или директив using

1. В **обозревателе решений** щелкните правой кнопкой мыши файл ThisAddIn.vb или ThisAddIn.cs и выберите в контекстном меню команду **Просмотреть код**.

2. В верхнюю часть файла с кодом добавьте следующие операторы `Imports` (Visual Basic) или директивы `using`, если это еще не сделано.

```
using System.Collections.Generic;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

```
Imports Microsoft.Office.Interop
```

Создание списка банковских счетов

1. В **обозревателе решений** щелкните правой кнопкой мыши имя проекта, выберите в контекстном меню команду **Добавить**, а затем щелкните пункт **Класс**. Назовите класс Account.vb, если используется Visual Basic, или Account.cs, если используется C#. Нажмите кнопку **Добавить**.

2. Замените определение класса `Account` следующим кодом. В определениях классов используются *автоматически реализуемые свойства*. Дополнительные сведения см. в разделе Автоматически реализуемые свойства.

```
class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

```
Public Class Account
    Property ID As Integer = -1
    Property Balance As Double
End Class
```

3. Чтобы создать список `bankAccounts`, содержащий два счета, добавьте в метод `ThisAddIn_Startup` в файле *ThisAddIn.vb* или *ThisAddIn.cs* следующий код. В объявлениях списков используются *инициализаторы коллекций*. Дополнительные сведения см. в разделе Инициализаторы коллекций.

```
var bankAccounts = new List<Account>
{
    new Account
    {
        ID = 345,
        Balance = 541.27
    },
    new Account
    {
        ID = 123,
        Balance = -127.44
    }
};
```

```
Dim bankAccounts As New List(Of Account) From {
    New Account With {
                          .ID = 345,
                          .Balance = 541.27
                      },
    New Account With {
                          .ID = 123,
                          .Balance = -127.44
                      }
    }
```

Экспорт данных в Excel

1. В том же самом файле добавьте в класс `ThisAddIn` следующий метод. Этот метод служит для настройки книги Excel и экспорта данных в нее.

```
void DisplayInExcel(IEnumerable<Account> accounts,
           Action<Account, Excel.Range> DisplayFunc)
{
    var excelApp = this.Application;
    // Add a new Excel workbook.
    excelApp.Workbooks.Add();
    excelApp.Visible = true;
    excelApp.Range["A1"].Value = "ID";
    excelApp.Range["B1"].Value = "Balance";
    excelApp.Range["A2"].Select();

    foreach (var ac in accounts)
    {
        DisplayFunc(ac, excelApp.ActiveCell);
        excelApp.ActiveCell.Offset[1, 0].Select();
    }
    // Copy the results to the Clipboard.
    excelApp.Range["A1:B3"].Copy();
}
```

```
Sub DisplayInExcel(ByVal accounts As IEnumerable(Of Account),
                   ByVal DisplayAction As Action(Of Account, Excel.Range))

    With Me.Application
        ' Add a new Excel workbook.
        .Workbooks.Add()
        .Visible = True
        .Range("A1").Value = "ID"
        .Range("B1").Value = "Balance"
        .Range("A2").Select()

        For Each ac In accounts
            DisplayAction(ac, .ActiveCell)
            .ActiveCell.Offset(1, 0).Select()
        Next

        ' Copy the results to the Clipboard.
        .Range("A1:B3").Copy()
    End With
End Sub
```

В этом методе используются две новые возможности C#. Оба эти возможности уже существуют в Visual Basic.

- У метода Add есть *необязательный параметр* для указания конкретного шаблона. Для необязательных параметров, впервые появившихся в C# 4, можно опускать аргумент, если нужно использовать значение параметра по умолчанию. Поскольку в предыдущем примере никакой аргумент не передается, в методе `Add` используется шаблон по умолчанию и создается новая книга. В эквивалентном операторе в более ранних версиях C# необходимо было использовать аргумент-местозаполнитель `excelApp.Workbooks.Add(Type.Missing)`.

  Дополнительные сведения см. в разделе Именованные и необязательные аргументы.

- Свойства `Range` и `Offset` объекта Range используют возможность *индексированных свойств*. Она позволяет использовать свойства типов COM с помощью стандартного синтаксиса C#. Кроме того, индексированные свойства позволяют использовать свойство `Value` объекта `Range`, устраняя необходимость в использовании свойства `Value2`. Свойство `Value` является индексированным, но индекс — необязательным. Совместная работа необязательных аргументов и индексированных свойств показана в следующем примере.

  ```
  // Visual C# 2010 provides indexed properties for COM programming.
  excelApp.Range["A1"].Value = "ID";
  excelApp.ActiveCell.Offset[1, 0].Select();
  ```

  В более ранних версиях этого языка приходилось использовать особый синтаксис.

  ```
  // In Visual C# 2008, you cannot access the Range, Offset, and Value
  // properties directly.
  excelApp.get_Range("A1").Value2 = "ID";
  excelApp.ActiveCell.get_Offset(1, 0).Select();
  ```

  Разработчики не могут создавать собственные индексированные свойства. Эта возможность поддерживает только использование имеющихся индексированных свойств.

  Дополнительные сведения см. в статье Практическое руководство. Использование индексированных свойств в программировании COM-взаимодействия (Руководство по программированию на C#).

2. Добавьте в конец метода `DisplayInExcel` следующий код, чтобы ширина столбца изменялась в соответствии с содержимым.

```
excelApp.Columns[1].AutoFit();
excelApp.Columns[2].AutoFit();
```

```
' Add the following two lines at the end of the With statement.
.Columns(1).AutoFit()
.Columns(2).AutoFit()
```

Эти дополнения демонстрируют еще одну возможность C#: значения `Object`, возвращаемые главными приложениями COM, например приложениями Office, и обрабатываются так, как если бы они имели тип dynamic. Это происходит автоматически, если параметр **Внедрить типы взаимодействия** имеет значение по умолчанию (`True`), или, что эквивалентно, если ссылка на сборку задается с помощью параметра компилятора **EmbedInteropTypes**. Тип `dynamic` делает возможным позднее связывание, уже доступное в Visual Basic, и не допускает явного приведения, которое требовалось бы в C# 3.0 и более ранних версиях языка.

Например, `excelApp.Columns[1]` возвращает `Object`, а `AutoFit` является методом Excel Range. Без типа `dynamic` необходимо выполнять приведение объекта, возвращаемого `excelApp.Columns[1]`, к экземпляру `Range` перед вызовом метода `AutoFit`.

```
// Casting is required in Visual C# 2008.
((Excel.Range)excelApp.Columns[1]).AutoFit();

// Casting is not required in Visual C# 2010.
excelApp.Columns[1].AutoFit();
```

Дополнительные сведения о внедрении типов взаимодействия см. в подразделах "Поиск ссылки PIA" и "Восстановление зависимости PIA" далее в этом разделе. Дополнительные сведения о `dynamic` см. в разделе dynamic или Использование типа dynamic.

Вызов метода DisplayInExcel

1. Добавьте следующий код в конец метода `ThisAddIn_StartUp`. Вызов метода `DisplayInExcel` содержит два аргумента. Первый аргумент представляет собой имя списка счетов, которые требуется обработать. Второй аргумент — это состоящее из нескольких строк лямбда-выражение, которое определяет, каким образом следует обрабатывать данные. Значения `ID` и `balance` для каждого из счетов отображаются в соседних ячейках, а если баланс имеет отрицательное значение, строка отображается красным. Дополнительные сведения см. в разделе Лямбда-выражения.

```
DisplayInExcel(bankAccounts, (account, cell) =>
// This multiline lambda expression sets custom processing rules
// for the bankAccounts.
{
    cell.Value = account.ID;
    cell.Offset[0, 1].Value = account.Balance;
    if (account.Balance < 0)
    {
        cell.Interior.Color = 255;
        cell.Offset[0, 1].Interior.Color = 255;
    }
});
```

```
DisplayInExcel(bankAccounts,
       Sub(account, cell)
           ' This multiline lambda expression sets custom
           ' processing rules for the bankAccounts.
           cell.Value = account.ID
           cell.Offset(0, 1).Value = account.Balance

           If account.Balance < 0 Then
               cell.Interior.Color = RGB(255, 0, 0)
               cell.Offset(0, 1).Interior.Color = RGB(255, 0, 0)
           End If
       End Sub)
```

2. Чтобы запустить программу, нажмите клавишу F5. Появится книга Excel, содержащая данные о счетах.

Добавление документа Word

1. Добавьте в конец метода `ThisAddIn_StartUp` следующий код, чтобы создать документ Word, содержащий ссылку на книгу Excel.

```
var wordApp = new Word.Application();
wordApp.Visible = true;
wordApp.Documents.Add();
wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);
```

```
Dim wordApp As New Word.Application
wordApp.Visible = True
wordApp.Documents.Add()
wordApp.Selection.PasteSpecial(Link:=True, DisplayAsIcon:=True)
```

   В этом коде демонстрируются некоторые новые функции C#: возможность опускать ключевое слово `ref` при программировании в модели COM, именованные аргументы и необязательные аргументы. Эти возможности уже существовали в Visual Basic. Метод PasteSpecial имеет семь параметров, которые определены как необязательные ссылочные параметры. Именованные и необязательные аргументы позволяют определять параметры, к которым требуется обращаться по имени, и передавать аргументы только для этих параметров. В этом примере аргументы передаются, чтобы показать, что необходимо создать ссылку на книгу в буфере (параметр `Link`) и что эта ссылка должна отображаться в документе Word в виде значка (параметр `DisplayAsIcon`). Кроме того, Visual C# позволяет опускать ключевое слово `ref` для таких аргументов.

Запуск приложения

1. Нажмите клавишу F5 для запуска приложения. Будет запущено приложение Excel, в котором будет открыта таблица, содержащая сведения о двух счетах из списка `bankAccounts`. Затем будет открыт документ Word, содержащий ссылку на таблицу Excel.

Очистка готового проекта

1. В Visual Studio в меню **Построение** выберите пункт **Очистить решение**. В противном случае надстройка будет запускаться при каждом открытии Excel на компьютере разработчика.

Поиск ссылки PIA

1. Запустите приложение снова, но не выбирайте пункт **Очистить решение**.

2. Выберите кнопку **Пуск**. Найдите **Microsoft Visual Studio <version>** и откройте командную строку разработчика.

3. В окне командной строки разработчика для Visual Studio введите команду `ildasm`, а затем нажмите клавишу ВВОД. Появится окно программы IL DASM.

4. В меню **Файл** в окне IL DASM выберите пункт **Файл** > **Открыть**. Дважды щелкните **Visual Studio <version>** и **Проекты**. Откройте папку проекта и найдите в папке bin/Debug файл *имя_проекта*.dll. Дважды щелкните файл *имя_проекта*.dll. В новом окне будут показаны атрибуты проекта, а также ссылки на другие модули и сборки. Обратите внимание, что в сборку включены пространства имен `Microsoft.Office.Interop.Excel` и `Microsoft.Office.Interop.Word`. По умолчанию в Visual Studio компилятор импортирует в сборку необходимые типы из сборки PIA, на которую указывает ссылка.

   Дополнительные сведения см. в разделе Практическое руководство. просмотреть одержимое сборки.

5. Дважды щелкните значок **МАНИФЕСТ**. Откроется окно со списком сборок, содержащих элементы, на которые имеются ссылки в проекте. Сборки `Microsoft.Office.Interop.Excel` и `Microsoft.Office.Interop.Word` не будут указаны в этом списке. Поскольку необходимые для проекта типы были импортированы в сборку проекта, ссылки на сборки PIA не требуется. Это упрощает развертывание. Сборки PIA не обязательно должны присутствовать на компьютере пользователя, а поскольку приложение не требует развертывания конкретной версии сборки PIA, можно разрабатывать приложения, которые работают с различными версиями Office, если в этих версиях имеются все необходимые интерфейсы API.

   Поскольку развертывать сборки PIA больше не требуется, можно создавать приложения для применения в сложных сценариях, чтобы эти приложения работали с несколькими версиями Office, включая и более ранние версии. Тем не менее это возможно только в том случае, если в коде не используются интерфейсы API, которые недоступны в используемой версии Office. Разработчик не всегда знает, был ли доступен тот или иной интерфейс API в более ранней версии, поэтому работать с более ранними версиями Office не рекомендуется.

   > **NOTE**
   > До Office 2003 сборки PIA не публиковались. Поэтому единственными способом создания сборки взаимодействия в Office 2002 или более ранних версиях является импорт ссылки COM.

6. Закройте окно манифеста и окно сборки.

Восстановление зависимости PIA

1. В **обозревателе решений** нажмите кнопку **Показать все файлы**. Разверните папку **Ссылки** и выберите **Microsoft.Office.Interop.Excel**. Нажмите клавишу F4, чтобы открыть окно **Свойства**.

2. В окне **Свойства** измените значение свойства **Внедрить типы взаимодействия** с **True** на **False**.

3. Повторите шаги 1 и 2 этой процедуры для сборки `Microsoft.Office.Interop.Word`.

4. В C# раскомментируйте два вызова метода `Autofit` в конце метода `DisplayInExcel`.

5. Нажмите клавишу F5, чтобы проверить, что проект по-прежнему выполняется правильно.

6. Повторите шаги 1–3 из предыдущей процедуры, чтобы открыть окно сборки. Обратите внимание, что сборки `Microsoft.Office.Interop.Word` и `Microsoft.Office.Interop.Excel` больше не входят в список внедренных сборок.

7. Дважды щелкните значок **МАНИФЕСТ** и просмотрите список сборок, на которые имеются ссылки. В списке будут указаны сборки `Microsoft.Office.Interop.Word` и `Microsoft.Office.Interop.Excel`.

Поскольку приложение содержит ссылки на сборки PIA Excel и Word, а свойство **Внедрить типы взаимодействия** имеет значение **False**, на компьютере пользователя должны храниться обе сборки.

8. В Visual Studio в меню **Построение** выберите пункт **Очистить решение**, чтобы очистить завершенный проект.

См. также

- Автоматически реализуемые свойства (Visual Basic)
- Автоматически реализуемые свойства (C#)
- Инициализаторы коллекций
- Инициализаторы объектов и коллекций
- Необязательные параметры
- Передача аргументов по позиции и по имени
- Именованные и необязательные аргументы
- Раннее и позднее связывание
- dynamic
- Использование типа dynamic
- Лямбда-выражения (Visual Basic)
- Лямбда-выражения (C#)
- Практическое руководство. Использование индексированных свойств в программировании COM-взаимодействия
- Пошаговое руководство: Внедрение данных о типах из сборок Microsoft Office в Visual Studio (C#)
- Пошаговое руководство: внедрение типов из управляемых сборок
- Пошаговое руководство: создание первой надстройки VSTO для Excel
- COM-взаимодействие
- Взаимодействие

Пример COM-класса (Руководство по программированию на C#)

18.12.2021 • 2 minutes to read

Далее приведен пример класса, который можно предоставить в качестве COM-объекта. После помещения этого кода в CS-файл и добавления в проект свойства **Регистрация для COM-взаимодействия** необходимо присвоить значение **Истина**. Дополнительные сведения см. в разделе Практическое руководство. Register a Component for COM Interop (Практическое руководство. Регистрация компонента для COM-взаимодействия).

Предоставление объектов Visual C# для COM требует объявления интерфейса класса, интерфейса событий (если необходимо) и самого класса. Члены класса должны соответствовать указанным ниже правилам, чтобы стать доступными для COM.

- Класс должен быть открытым.
- Свойства, методы и события должны быть открытыми.
- Свойства и методы должны быть объявлены в интерфейсе класса.
- События должны быть объявлены в интерфейсе событий.

Другие открытые элементы в классе, которые не объявлены в этих интерфейсах, не будут доступны для COM, но будут доступны другим объектам .NET.

Чтобы предоставить свойства и методы COM, их необходимо объявить в интерфейсе класса, пометить атрибутом `DispId` и реализовать в классе. Порядок объявления членов в интерфейсе — это порядок, используемый для виртуальной таблицы COM.

Чтобы предоставить события из класса, их необходимо объявить в интерфейсе событий и пометить атрибутом `DispId`. Класс не должен реализовывать этот интерфейс.

Класс реализует интерфейс класса; он может реализовывать несколько интерфейсов, но первой реализацией будет интерфейс класса по умолчанию. Реализуйте методы и свойства, доступные модели COM здесь. Они должны быть помечены как открытые и соответствовать объявлениям в интерфейсе класса. Кроме того, объявите здесь события, инициируемые классом. Они должны быть помечены как открытые и соответствовать объявлениям в интерфейсе событий.

Пример

```
using System.Runtime.InteropServices;

namespace project_name
{
    [Guid("EAA4976A-45C3-4BC5-BC0B-E474F4C3C83F")]
    public interface ComClass1Interface
    {
    }

    [Guid("7BD20046-DF8C-44A6-8F6B-687FAA26FA71"),
        InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
    public interface ComClass1Events
    {
    }

    [Guid("0D53A3E8-E51A-49C7-944E-E72A2064F938"),
        ClassInterface(ClassInterfaceType.None),
        ComSourceInterfaces(typeof(ComClass1Events))]
    public class ComClass1 : ComClass1Interface
    {
    }
}
```

См. также

- Руководство по программированию на C#
- Взаимодействие
- Страница "Сборка" в конструкторе проектов (C#)

Справочник по C#

18.12.2021 • 3 minutes to read

Этот раздел содержит подробные справочные сведения о ключевых словах, операторах, специальных символах, директивах препроцессора, параметрах компилятора и ошибках и предупреждениях компилятора в среде C#.

Содержание раздела

Ключевые слова в C#

Ссылки на сведения о ключевых словах и синтаксисе языка C#.

Операторы в C#

Ссылки на сведения об операторах и синтаксисе языка C#.

Специальные символы в C#

Предоставляет ссылки на сведения о специальных контекстные символов в C# и их использовании.

Директивы препроцессора C#

Ссылки на сведения о командах компилятора для внедрения в исходном коде C#.

Параметры компилятора C#

Сведения о параметрах компилятора и их использовании.

Ошибки компилятора C#

Фрагменты кода, демонстрирующие причины и способы исправления ошибок и предупреждений компилятора C#.

Спецификация языка C#

Спецификация языка C# версии 6.0 Это черновой вариант для языка C# версии 6.0. Этот документ будет пересмотрен в рамках работы с комитетом по стандартам C# ECMA. Версия 5.0 была выпущена в декабре 2017 г. как стандартный 5-й выпуск ECMA-334.

Возможности, которые были реализованы в языке C# версий более поздних чем 6.0, представлены в предложениях по спецификации языка. В этих документах описываются изменения спецификации языка, связанные с добавлением новых функций. Это черновой вариант для формы. Эти спецификации будут улучшены и отправлены в комитет по стандартам ECMA для официального анализа и внедрения в будущую версию стандарта C#.

Предложения по спецификации C# 7.0

В версии C# 7.0 реализован ряд новых возможностей, включая сопоставления шаблонов, локальные функции, объявления выходной переменной, выражения throw, двоичные литералы и разделители между цифрами. Эта папка содержит спецификации для каждой из этих функций.

Предложения по спецификации C# 7.1

В версию C# 7.1 добавлено несколько новых возможностей. Можно написать метод `Main`, возвращающий `Task` или `Task<int>`. Это позволяет добавлять модификатор `async` в метод `Main`. Выражение `default` можно использовать без типа в тех расположениях, где возможен вывод типа. Кроме того, появилось еще одно дополнительное усовершенствование: вывод имен элементов кортежа. И, наконец, сопоставление шаблонов можно использовать с универсальными шаблонами.

Предложения по спецификации C# 7.2

В версию C#7.2 добавлен ряд простых функций. С помощью ключевого слова `in` можно передавать

аргументы по ссылке только для чтения. Внесен ряд незначительных изменений для поддержки безопасности во время компиляции для `Span` и связанных типов. В некоторых ситуациях можно использовать именованные аргументы, если следующие за ними аргументы являются позиционными. Модификатор доступа `private protected` позволяет указывать, что вызывающие объекты ограничены производными типами, реализованными в той же сборке. Оператор `?:` можно использовать для разрешения в ссылку на переменную. С помощью разделителя начальных цифр можно форматировать шестнадцатеричные и двоичные числа.

Предложения по спецификации C# 7.3

Версия C# 7.3 является очередным промежуточным выпуском, содержащим несколько небольших обновлений. К параметрам универсальных типов можно применять новые ограничения. Другие изменения упрощают работу с полями `fixed`, включая использование выделений `stackalloc`. Локальные переменные, объявленные с ключевым словом `ref`, можно переназначать для указания на новое хранилище. Можно применять атрибуты к автоматически реализуемым свойствам, предназначенным для созданного компилятором резервного поля. Переменные выражений можно использовать в инициализаторах. Кортежи можно проверять на равенство (или неравенство). Кроме того, были внесены некоторые улучшения в разрешение перегрузки.

Предложения по спецификации C# 8.0

Версия C# 8.0 доступна для .NET Core 3.0. В число возможностей входят использование ссылочных типов, допускающих значения NULL, рекурсивное сопоставление шаблонов, методы интерфейса по умолчанию, асинхронные потоки, диапазоны и индексы, использование шаблонов и объявлений using, назначение объединения со значением NULL и члены экземпляров с доступом только на чтение.

Предложения по спецификации C# 9

Версия C# 9 доступна для .NET 5. Предлагаемые функции включают записи, инструкции верхнего уровня, усовершенствованное сопоставление шаблонов, методы задания только для инициализации, выражения new с целевым типом, модульные инициализаторы, расширение методов partial, статические анонимные функции, условные выражения с целевым типом, ковариантные типы возвращаемого значения, метод расширения GetEnumerator в циклах foreach, пустые переменные в качестве параметров лямбда-выражений, атрибуты для локальных функций, целочисленные значения с системным размером, указатели на функцию, подавляемое порождение флага localsinit и аннотации для параметров неограниченного типа.

Предложения по спецификации C# 10

Версия C# 10 доступна для .NET 6. Предлагаемые функции включают структуры записи, конструкторы структур без параметров, глобальные директивы "using", пространства имен на уровне файлов, шаблоны расширенных свойств, улучшенные интерполированные строки, постоянные интерполированные строки, улучшения функции лямбда, поддержку атрибута сведений о вызывающем выражении аргумента, расширенные директивы `#line`, универсальные атрибуты, улучшенный анализ определенного присваивания и переопределение `AsyncMethodBuilder`.

Связанные разделы

Использование среды разработки Visual Studio для C#

Ссылки на концептуальные разделы и разделы задач, описывающие интегрированную среду разработки и редактор.

Руководство по программированию на C#

Сведения об использовании языка программирования C#.

Управление версиями языка C#

18.12.2021 • 4 minutes to read

Компилятор C# последней версии определяет версию языка по умолчанию на основе целевой платформы или платформ проекта. Visual Studio не предоставляет пользовательский интерфейс для изменения этого значения, но его можно изменить, отредактировав файл *CSPROJ*. Выбор значения по умолчанию гарантирует, что вы используете последнюю версию языка, совместимую с вашей целевой платформой. Вы получите преимущество в виде доступа к последним функциям языка, совместимым с целевым объектом проекта. Этот вариант по умолчанию также гарантирует, что вы не будете использовать язык, который требует такие типы или поведение во время выполнения, которые недоступны в целевой платформе. Выбор более новой версии языка, чем значение по умолчанию, может усложнить диагностику ошибок во время компиляции и выполнения.

Приведенные в этой статье правила относятся к компилятору, поставляемому с Visual Studio 2019, или к пакету SDK для .NET. Компиляторы C#, которые являются частью установки Visual Studio 2017 или более ранних версий пакета SDK для .NET Core предназначены для C# 7.0 по умолчанию.

C# 8.0 поддерживается только в .NET Core 3.x и более поздних версиях. Для многих новых функций нужны функции среды выполнения и библиотеки, появившиеся в .NET Core 3. x:

- Для реализации интерфейса по умолчанию требуются новые функции в среде CLR .NET Core 3.0.
- Для асинхронных потоков требуются новые типы System.IAsyncDisposable, System.Collections.Generic.IAsyncEnumerable<T> и System.Collections.Generic.IAsyncEnumerator<T>.
- Для индексов и диапазонов требуются новые типы System.Index и System.Range.
- Ссылочные типы, допускающие значение NULL, используют несколько атрибутов для предоставления улучшенных предупреждений. Эти атрибуты были добавлены в .NET Core 3.0. Другие целевые платформы не были помечены ни одним из этих атрибутов. Это означает, что предупреждения, допускающие значение NULL, могут неточно отражать потенциальные проблемы.

C# 9 поддерживается только в .NET 5 и более поздних версиях.

C# 10 поддерживается только в .NET 6 и более поздних версиях.

Значения по умолчанию

Компилятор определяет значение по умолчанию на основе следующих правил:

ЦЕЛЕВАЯ ПЛАТФОРМА	VERSION	ВЕРСИЯ ЯЗЫКА C# ПО УМОЛЧАНИЮ
.NET	6.x	C# 10
.NET	5.x	C# 9.0
.NET Core	3.x	C# 8.0
.NET Core	2.x	C# 7.3
.NET Standard	2.1	C# 8.0
.NET Standard	2.0	C# 7.3

ЦЕЛЕВАЯ ПЛАТФОРМА	VERSION	ВЕРСИЯ ЯЗЫКА C# ПО УМОЛЧАНИЮ
.NET Standard	1.x	C# 7.3
.NET Framework	все	C# 7.3

Если проект предназначен для платформы в предварительной версии с поддержкой соответствующего языка в предварительной версии, будет использоваться язык, поддерживаемый в предварительной версии. Вы можете использовать новейшие возможности в этой предварительной версии в любой среде, не затрагивая проекты, предназначенные для выпущенной версии .NET Core.

> **IMPORTANT**
> В Visual Studio 2017 во все создаваемые файлы проектов добавлялась запись `<LangVersion>latest</LangVersion>`. На момент добавления последней считалась версия *C# 7.0*. Но после обновления до Visual Studio 2019 эта запись указывает на последнюю выпущенную версию, независимо от целевой платформы. Теперь для этих проектов переопределяется поведение по умолчанию. Вам нужно изменить файл проекта и удалить этот узел. После этого проект будет использовать версию компилятора, рекомендованную для вашей целевой платформы.

Переопределение значения по умолчанию

Если необходимо явно указать версию C#, это можно сделать несколькими способами:

- Вручную изменить файл проекта.
- Задать версию языка для нескольких проектов в подкаталоге.
- Настроить параметр компилятора **LangVersion**.

> **TIP**
> Чтобы узнать, какую версию языка вы используете в данный момент, поставьте `#error version` (с учетом регистра) в коде. Это позволяет компилятору вывести ошибку CS8304 с сообщением, содержащим сведения об используемой версии компилятора и текущей выбранной версии языка. Дополнительные сведения см. в статье #error (справочник по C#).

Изменение файла проекта

Версию языка можно задать в файле проекта. Например, если доступ к предварительной версии функций должен быть задан явным образом, можно добавить следующий элемент:

```
<PropertyGroup>
   <LangVersion>preview</LangVersion>
</PropertyGroup>
```

Значение `preview` использует последнюю предварительную версию языка C#, которую поддерживает компилятор.

Настройка нескольких проектов

Чтобы настроить несколько проектов, вы можете создать файл **Directory.Build.props**, содержащий элемент `<LangVersion>`. Обычно это делается в каталоге решения. Добавьте следующий код в файл **Directory.Build.props** в каталоге решения:

```
<Project>
 <PropertyGroup>
   <LangVersion>preview</LangVersion>
 </PropertyGroup>
</Project>
```

Сборки во всех подкаталогах каталога, который содержит этот файл, будут использовать предварительную версию C#. Дополнительные сведения см. в статье Настройка сборки.

Справочник по версиям языка C#

В следующей таблице показаны все текущие версии языка C#. Ваш компилятор может не распознавать все значения, если имеет более раннюю версию. При установке .последней версии пакета SDK для .NET вы получаете доступ ко всем значениям в таблице.

ЗНАЧЕНИЕ	ЗНАЧЕНИЕ
`preview`	Компилятор допускает использование любого допустимого синтаксиса языка из последней предварительной версии.
`latest`	Компилятор принимает синтаксис из последней выпущенной версии компилятора (включая дополнительный номер версии).
`latestMajor` (`default`)	Компилятор принимает синтаксис из последней основной версии компилятора.
`10.0`	Компилятор принимает только синтаксис, включенный в спецификацию C# 10 или более ранних версий.
`9.0`	Компилятор принимает только синтаксис, включенный в спецификацию C# 9 или более ранних версий.
`8.0`	Компилятор принимает только синтаксис, включенный в спецификацию C# 8.0 или более ранней версии.
`7.3`	Компилятор принимает только синтаксис, включенный в спецификацию C# 7.3 или более ранней версии.
`7.2`	Компилятор принимает только синтаксис, включенный в спецификацию C# 7.2 или более ранней версии.
`7.1`	Компилятор принимает только синтаксис, включенный в спецификацию C# 7.1 или более ранней версии.
`7`	Компилятор принимает только синтаксис, включенный в спецификацию C# 7.0 или более ранней версии.
`6`	Компилятор принимает только синтаксис, включенный в спецификацию C# 6.0 или более ранней версии.
`5`	Компилятор принимает только синтаксис, включенный в спецификацию C# 5.0 или более ранней версии.

ЗНАЧЕНИЕ	ЗНАЧЕНИЕ
`4`	Компилятор принимает только синтаксис, включенный в спецификацию C# 4.0 или более ранней версии.
`3`	Компилятор принимает только синтаксис, включенный в спецификацию C# 3.0 или более ранней версии.
`ISO-2` (или `2`)	Компилятор принимает только синтаксис, включенный в спецификацию ISO/IEC 23270:2006 C# (2.0).
`ISO-1` (или `1`)	Компилятор принимает только синтаксис, включенный в спецификацию ISO/IEC 23270:2003 C# (1.0/1.2).

Типы значений (справочник по C#)

18.12.2021 • 3 minutes to read

Типы значений и ссылочные типы — это две основные категории типов C#. Переменная типа значения содержит экземпляр типа. Это отличается от переменной ссылочного типа, которая содержит ссылку на экземпляр типа. По умолчанию при назначении, передаче аргумента в метод и возврате результата метода копируются значения переменных. В случае переменных типа значения копируются соответствующие экземпляры типа. В следующем примере продемонстрировано такое поведение.

```
using System;

public struct MutablePoint
{
    public int X;
    public int Y;

    public MutablePoint(int x, int y) => (X, Y) = (x, y);

    public override string ToString() => $"({X}, {Y})";
}

public class Program
{
    public static void Main()
    {
        var p1 = new MutablePoint(1, 2);
        var p2 = p1;
        p2.Y = 200;
        Console.WriteLine($"{nameof(p1)} after {nameof(p2)} is modified: {p1}");
        Console.WriteLine($"{nameof(p2)}: {p2}");

        MutateAndDisplay(p2);
        Console.WriteLine($"{nameof(p2)} after passing to a method: {p2}");
    }

    private static void MutateAndDisplay(MutablePoint p)
    {
        p.X = 100;
        Console.WriteLine($"Point mutated in a method: {p}");
    }
}
// Expected output:
// p1 after p2 is modified: (1, 2)
// p2: (1, 200)
// Point mutated in a method: (100, 200)
// p2 after passing to a method: (1, 200)
```

Как показано в предыдущем примере, операции с переменной типа значения влияют только на этот экземпляр типа значения, хранящийся в переменной.

Если тип значения содержит элемент данных ссылочного типа, то при копировании экземпляра типа значения копируется только ссылка на экземпляр ссылочного типа. Как скопированный, так и исходный экземпляр типа значения имеют доступ к одному и тому же экземпляру ссылочного типа. В следующем примере продемонстрировано такое поведение.

```
using System;
using System.Collections.Generic;

public struct TaggedInteger
{
    public int Number;
    private List<string> tags;

    public TaggedInteger(int n)
    {
        Number = n;
        tags = new List<string>();
    }

    public void AddTag(string tag) => tags.Add(tag);

    public override string ToString() => $"{Number} [{string.Join(", ", tags)}]";
}

public class Program
{
    public static void Main()
    {
        var n1 = new TaggedInteger(0);
        n1.AddTag("A");
        Console.WriteLine(n1);  // output: 0 [A]

        var n2 = n1;
        n2.Number = 7;
        n2.AddTag("B");

        Console.WriteLine(n1);  // output: 0 [A, B]
        Console.WriteLine(n2);  // output: 7 [A, B]
    }
}
```

NOTE

Чтобы сделать код менее подверженным ошибкам и более надежным, определите и используйте неизменяемые типы значений. В этой статье изменяемые типы значений используются только в демонстрационных целях.

Виды типов значений и ограничения типов

Тип значения может относится к одному из двух следующих видов:

- тип структуры, который инкапсулирует данные и связанные функции;
- тип перечисления, который определяется набором именованных констант и представляет выбор или сочетание вариантов для выбора.

Тип значений `T?`, допускающий значение NULL, представляет все значения своего базового типа значения `T`, а также дополнительное значение NULL. Вы не можете назначить `null` переменной типа значения, если только это не тип, допускающий значение NULL.

Вы можете использовать ограничение `struct`, чтобы указать, что параметр типа является типом значения, не допускающим значения NULL. Типы структуры и перечисления удовлетворяют ограничению `struct`. Начиная с C# версии 7.3, вы можете использовать тип `System.Enum` в ограничении базового класса (которое известно как ограничение перечисления), чтобы указать, что параметр типа является типом перечисления.

Встроенные типы значений

C# предоставляет следующие встроенные типы значений, которые также называются *простыми типами*.

- Целочисленные типы
- Числовые типы с плавающей запятой
- bool, представляющий логическое значение
- char, представляющий символ Юникода UTF-16

Все простые типы являются типами структур и отличаются от других типов структур тем, что разрешают некоторые дополнительные операции.

- Литералы можно использовать для предоставления значения простого типа. Например, `'A'` — это литерал типа `char`, а `2001` — литерал типа `int`.
- Константы простых типов можно объявить с помощью ключевого слова const. Невозможно использовать константы других типов структур.
- Константные выражения, операнды которых являются константами простых типов, вычисляются во время компиляции.

Начиная с версии 7.0 в языке C# поддерживаются кортежи значений. Кортеж значений — это тип значения, не являющийся простым.

Спецификация языка C#

Дополнительные сведения см. в следующих разделах статьи Спецификация языка C#:

- Типы значений
- Простые типы
- Переменные

См. также раздел

- справочник по C#
- System.ValueType
- Ссылочные типы

Целочисленные типы (справочник по C#)

18.12.2021 • 3 minutes to read

Целочисленные типы представляют целые числа. Все целочисленные типы являются типами значений. Они также представляют собой простые типы и могут быть инициализированы литералами. Все целочисленные типы поддерживают арифметические операторы, побитовые логические операторы, операторы сравнения и равенства.

Характеристики целочисленных типов

C# поддерживает следующие предварительно определенные целочисленные типы:

КЛЮЧЕВОЕ СЛОВО ИЛИ ТИП C#	ДИАПАЗОН	РАЗМЕР	ТИП .NET
`sbyte`	От -128 до 127	8-разрядное целое число со знаком	System.SByte
`byte`	От 0 до 255	8-разрядное целое число без знака	System.Byte
`short`	От -32 768 до 32 767	16-разрядное целое число со знаком	System.Int16
`ushort`	От 0 до 65 535	16-разрядное целое число без знака	System.UInt16
`int`	От -2 147 483 648 до 2 147 483 647	32-разрядное целое число со знаком	System.Int32
`uint`	От 0 до 4 294 967 295	32-разрядное целое число без знака	System.UInt32
`long`	От -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807	64-разрядное целое число со знаком	System.Int64
`ulong`	От 0 до 18 446 744 073 709 551 615	64-разрядное целое число без знака	System.UInt64
`nint`	Зависит от платформы	32- или 64-разрядное целое число со знаком	System.IntPtr
`nuint`	Зависит от платформы	32- или 64-разрядное целое число без знака	System.UIntPtr

Во всех строках таблицы, кроме двух последних, каждое ключевое слово типа C# из крайнего левого столбца является псевдонимом для соответствующего типа .NET. Ключевое слово и имя типа .NET являются взаимозаменяемыми. Например, следующие объявления объявляют переменные одного типа:

```
int a = 123;
System.Int32 b = 123;
```

Типы `nint` и `nuint` в последних двух строках таблицы являются целыми числами собственного размера. В коде такие числа представлены определенными типами .NET, но в каждом случае ключевое слово и тип .NET не взаимозаменяемы. Для `nint` и `nuint` компилятор предоставляет преобразования и операции, как для целочисленных типов. Они отличаются от преобразований и операций для типов указателей `System.IntPtr` и `System.UIntPtr`. Дополнительные сведения см. в статье о типах `nint` и `nuint`.

По умолчанию все целочисленные типы имеют значение `0`. Все целочисленные типы, кроме целых чисел собственного размера, имеют константы `MinValue` и `MaxValue` с минимальным и максимальным значением этого типа.

Используйте структуру System.Numerics.BigInteger, чтобы представить целое число со знаком без верхней и нижней границ.

Целочисленные литералы

Целочисленные литералы могут быть:

- *десятичным числом*: без префикса;
- *шестнадцатеричным числом*: с префиксом `0x` или `0X`;
- *двоичными*: с префиксом `0b` или `0B` (доступно в C# 7.0 и более поздних версиях).

В приведенном ниже коде показан пример каждого из них.

```
var decimalLiteral = 42;
var hexLiteral = 0x2A;
var binaryLiteral = 0b_0010_1010;
```

В предыдущем примере также показано использование `_` в качестве *цифрового разделителя*, который поддерживается, начиная с версии C# 7.0. Цифровой разделитель можно использовать со всеми видами числовых литералов.

Тип целочисленного литерала определяется его суффиксом следующим образом:

- Если литерал не имеет суффикса, его типом будет первый из следующих типов, в котором может быть представлено его значение: `int`, `uint`, `long`, `ulong`.

  > **NOTE**
  > Литералы интерпретируется как положительные значения. Например, литерал `0xFF_FF_FF_FF` представляет число `4294967295` типа `uint`, хотя он имеет то же битовое представление, что и число `-1` типа `int`. Если вам требуется значение определенного типа, приведите литерал к этому типу. Используйте оператор `unchecked`, если представить значение литерала в целевом типе невозможно. Например, `unchecked((int)0xFF_FF_FF_FF)` выдает `-1`.

- Если у литерала есть суффикс `U` или `u`, его типом будет первый из следующих типов, в котором может быть представлено его значение: `uint`, `ulong`.

- Если у литерала есть суффикс `L` или `l`, его типом будет первый из следующих типов, в котором может быть представлено его значение: `long`, `ulong`.

> **NOTE**
> Строчную букву `l` можно использовать в качестве суффикса. Однако при этом выдается предупреждение компилятора, так как букву `l` можно перепутать с цифрой `1`. Для ясности используйте `L`.

- Если у литерала есть суффикс `UL`, `Ul`, `uL`, `ul`, `LU`, `Lu`, `lU` или `lu`, его тип — `ulong`.

Если значение, представленное целочисленным литералом, превышает UInt64.MaxValue, происходит ошибка компиляции CS1021.

Если определенный тип целочисленного литерала — `int`, а значение, представленное литералом, находится в диапазоне целевого типа, значение можно неявно преобразовать в `sbyte`, `byte`, `short`, `ushort`, `uint`, `ulong`, `nint` или `nuint`:

```
byte a = 17;
byte b = 300;   // CS0031: Constant value '300' cannot be converted to a 'byte'
```

Как показано в предыдущем примере, если значение литерала выходит за пределы диапазона целевого типа, возникает ошибка компилятора CS0031.

Можно также использовать приведение для преобразования значения, представленного целочисленным литералом, в тип, отличный от определенного типа литерала:

```
var signedByte = (sbyte)42;
var longVariable = (long)42;
```

Преобразования

Любой целочисленный тип можно преобразовать в любой другой целочисленный тип. Если целевой тип может хранить все значения исходного типа, преобразование является неявным. В противном случае необходимо использовать выражение приведения для выполнения явного преобразования. Для получения дополнительной информации см. статью Встроенные числовые преобразования.

Спецификация языка C#

Дополнительные сведения см. в следующих разделах статьи Спецификация языка C#:

- Целочисленные типы
- Целочисленные литералы

См. также

- справочник по C#
- Типы значений
- Типы с плавающей запятой
- Строки стандартных числовых форматов
- Числовые значения в .NET

Типы `nint` и `nuint` (справочник по C#)

18.12.2021 • 2 minutes to read

Начиная с C# 9.0, вы можете использовать ключевые слова `nint` и `nuint` для определения *целых чисел собственного размера*. Эти целые числа будут 32-разрядными при использовании в 32-битных процессах и 64-разрядными при использовании в 64-битных процессах. Их можно использовать для сценариев взаимодействия, с низкоуровневыми библиотеками и для оптимизации производительности в сценариях, где часто выполняются математические операции с целыми числами.

Целочисленные типы собственного размера представляются внутренне как типы .NET System.IntPtr и System.UIntPtr. В отличие от других числовых типов, эти ключевые слова — не просто псевдонимы для типов. Следующие инструкции не являются эквивалентными:

```
nint a = 1;
System.IntPtr a = 1;
```

Для `nint` и `nuint` компилятор предоставляет такие же преобразования и операции, как для всех целочисленных типов.

Собственный размер целого числа во время выполнения

Чтобы узнать размер целого числа собственного размера во время выполнения, используйте `sizeof()`. Но код должен компилироваться в контексте unsafe. Пример:

```
Console.WriteLine($"size of nint = {sizeof(nint)}");
Console.WriteLine($"size of nuint = {sizeof(nuint)}");

// output when run in a 64-bit process
//size of nint = 8
//size of nuint = 8

// output when run in a 32-bit process
//size of nint = 4
//size of nuint = 4
```

Такое же значение можно получить с помощью статических свойств IntPtr.Size и UIntPtr.Size.

MinValue и MaxValue

Чтобы получить минимальное и максимальное значения целых чисел собственного размера во время выполнения, используйте `MinValue` и `MaxValue` в качестве статических свойств с ключевыми словами `nint` и `nuint`, как показано в следующем примере:

```
Console.WriteLine($"nint.MinValue = {nint.MinValue}");
Console.WriteLine($"nint.MaxValue = {nint.MaxValue}");
Console.WriteLine($"nuint.MinValue = {nuint.MinValue}");
Console.WriteLine($"nuint.MaxValue = {nuint.MaxValue}");

// output when run in a 64-bit process
//nint.MinValue = -9223372036854775808
//nint.MaxValue = 9223372036854775807
//nuint.MinValue = 0
//nuint.MaxValue = 18446744073709551615

// output when run in a 32-bit process
//nint.MinValue = -2147483648
//nint.MaxValue = 2147483647
//nuint.MinValue = 0
//nuint.MaxValue = 4294967295
```

Константы

Значения констант можно использовать в таких диапазонах:

- для `nint`: Int32.MinValue–Int32.MaxValue;
- для `nuint`: UInt32.MinValue–UInt32.MaxValue.

Преобразования

Компилятор предоставляет неявные и явные преобразования в другие числовые типы. Для получения дополнительной информации см. статью Встроенные числовые преобразования.

Литералы

Для целочисленных литералов собственного размера не поддерживается прямой синтаксис. Суффикса, указывающего на то, что литерал является целым числом собственного размера (например, `L` для обозначения `long`), нет. Вместо этого можно использовать явные или неявные приведения других целочисленных значений. Пример:

```
nint a = 42
nint a = (nint)42;
```

Неподдерживаемые элементы IntPtr и UIntPtr

Следующие элементы структур IntPtr и UIntPtr не поддерживаются для типов `nint` и `nuint`:

- Параметризованные конструкторы.
- Add(IntPtr, Int32)
- CompareTo
- Size — вместо этого используйте sizeof(). Хотя `nint.Size` не поддерживается, можно получить эквивалентное значение с помощью `IntPtr.Size`.
- Subtract(IntPtr, Int32)
- ToInt32
- ToInt64
- ToPointer
- Zero. Вместо этого используйте 0.

Спецификация языка C#

Дополнительные сведения см. в статье о спецификации языка C# и в статье Целые числа собственного размера в предложениях по функциям C# 9.0.

См. также раздел

- справочник по C#
- Типы значений
- Целочисленные типы
- Встроенные числовые преобразования

Числовые типы с плавающей запятой (справочник по C#)

18.12.2021 • 3 minutes to read

Числовые типы с плавающей запятой представляют действительные числа. Все числовые типы с плавающей запятой являются типами значений. Они также представляют собой простые типы и могут быть инициализированы литералами. Все числовые типы с плавающей запятой поддерживают арифметические операторы, а также операторы сравнения и равенства.

Характеристики типов с плавающей запятой

C# поддерживает следующие предварительно определенные типы с плавающей запятой:

КЛЮЧЕВОЕ СЛОВО ИЛИ ТИП C#	ПРИБЛИЗИТЕЛЬНЫЙ ДИАПАЗОН ЗНАЧЕНИЙ	ТОЧНОСТЬ	РАЗМЕР	ТИП .NET
`float`	От $\pm 1{,}5 \times 10^{-45}$ до $\pm 3{,}4 \times 10^{38}$	6–9 цифр	4 байта	System.Single
`double`	от $\pm 5{,}0 \times 10^{-324}$ до $\pm 1{,}7 \times 10^{308}$	15–17 цифр	8 байт	System.Double
`decimal`	от $\pm 1{,}0 \times 10^{-28}$ до $\pm 7{,}9228 \times 10^{28}$	28-29 знаков	16 байт	System.Decimal

В приведенной выше таблице каждый тип ключевого слова C# из крайнего левого столбца является псевдонимом для соответствующего типа .NET. Они взаимозаменяемые. Например, следующие объявления объявляют переменные одного типа:

```
double a = 12.3;
System.Double b = 12.3;
```

По умолчанию все типы с плавающей запятой имеют значение `0`. Все типы с плавающей запятой имеют константы `MinValue` и `MaxValue` с минимальным и максимальными итоговыми значениями этого типа. Типы `float` и `double` также предоставляют константы, обозначающие бесконечные и нечисловые значения. Например, тип `double` предоставляет следующие константы: Double.NaN, Double.NegativeInfinity и Double.PositiveInfinity.

Тип `decimal` подходит, если требуемая степень точности определяется числом цифр справа от десятичной запятой. Такие числа обычно используются в финансовых приложениях для денежных сумм (например, 1,00 долл. США), процентных ставок (например, 2,625 %) и т. д. Даже числа, точные только до одной десятичной цифры, точнее обрабатываются типом `decimal`: 0,1, например, можно в точности представить экземпляром `decimal`. При этом не существует экземпляра `double` или `float`, который точно представляет 0,1. Из-за этой разницы в числовых типах в арифметических вычислениях могут возникать непредвиденные ошибки округления при использовании `double` или `float` для десятичных данных. Вы можете использовать `double` вместо `decimal`, если оптимизация производительности важнее, чем обеспечение точности. Но любая разница в производительности останется незамеченной для всех приложений, кроме самых требовательных к вычислениям. Еще одна возможная причина, по

которой следует избегать `decimal`, — это минимальные требования к хранилищу. Например, ML.NET использует `float`, так как разница между 4 байтами и 16 байтами суммируется для очень больших наборов данных. Для получения дополнительной информации см. System.Decimal.

В одном и том же выражении можно сочетать и целочисленные типы, и типы `float` и `double`. В этом случае целочисленные типы неявно преобразуются в один из типов с плавающей запятой. При необходимости тип `float` неявно преобразуется в `double`. Выражение вычисляется следующим образом.

- Если в выражении есть тип `double`, оно оценивается как `double` или `bool` в реляционных сравнениях или сравнениях на равенство.
- Если в выражении нет типа `double`, оно оценивается как `float` или `bool` в реляционных сравнениях или сравнениях на равенство.

Можно также смешивать целочисленные типы и тип `decimal` в выражении. В этом случае целочисленные типы неявно преобразуются в тип `decimal`, а выражение вычисляется как `decimal` или `bool` в реляционных сравнениях и сравнениях на равенство.

Тип `decimal` нельзя смешивать с типами `float` и `double` в выражении. В этом случае, если требуется выполнить арифметические операции или операции сравнения или равенства, необходимо явно преобразовать операнды из типа или в тип `decimal`, как показано в следующем примере:

```
double a = 1.0;
decimal b = 2.1m;
Console.WriteLine(a + (double)b);
Console.WriteLine((decimal)a + b);
```

Можно использовать строки стандартных числовых форматов или строки пользовательских числовых форматов для форматирования значения с плавающей запятой.

Вещественные литералы

Тип реального литерала определяется его суффиксом следующим образом:

- Литерал без суффикса или с суффиксом `d` или `D` имеет тип `double`.
- Литерал с суффиксом `f` или `F` имеет тип `float`.
- Литерал с суффиксом `m` или `M` имеет тип `decimal`.

В приведенном ниже коде показан пример каждого из них.

```
double d = 3D;
d = 4d;
d = 3.934_001;

float f = 3_000.5F;
f = 5.4f;

decimal myMoney = 3_000.5m;
myMoney = 400.75M;
```

В предыдущем примере также показано использование `_` в качестве *цифрового разделителя*, который поддерживается, начиная с версии C# 7.0. Цифровой разделитель можно использовать со всеми видами числовых литералов.

Можно также использовать экспоненциальное представление, то есть указать экспоненту вещественного литерала, как показано в следующем примере:

```
double d = 0.42e2;
Console.WriteLine(d);  // output 42

float f = 134.45E-2f;
Console.WriteLine(f);  // output: 1.3445

decimal m = 1.5E6m;
Console.WriteLine(m);  // output: 1500000
```

Преобразования

Существует только одно неявное преобразование между числовыми типами с плавающей запятой: из `float` в `double`. Однако можно преобразовать любой тип с плавающей запятой в любой другой тип с плавающей запятой с помощьюявного приведения. Для получения дополнительной информации см. статью Встроенные числовые преобразования.

Спецификация языка C#

Дополнительные сведения см. в следующих разделах статьи Спецификация языка C#:

- Типы с плавающей запятой
- Тип decimal
- Вещественные литералы

См. также

- справочник по C#
- Типы значений
- Целочисленные типы
- Строки стандартных числовых форматов
- Числовые значения в .NET
- System.Numerics.Complex

Встроенные числовые преобразования (справочник по C#)

18.12.2021 • 3 minutes to read

C# предоставляет набор целочисленных типов и типы с плавающей запятой. Существует преобразование между любыми двумя числовыми типами: неявное или явное. Для выполнения явного преобразования необходимо использовать выражение приведения.

Неявные числовые преобразования

В следующей таблице приведены предопределенные неявные преобразования между встроенными числовыми типами:

исходный тип	**кому**
sbyte	`short`, `int`, `long`, `float`, `double`, `decimal` или `nint`
byte	`short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, `decimal`, `nint` или `nuint`
short	`int`, `long`, `float`, `double` или `decimal` либо `nint`
ushort	`int`, `uint`, `long`, `ulong`, `float`, `double` или `decimal`, `nint` или `nuint`
int	`long`, `float`, `double` или `decimal`, `nint`
uint	`long`, `ulong`, `float`, `double` или `decimal` либо `nuint`
long	`float`, `double` или `decimal`
ulong	`float`, `double` или `decimal`
float	`double`
nint	`long`, `float`, `double` или `decimal`
nuint	`ulong`, `float`, `double` или `decimal`

NOTE

Неявные преобразования из `int`, `uint`, `long`, `ulong`, `nint` или `nuint` в `float` и из `long`, `ulong`, `nint` или `nuint` в `double` могут приводить к потере точности, но не к потере порядка величин. Другие неявные числовые преобразования никогда не теряют никаких сведений.

Также обратите внимание на следующее.

- Любой целочисленный тип неявно преобразуется в любой числовой тип с плавающей запятой.
- Не поддерживается неявное преобразование в типы `byte` и `sbyte`. Не поддерживается неявное преобразование из типов `double` и `decimal`.
- Не поддерживается неявное преобразование между типом `decimal` и типами `float` или `double`.
- Значение константного выражения типа `int` (например, значение, представленное целочисленным литералом) может быть неявно преобразовано в `sbyte`, `byte`, `short`, `ushort`, `uint`, `ulong`, `nint` или `nuint`, если оно находится в диапазоне целевого типа:

  ```
  byte a = 13;
  byte b = 300;  // CS0031: Constant value '300' cannot be converted to a 'byte'
  ```

  Как показано в предыдущем примере, если значение константы выходит за пределы диапазона целевого типа, возникает ошибка компилятора CS0031.

Явные числовые преобразования

В следующей таблице показаны предопределенные явные преобразования между встроенными числовыми типами, для которых нет неявного преобразования:

ИСХОДНЫЙ ТИП	КОМУ
sbyte	`byte`, `ushort`, `uint` или `ulong` либо `nuint`
byte	`sbyte`
short	`sbyte`, `byte`, `ushort`, `uint`, `ulong` или `nuint`
ushort	`sbyte`, `byte` или `short`
int	`sbyte`, `byte`, `short`, `ushort`, `uint`, `ulong` или `nuint`
uint	`sbyte`, `byte`, `short`, `ushort` или `int`
long	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `ulong`, `nint` или `nuint`
ulong	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `nint` или `nuint`
float	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `decimal`, `nint` или `nuint`
double	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `decimal`, `nint` или `nuint`
decimal	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, `nint` или `nuint`

ИСХОДНЫЙ ТИП	КОМУ
nint	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `ulong` или `nuint`
nuint	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long` или `nint`

> **NOTE**
> Явное числовое преобразование может привести к утрате данных или созданию исключения, обычно OverflowException.

Также обратите внимание на следующее.

- При преобразовании значения целочисленного типа в другой целочисленный тип результат зависит от переполнения контекста проверки. В проверенном контексте преобразование выполняется успешно, если исходное значение находится в диапазоне конечного типа. В противном случае возникает исключение OverflowException. В непроверяемом контексте преобразование всегда завершается успешно и выполняется следующим образом.
  - Если исходный тип больше целевого, исходное значение усекается путем отбрасывания его "лишних" самых значимых битов. Результат затем обрабатывается как значение целевого типа.
  - Если исходный тип меньше целевого, исходное значение дополняется знаками или нулями, чтобы иметь тот же размер, что и целевой тип. Знаки добавляются, если исходный тип имеет знак. Если у исходного типа нет знака, добавляются нули. Результат затем обрабатывается как значение целевого типа.
  - Если исходный тип совпадает по размеру с целевым, исходное значение обрабатывается как значение целевого типа.
- При преобразовании значения `decimal` в целочисленный тип оно округляется в сторону нуля до ближайшего целого значения. Если итоговое целое значение находится вне диапазона целевого типа, возникает исключение OverflowException.
- При преобразовании значения `double` или `float` в целочисленный тип оно округляется в сторону нуля до ближайшего целого значения. Если полученное целое значение выходит за пределы диапазона целевого типа, результат будет зависеть от контекста проверки переполнения. В проверенном контексте возникает исключение OverflowException, а в непроверенном контексте результатом будет неопределенное значение целевого типа.
- При преобразовании из `double` в `float` значение `double` округляется до ближайшего значения `float`. Если значение `double` слишком мало или слишком велико для типа `float`, результатом будет ноль или бесконечность соответственно.
- При преобразовании из `float` или `double` в `decimal` исходное значение преобразуется в представление `decimal` и при необходимости округляется до ближайшего числа после 28-го десятичного разряда. В зависимости от исходного значения возможны следующие результаты:
  - Если исходное значение слишком мало для представления в виде `decimal`, результатом будет ноль.
  - Если исходное значение не является числом (NaN), равно бесконечности или слишком

велико для представления в виде `decimal`, возникает исключение OverflowException.

- При преобразовании из `decimal` в `float` или `double` исходное значение округляется до ближайшего значения `float` или `double` соответственно.

Спецификация языка C#

Дополнительные сведения см. в следующих разделах статьи Спецификация языка C#:

- Неявные числовые преобразования
- Явные числовые преобразования

См. также

- справочник по C#
- Приведение и преобразование типов

bool (справочник по C#)

18.12.2021 • 2 minutes to read

Ключевое слово типа `bool` — это псевдоним для типа структуры System.Boolean .NET, представляющий логическое значение: `true` или `false`.

Для выполнения логических операций со значениями типа `bool` используйте логические операторы. Тип `bool` является типом результата операторов сравнения и равенства. Выражение `bool` может быть управляющим условным выражением в операторах if, do, while и for и условном операторе `?:`.

Значение по умолчанию для типа `bool` — `false`.

Литералы

Вы можете использовать литералы `true` и `false` для инициализации переменной `bool` или передачи значения `bool`:

```
bool check = true;
Console.WriteLine(check ? "Checked" : "Not checked");  // output: Checked

Console.WriteLine(false ? "Checked" : "Not checked");  // output: Not checked
```

Трехзначная булева логика

Используйте тип `bool?`, допускающий значение NULL, если нужно использовать трехзначную логику, например, при работе с базами данных, которые поддерживают трехзначный логический тип. Для операндов `bool?` предопределенные операторы `&` и `|` поддерживают троичную логику. См. подробнее о логических операторах, поддерживающих значение NULL в описании логических операторов.

См. дополнительные сведения о типах значений, допускающих значение NULL.

Преобразования

В C# доступно только два преобразования, использующих тип `bool`. Это неявное преобразование в соответствующий тип `bool?`, допускающий значение NULL, и явное преобразование из типа `bool?`. Однако .NET предоставляет дополнительные методы, позволяющие выполнять преобразование в тип `bool` или из него. Дополнительные сведения см. в разделе о преобразовании в логические значения и из них справочника по API System.Boolean.

Спецификация языка C#

Дополнительные сведения см. в разделе Тип boolспецификации языка C#.

См. также раздел

- справочник по C#
- Типы значений
- Операторы true и false

char (Справочник по C#)

18.12.2021 • 2 minutes to read

Ключевое слово типа `char` — это псевдоним для типа структуры System.Char .NET, представляющий символ UTF-16 в Юникоде.

TYPE	ДИАПАЗОН	РАЗМЕР	ТИП .NET
`char`	От U+0000 до U+FFFF	16 разрядов	System.Char

Значение по умолчанию для типа `char` — `\0`, то есть U+0000.

Тип `char` поддерживает сравнение, проверку равенства, а также операции инкремента и декремента. Кроме того, для операндов `char` арифметические и побитовые логические операторы выполняют операцию с соответствующими кодами символов и создают результат типа `int`.

Тип string представляет текст как последовательность значений `char`.

Литералы

Значение `char` можно указать следующим образом:

- символьный литерал;
- escape-последовательность Юникода, то есть символы `\u`, за которыми следует шестнадцатеричное представление кода символа из четырех символов;
- шестнадцатеричная escape-последовательность, то есть символы `\x`, за которыми следует шестнадцатеричное представление кода символа.

```
var chars = new[]
{
    'j',
    '\u006A',
    '\x006A',
    (char)106,
};
Console.WriteLine(string.Join(" ", chars));  // output: j j j j
```

Как показано в предыдущем примере, можно также привести значение кода символа к соответствующему значению `char`.

> **NOTE**
>
> В случае escape-последовательности Юникода необходимо указать все четыре шестнадцатеричные цифры. То есть `\u006A` — допустимая escape-последовательность, а `\u06A` и `\u6A` нет.
>
> В случае шестнадцатеричной escape-последовательности начальные нули можно опустить. То есть `\x006A`, `\x06A` и `\x6A` — допустимые escape-последовательности, соответствующие одному символу.

Преобразования

Тип `char` неявно преобразуется в следующие целочисленные типы: `ushort`, `int`, `uint`, `long` и `ulong`.

Он также может быть неявно преобразован во встроенные числовые типы с плавающей запятой: `float`, `double` и `decimal`. Он явно преобразуется в целочисленные типы `sbyte`, `byte` и `short`.

Неявные преобразования из других типов в тип `char` не предусмотрены. Но любой целочисленный тип или числовой тип с плавающей запятой явно преобразуется в `char`.

Спецификация языка C#

Дополнительные сведения см. в разделе Целочисленные типы в статье Спецификации языка C#.

См. также

- справочник по C#
- Типы значений
- Строки
- System.Text.Rune
- Кодировка символов в .NET

Типы перечислений (справочник по C#)

18.12.2021 • 3 minutes to read

Тип перечисления (или *тип enum*) — это тип значения, определенный набором именованных констант применяемого целочисленного типа. Чтобы определить тип перечисления, используйте ключевое слово `enum` и укажите имена *элементов перечисления*.

```
enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}
```

По умолчанию связанные значения констант элементов перечисления имеют тип `int`. Они начинаются с нуля и увеличиваются на единицу в соответствии с порядком текста определения. Вы можете явно указать любой другой целочисленный тип в качестве базового типа перечисления. Вы можете также явно указать соответствующие значения констант, как показано в следующем примере.

```
enum ErrorCode : ushort
{
    None = 0,
    Unknown = 1,
    ConnectionLost = 100,
    OutlierReading = 200
}
```

Вы не можете определить метод внутри определения типа перечисления. Чтобы добавить функциональные возможности к типу перечисления, создайте метод расширения.

Значением по умолчанию для типа перечисления `E` является значение, созданное выражением `(E)0`, даже если ноль не имеет соответствующего элемента перечисления.

Тип перечисления используется для представления выбора из набора взаимоисключающих значений или комбинации вариантов выбора. Чтобы представить комбинацию вариантов выбора, определите тип перечисления как битовые флаги.

Типы перечислений как битовые флаги

Если вам необходимо, чтобы тип перечисления представлял комбинацию вариантов выбора, определите элементы перечисления для этих вариантов так, чтобы отдельный выбор был битовым полем. То есть связанные значения этих элементов перечисления должны быть степенями двух. Затем вы можете использовать побитовые логические операторы `|` или `&`, чтобы комбинировать варианты выбора или пересекать комбинации вариантов выбора соответственно. Чтобы указать, что тип перечисления объявляет битовые поля, примените к нему атрибут Flags. Как показано в следующем примере, вы можете включить некоторые типичные комбинации в определение типа перечисления.

```
[Flags]
public enum Days
{
    None      = 0b_0000_0000,  // 0
    Monday    = 0b_0000_0001,  // 1
    Tuesday   = 0b_0000_0010,  // 2
    Wednesday = 0b_0000_0100,  // 4
    Thursday  = 0b_0000_1000,  // 8
    Friday    = 0b_0001_0000,  // 16
    Saturday  = 0b_0010_0000,  // 32
    Sunday    = 0b_0100_0000,  // 64
    Weekend   = Saturday | Sunday
}

public class FlagsEnumExample
{
    public static void Main()
    {
        Days meetingDays = Days.Monday | Days.Wednesday | Days.Friday;
        Console.WriteLine(meetingDays);
        // Output:
        // Monday, Wednesday, Friday

        Days workingFromHomeDays = Days.Thursday | Days.Friday;
        Console.WriteLine($"Join a meeting by phone on {meetingDays & workingFromHomeDays}");
        // Output:
        // Join a meeting by phone on Friday

        bool isMeetingOnTuesday = (meetingDays & Days.Tuesday) == Days.Tuesday;
        Console.WriteLine($"Is there a meeting on Tuesday: {isMeetingOnTuesday}");
        // Output:
        // Is there a meeting on Tuesday: False

        var a = (Days)37;
        Console.WriteLine(a);
        // Output:
        // Monday, Wednesday, Saturday
    }
}
```

Дополнительные сведения и примеры см. на странице справочника по API System.FlagsAttribute и в разделе о неисключительных элементах и атрибутах Flags страницы справочника по API System.Enum.

Тип System.Enum и ограничение перечисления

Тип System.Enum является абстрактным базовым классом всех типов перечисления. Он предоставляет различные методы, позволяющие получить информацию о типе перечисления и его значениях. Дополнительные сведения и примеры см. на странице справочника по API System.Enum.

Начиная с C# версии 7.3, вы можете использовать тип `System.Enum` в ограничении базового класса (которое известно как ограничение перечисления), чтобы указать, что параметр типа является типом перечисления. Любой тип перечисления также удовлетворяет ограничению `struct`, которое используется для указания того, что параметр типа является типом значения, не допускающим значения NULL.

Преобразования

Для любого типа перечисления существуют явные преобразования между типом перечисления и его базовым целочисленным типом. Если вы привели значение перечисления к его базовому типу, то результатом будет связанное целочисленное значение элемента перечисления.

```
public enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}

public class EnumConversionExample
{
    public static void Main()
    {
        Season a = Season.Autumn;
        Console.WriteLine($"Integral value of {a} is {(int)a}");  // output: Integral value of Autumn is 2

        var b = (Season)1;
        Console.WriteLine(b);  // output: Summer

        var c = (Season)4;
        Console.WriteLine(c);  // output: 4
    }
}
```

Используйте метод Enum.IsDefined, чтобы определить, содержит ли тип перечисления элемент перечисления с определенным связанным значением.

Для любого типа перечисления существует упаковка — преобразование и распаковка — преобразование в тип System.Enum и из него соответственно.

Спецификация языка C#

Дополнительные сведения см. в следующих разделах статьи Спецификация языка C#:

- Перечисления
- Значения перечислений и операции с ними
- Enumeration logical operators (Логические операторы перечисления)
- Enumeration comparison operators (Операторы сравнения перечислений)
- Explicit enumeration conversions (Явные преобразования перечислений)
- Implicit enumeration conversions (Неявные преобразования перечислений)

См. также

- справочник по C#
- Строки форматов перечисления
- Рекомендации по разработке перечислений
- Рекомендации по разработке. Соглашения об именовании перечислений
- Выражение `switch`
- Инструкция `switch`

Типы структур (справочник по C#)

18.12.2021 • 9 minutes to read

Тип структуры представляет собой тип значения, который может инкапсулировать данные и связанные функции. Для определения типа структуры используется ключевое слово `struct`:

```
public struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; }
    public double Y { get; }

    public override string ToString() => $"({X}, {Y})";
}
```

Типы структуры имеют *семантики значений*. То есть переменная типа структуры содержит экземпляр этого типа. По умолчанию значения переменных копируются при назначении, передаче аргумента в метод и возврате результата метода. В случае переменной типа структуры копируется экземпляр типа. Дополнительные сведения см. в разделе Типы значений.

Как правило, типы структуры используются для проектирования небольших ориентированных на данные типов, которые предоставляют минимум поведения или не предоставляют его вовсе. Например, платформа .NET использует типы структуры для представления числа (как целого, так и вещественного), логического значения, символа Юникода, экземпляра времени. Если вы сконцентрированы на поведении типа, рекомендуется определить класс. Типы классов имеют *семантики ссылок*. То есть переменная типа класса содержит ссылку на экземпляр этого типа, а не сам экземпляр.

Поскольку типы структуры имеют семантику значений, рекомендуется определять *неизменяемые* типы структуры.

Структура `readonly`

Начиная с C# версии 7.2, чтобы объявить, что тип структуры является неизменяемым, используйте модификатор `readonly`. Все элементы данных структуры `readonly` должны быть доступны только для чтения:

- Любое объявление поля должно иметь `readonly` модификатор.
- Все свойства, включая автоматические реализованные, должны быть доступны только для чтения. В C# 9.0 и более поздних версий свойство может иметь метод доступа `init`.

Это гарантирует, что ни один из элементов структуры `readonly` не изменит состояние структуры. В C# 8.0 и более поздних версиях это означает, что другие члены экземпляра, кроме конструкторов, неявно `readonly`.

NOTE

В структуре `readonly` элемент данных изменяемого ссылочного типа по-прежнему может изменять свое собственное состояние. Например, вы не можете заменить экземпляр List<T>, но можете добавить в него новые элементы.

В следующем коде определяется структура `readonly` с методами задания свойств только для инициализации, которые доступны в C# 9.0 и более поздних версий:

```
public readonly struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; init; }
    public double Y { get; init; }

    public override string ToString() => $"({X}, {Y})";
}
```

Члены экземпляров `readonly`

Начиная с C# 8.0, можно также использовать модификатор `readonly`, чтобы объявить, что член экземпляра не изменяет состояние структуры. Если не удается объявить весь тип структуры как `readonly`, используйте модификатор `readonly`, чтобы пометить члены экземпляров, которые не изменяют состояние структуры.

В члене экземпляра `readonly` невозможно назначать поля экземпляра структуры. Однако член `readonly` может вызвать член, не являющийся `readonly`. В этом случае компилятор создает копию экземпляра структуры и вызывает в ней член, не являющийся `readonly`. В результате исходный экземпляр структуры остается без изменений.

Как правило, модификатор `readonly` применяется к следующим типам элементов экземпляров.

- Методы.

  ```
  public readonly double Sum()
  {
      return X + Y;
  }
  ```

  Можно также применить модификатор `readonly` к методам, переопределяющим методы, объявленные в System.Object.

  ```
  public readonly override string ToString() => $"({X}, {Y})";
  ```

- Свойства и индексаторы.

```
private int counter;
public int Counter
{
    readonly get => counter;
    set => counter = value;
}
```

Если необходимо применить модификатор `readonly` к методам доступа свойства или индексатора, примените его в объявлении свойства или индексатора.

> **NOTE**
> Компилятор объявляет метод доступа `get` автоматически реализуемого свойства как `readonly` независимо от наличия модификатора `readonly` в объявлении свойства.

В C# 9.0 и более поздних версий вы можете применить модификатор `readonly` к свойству или индексатору с помощью метода доступа `init`:

```
public readonly double X { get; init; }
```

Модификатор `readonly` не применяется к статическим членам типа структуры.

Компилятор может использовать модификатор `readonly` для оптимизации производительности. Дополнительные сведения см. в статье Написание безопасного и эффективного кода C#.

Обратимое изменение

Начиная с C# 10 можно использовать выражение `with` для создания копии экземпляра с типом структуры, в котором изменяются указанные свойства и поля. Как показано в следующем примере, для указания элементов для изменения и их новых значений используется синтаксис инициализатора объектов.

```
public readonly struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; init; }
    public double Y { get; init; }

    public override string ToString() => $"({X}, {Y})";
}

public static void Main()
{
    var p1 = new Coords(0, 0);
    Console.WriteLine(p1);  // output: (0, 0)

    var p2 = p1 with { X = 3 };
    Console.WriteLine(p2);  // output: (3, 0)

    var p3 = p1 with { X = 1, Y = 4 };
    Console.WriteLine(p3);  // output: (1, 4)
}
```

Ограничения при проектировании типа структуры

При проектировании типа структуры вы будете располагать теми же возможностями, что и для типа класса, за исключением следующих аспектов:

- Вы не можете объявить конструктор без параметров. Каждый тип структуры уже предоставляет неявный конструктор без параметров, который создает значение по умолчанию типа.

  > **NOTE**
  > Начиная с C# версии 10 вы можете объявить конструктор без параметров в типе структуры. Дополнительные сведения см. в разделе Конструкторы без параметров и инициализаторы полей.

- Вы не можете инициализировать поле экземпляра или свойство при его объявлении. Однако можно инициализировать статическое поле или поле константы или статическое свойство при его объявлении.

  > **NOTE**
  > Начиная с C# версии 10 вы можете инициализировать поле экземпляра или свойство при его объявлении. Дополнительные сведения см. в разделе Конструкторы без параметров и инициализаторы полей.

- Конструктор типа структуры должен инициализировать все поля экземпляров данного типа.
- Тип структуры не может наследовать от другого типа класса или структуры и не может быть базовым для класса. Однако тип структуры может реализовывать интерфейсы.
- Вы не можете объявить метод завершения в типе структуры.

Конструкторы без параметров и инициализаторы полей

Начиная с C# 10 вы можете объявить конструктор экземпляра без параметров в типе структуры, как показано в следующем примере:

```
public readonly struct Measurement
{
    public Measurement()
    {
        Value = double.NaN;
        Description = "Undefined";
    }

    public Measurement(double value, string description)
    {
        Value = value;
        Description = description;
    }

    public double Value { get; init; }
    public string Description { get; init; }

    public override string ToString() => $"{Value} ({Description})";
}

public static void Main()
{
    var m1 = new Measurement();
    Console.WriteLine(m1);  // output: NaN (Undefined)

    var m2 = default(Measurement);
    Console.WriteLine(m2);  // output: 0 ()

    var ms = new Measurement[2];
    Console.WriteLine(string.Join(", ", ms));  // output: 0 (), 0 ()
}
```

Как видно в примере выше, выражение значения по умолчанию игнорирует конструктор без параметров и создает для типа структуры значение по умолчанию, полученное путем присвоения значений по умолчанию всем полям типа значения (шаблон с нулевыми битами) и значения `null` всем полям ссылочного типа. При создании экземпляра массива типа структуры также игнорируется конструктор без параметров и создается массив, заполненный значениями по умолчанию для типа структуры.

Начиная с C# 10 вы можете также инициализировать свойство или поле экземпляра в его объявлении, как показано в следующем примере:

```
public readonly struct Measurement
{
    public Measurement(double value)
    {
        Value = value;
    }

    public Measurement(double value, string description)
    {
        Value = value;
        Description = description;
    }

    public double Value { get; init; }
    public string Description { get; init; } = "Ordinary measurement";

    public override string ToString() => $"{Value} ({Description})";
}

public static void Main()
{
    var m1 = new Measurement(5);
    Console.WriteLine(m1);  // output: 5 (Ordinary measurement)

    var m2 = new Measurement();
    Console.WriteLine(m2);  // output: 0 ()
}
```

Если вы не объявите конструктор без параметров явным образом, тип структуры предоставит стандартный конструктор без параметров со следующим поведением:

- Если тип структуры имеет явные конструкторы экземпляров или не имеет инициализаторов полей, неявный конструктор без параметров создает значение по умолчанию с типом структуры, независимо от существующих инициализаторов полей, как показано в примере выше.
- Если тип структуры не имеет явных конструкторов экземпляров, но содержит инициализаторы полей, компилятор создает общедоступный конструктор без параметров, который выполняет указанные инициализации полей, как показано в следующем примере:

```
public struct Coords
{
    public double X = double.NaN;
    public double Y = double.NaN;

    public override string ToString() => $"({X}, {Y})";
}

public static void Main()
{
    var p1 = new Coords();
    Console.WriteLine(p1);  // output: (NaN, NaN)

    var p2 = default(Coords);
    Console.WriteLine(p2);  // output: (0, 0)

    var ps = new Coords[3];
    Console.WriteLine(string.Join(", ", ps));  // output: (0, 0), (0, 0), (0, 0)
}
```

Как видно в примере выше, выражение значения по умолчанию и процедура создания экземпляра массива игнорируют инициализаторы полей.

Дополнительные сведения см. в примечании к предложению новой возможности в разделе Конструкторы структур без параметров.

Создание экземпляра типа структуры

В C# необходимо инициализировать объявленную переменную, прежде чем ее можно будет использовать. Так как переменная типа структуры не может быть `null` (если только это не переменная типа значения, допускающего значение NULL), необходимо создать экземпляр соответствующего типа. Для этого можно использовать несколько способов.

Как правило, экземпляр типа структуры создается посредством вызова соответствующего конструктора с помощью оператора `new`. Каждый тип структуры имеет по крайней мере один конструктор. Это неявный конструктор без параметров, который создает значение по умолчанию данного типа. Выражения значения по умолчанию можно также использовать для создания значения типа по умолчанию.

Если все поля экземпляров типа структуры доступны, можно также создать его экземпляр без оператора `new`. В этом случае необходимо инициализировать все поля экземпляров перед первым использованием экземпляра. Следующий пример показывает, как это сделать:

```
public static class StructWithoutNew
{
    public struct Coords
    {
        public double x;
        public double y;
    }

    public static void Main()
    {
        Coords p;
        p.x = 3;
        p.y = 4;
        Console.WriteLine($"({p.x}, {p.y})");  // output: (3, 4)
    }
}
```

В случае встроенных типов значения используйте соответствующие литералы, чтобы указать значение типа.

Передача переменных типа структуры по ссылке

При передаче переменной типа структуры в метод в качестве аргумента или возврате значения типа структуры из метода копируется весь экземпляр типа структуры. Это может повлиять на производительность кода в высокопроизводительных сценариях, где используются большие типы структуры. Копирования значений можно избежать, передав переменную типа структуры по ссылке. Используйте модификаторы параметра метода `ref`, `out` или `in`, чтобы указать, что аргумент должен передаваться по ссылке. Чтобы возвратить результат метода по ссылке, используйте ref returns. Дополнительные сведения см. в статье Написание безопасного и эффективного кода C#.

Структура `ref`

Начиная с C# 7.2 в объявлении типа структуры можно использовать модификатор `ref`. Экземпляры типа структуры `ref` выделяются в стеке и не могут временно перейти в управляемую кучу. Для этого компилятор ограничивает использование типов структуры `ref` следующим образом:

- Структура `ref` не может быть типом элемента массива.

- Структура `ref` не может быть объявленным типом поля класса или структурой, отличной от `ref`.
- Структура `ref` не может реализовывать интерфейсы.
- Структура `ref` не может быть упакована в System.ValueType или System.Object.
- Структура `ref` не может быть аргументом типа.
- Переменная структуры `ref` не может быть зафиксирована лямбда-выражением или локальной функцией.
- Переменную структуры `ref` нельзя использовать в методе `async`. Однако переменные структуры `ref` можно использовать в синхронных методах, например в тех, которые возвращают Task или Task<TResult>.
- Переменную структуры `ref` нельзя использовать в итераторах.

Как правило, тип структуры `ref` определяется, если требуется тип, который также содержит члены данных типов структуры `ref`:

```
public ref struct CustomRef
{
    public bool IsValid;
    public Span<int> Inputs;
    public Span<int> Outputs;
}
```

Чтобы объявить структуру `ref` как `readonly`, объедините модификаторы `readonly` и `ref` в объявлении типа (модификатор `readonly` должен предшествовать модификатору `ref`):

```
public readonly ref struct ConversionRequest
{
    public ConversionRequest(double rate, ReadOnlySpan<double> values)
    {
        Rate = rate;
        Values = values;
    }

    public double Rate { get; }
    public ReadOnlySpan<double> Values { get; }
}
```

В .NET примерами структуры `ref` являются System.Span<T> и System.ReadOnlySpan<T>.

Ограничение struct

Вы можете использовать ключевое слово `struct` в ограничении `struct`, чтобы указать, что параметр типа является типом значения, не допускающим значения NULL. Типы структуры и перечисления удовлетворяют ограничению `struct`.

Преобразования

Для любого типа структуры (за исключением типов структуры `ref`) существует упаковка-преобразование и распаковка-преобразование в типы System.ValueType и System.Object и из них. Существуют упаковка-преобразование и распаковка-преобразование между типом структуры и любым интерфейсом, который он реализует.

Спецификация языка C#

Дополнительные сведения см. в разделе Структуры в спецификации языка C#.

Дополнительные сведения о функциях, появившихся в C# 7.2 и более поздних версиях, см. в следующих заметках о функциях.

- Структуры только для чтения
- Члены экземпляров только для чтения
- Безопасность во время компиляции для типов, схожих с ссылочными
- Конструкторы структур без параметров
- Разрешить выражение `with` для структур

См. также

- справочник по C#
- Рекомендации по проектированию — выбор между классом и структурой
- Рекомендации по проектированию — проектирование структуры
- Система типов C#

Типы кортежей (справочник по C#)

18.12.2021 • 7 minutes to read

Кортежи, доступные в C# 7.0 и более поздних версиях, предоставляют краткий синтаксис для группирования нескольких элементов данных в упрощенную структуру данных. В следующем примере показано, как можно объявить переменную кортежа, инициализировать ее и получить доступ к ее элементам данных.

```
(double, int) t1 = (4.5, 3);
Console.WriteLine($"Tuple with elements {t1.Item1} and {t1.Item2}.");
// Output:
// Tuple with elements 4.5 and 3.

(double Sum, int Count) t2 = (4.5, 3);
Console.WriteLine($"Sum of {t2.Count} elements is {t2.Sum}.");
// Output:
// Sum of 3 elements is 4.5.
```

Как показано в предыдущем примере, для определения типа кортежа необходимо указать типы всех его элементов данных и, при необходимости, имена полей. В типе кортежа невозможно определить методы, но можно использовать методы, предоставляемые .NET, как показано в следующем примере.

```
(double, int) t = (4.5, 3);
Console.WriteLine(t.ToString());
Console.WriteLine($"Hash code of {t} is {t.GetHashCode()}.");
// Output:
// (4.5, 3)
// Hash code of (4.5, 3) is 718460086.
```

Начиная с C# 7.3, типы кортежей поддерживают операторы равенства `==` и `!=`. Дополнительные сведения см. в разделе Равенство кортежей.

Типы кортежей являются типами значений, а элементы кортежа — общедоступными полями. Поэтому кортежи представляют собой *изменяемые* типы значений.

> **NOTE**
> Для кортежей требуется тип System.ValueTuple и связанные универсальные типы (например, System.ValueTuple<T1,T2>), доступные в .NET Core и .NET Framework 4.7 и более поздних версий. Чтобы использовать кортежи в проекте, предназначенном для .NET Framework 4.6.2 или более ранней версии, добавьте в проект пакет NuGet `System.ValueTuple`.

Можно определить кортежи со сколь угодно большим числом элементов.

```
var t =
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13, 14, 15, 16, 17, 18,
19, 20, 21, 22, 23, 24, 25, 26);
Console.WriteLine(t.Item26);  // output: 26
```

Варианты использования кортежей

Чаще всего кортежи используются как возвращаемый методом тип. То есть вместо определения параметров метода `out` можно сгруппировать результаты метода в возвращаемый тип кортежа, как показано в следующем примере.

```
var xs = new[] { 4, 7, 9 };
var limits = FindMinMax(xs);
Console.WriteLine($"Limits of [{string.Join(" ", xs)}] are {limits.min} and {limits.max}");
// Output:
// Limits of [4 7 9] are 4 and 9

var ys = new[] { -9, 0, 67, 100 };
var (minimum, maximum) = FindMinMax(ys);
Console.WriteLine($"Limits of [{string.Join(" ", ys)}] are {minimum} and {maximum}");
// Output:
// Limits of [-9 0 67 100] are -9 and 100

(int min, int max) FindMinMax(int[] input)
{
    if (input is null || input.Length == 0)
    {
        throw new ArgumentException("Cannot find minimum and maximum of a null or empty array.");
    }

    var min = int.MaxValue;
    var max = int.MinValue;
    foreach (var i in input)
    {
        if (i < min)
        {
            min = i;
        }
        if (i > max)
        {
            max = i;
        }
    }
    return (min, max);
}
```

Как показано в предыдущем примере, с возвращаемым экземпляром кортежа можно работать напрямую или деконструировать его в отдельные переменные.

Типы кортежей можно также использовать вместо анонимных типов, например в запросах LINQ. Дополнительные сведения см. в статье Выбор между анонимными типами и кортежами.

Как правило, кортежи используются для группирования слабо связанных элементов данных. Это целесообразно в закрытых и внутренних служебных методах. При работе с общедоступным API рассмотрите возможность определения типа класса или структуры.

Имена полей кортежей

Имена полей кортежей указываются явным образом либо в выражении инициализации кортежа, либо в определении типа кортежа, как показано в следующем примере.

```
var t = (Sum: 4.5, Count: 3);
Console.WriteLine($"Sum of {t.Count} elements is {t.Sum}.");

(double Sum, int Count) d = (4.5, 3);
Console.WriteLine($"Sum of {d.Count} elements is {d.Sum}.");
```

Начиная с C# 7.1, если имя поля не указано, оно может быть выведено из имени соответствующей

переменной в выражении инициализации кортежа, как показано в следующем примере.

```
var sum = 4.5;
var count = 3;
var t = (sum, count);
Console.WriteLine($"Sum of {t.count} elements is {t.sum}.");
```

Это называется инициализаторами проекций кортежа. Имя переменной не проецируется на имя поля кортежа в следующих случаях:

- Имя кандидата — это имя элемента типа кортежа, например `Item3`, `ToString` или `Rest`.
- Имя кандидата является дубликатом другого имени поля кортежа, явного или неявного.

В этих случаях необходимо либо явно указать имя поля, либо получить доступ к полю по имени по умолчанию.

По умолчанию поля кортежа имеют имена `Item1`, `Item2`, `Item3` и т. д. Всегда можно использовать имя поля по умолчанию, даже если имя поля указано явно или является выводимым, как показано в следующем примере.

```
var a = 1;
var t = (a, b: 2, 3);
Console.WriteLine($"The 1st element is {t.Item1} (same as {t.a}).");
Console.WriteLine($"The 2nd element is {t.Item2} (same as {t.b}).");
Console.WriteLine($"The 3rd element is {t.Item3}.");
// Output:
// The 1st element is 1 (same as 1).
// The 2nd element is 2 (same as 2).
// The 3rd element is 3.
```

Имена полей не учитываются при присваивании кортежа и сравнении кортежей на равенство.

Во время компиляции компилятор заменяет имена полей не по умолчанию соответствующими именами по умолчанию. В результате явно указанные или выводимые имена полей будут недоступны во время выполнения.

Присваивание и деконструкция кортежей

В C# поддерживается присваивание между типами кортежей, которые соответствуют обоим следующим условиям:

- оба типа кортежей должны содержать одинаковое количество элементов;
- для каждой позиции кортежа тип правого элемента кортежа аналогичен типу соответствующего левого элемента кортежа или может быть неявно преобразован в этот тип.

Значения элементов кортежа присваиваются в порядке расположения элементов кортежа. Имена полей кортежа не учитываются и не присваиваются, как показано в следующем примере.

```
(int, double) t1 = (17, 3.14);
(double First, double Second) t2 = (0.0, 1.0);
t2 = t1;
Console.WriteLine($"{nameof(t2)}: {t2.First} and {t2.Second}");
// Output:
// t2: 17 and 3.14

(double A, double B) t3 = (2.0, 3.0);
t3 = t2;
Console.WriteLine($"{nameof(t3)}: {t3.A} and {t3.B}");
// Output:
// t3: 17 and 3.14
```

Оператор присваивания `=` можно также использовать для *деконструкции* экземпляра кортежа в отдельные переменные. Существует три способа деконструкции кортежа.

- Вы можете явно объявить тип каждой переменной в скобках.

```
var t = ("post office", 3.6);
(string destination, double distance) = t;
Console.WriteLine($"Distance to {destination} is {distance} kilometers.");
// Output:
// Distance to post office is 3.6 kilometers.
```

- Вы можете использовать ключевое слово `var` за пределами круглых скобок, чтобы объявить неявно типизированные переменные и позволить компилятору вывести их типы.

```
var t = ("post office", 3.6);
var (destination, distance) = t;
Console.WriteLine($"Distance to {destination} is {distance} kilometers.");
// Output:
// Distance to post office is 3.6 kilometers.
```

- Вы можете использовать существующие переменные.

```
var destination = string.Empty;
var distance = 0.0;

var t = ("post office", 3.6);
(destination, distance) = t;
Console.WriteLine($"Distance to {destination} is {distance} kilometers.");
// Output:
// Distance to post office is 3.6 kilometers.
```

Подробнее о деконструкции кортежей с помощью и других типов см. в статье Деконструкция кортежей и других типов.

Равенство кортежей

Начиная с C# 7.3, типы кортежей поддерживают операторы `==` и `!=`. Эти операторы сравнивают элементы левого операнда с соответствующими элементами правого операнда в соответствии с порядком расположения элементов кортежа.

```
(int a, byte b) left = (5, 10);
(long a, int b) right = (5, 10);
Console.WriteLine(left == right);  // output: True
Console.WriteLine(left != right);  // output: False

var t1 = (A: 5, B: 10);
var t2 = (B: 5, A: 10);
Console.WriteLine(t1 == t2);  // output: True
Console.WriteLine(t1 != t2);  // output: False
```

Как показано в предыдущем примере, в операциях `==` и `!=` не учитываются имена полей кортежей.

Два кортежа сравнимы, если выполнены оба следующих условия:

- оба кортежа содержат одинаковое количество элементов. Например, `t1 != t2` не компилируется, если `t1` и `t2` имеют разное количество элементов.
- Для каждой позиции кортежа соответствующие элементы из левого и правого операндов кортежа сравниваются с помощью операторов `==` и `!=`. Например, `(1, (2, 3)) == ((1, 2), 3)` не компилируется, поскольку `1` не сравнивается с помощью `(1, 2)`.

Операторы `==` и `!=` сравнивают кортежи с сокращенной обработкой. Это значит, что операция останавливается, как только она соответствует паре неравных элементов или достигает конца кортежей. Однако перед любым сравнением *все* элементы кортежа вычисляются, как показано в следующем примере.

```
Console.WriteLine((Display(1), Display(2)) == (Display(3), Display(4)));

int Display(int s)
{
    Console.WriteLine(s);
    return s;
}
// Output:
// 1
// 2
// 3
// 4
// False
```

Кортежи как параметры вывода

Как правило, вы выполняете рефакторинг метода, имеющего параметры `out`, в метод, возвращающий кортеж. Однако бывают случаи, когда параметр `out` может иметь тип кортежа. В следующем примере показано, как работать с кортежами в виде параметров `out`.

```
var limitsLookup = new Dictionary<int, (int Min, int Max)>()
{
    [2] = (4, 10),
    [4] = (10, 20),
    [6] = (0, 23)
};

if (limitsLookup.TryGetValue(4, out (int Min, int Max) limits))
{
    Console.WriteLine($"Found limits: min is {limits.Min}, max is {limits.Max}");
}
// Output:
// Found limits: min is 10, max is 20
```

Кортежи и `System.Tuple`

Кортежи C# с типами System.ValueTuple, отличаются от кортежей, представленных типами System.Tuple. Основные различия заключаются в следующем.

- Типы `System.ValueTuple` являются типами значений. Типы `System.Tuple` являются ссылочными типами.
- Типы `System.ValueTuple` являются изменяемыми. Типы `System.Tuple` являются неизменяемыми.
- Элементами данных типов `System.ValueTuple` являются поля. Элементами данных типов `System.Tuple` являются свойства.

Спецификация языка C#

Дополнительные сведения см. в следующих примечаниях к предлагаемой функции.

- Выводимые имена кортежей (инициализаторы проекций кортежа)
- Поддержка `==` и `!=` для типов кортежей

См. также

- справочник по C#
- Типы значений
- Выбор между анонимными типами и кортежами
- System.ValueTuple

Справочник по C#. Типы значений, допускающие значение NULL

18.12.2021 • 7 minutes to read

Тип значений, допускающий значение NULL, или `T?`, представляет все значения своего базового типа значения `T`, а также дополнительное значение NULL. Например, можно присвоить переменной `bool?` любое из следующих трех значений: `true`, `false` или `null`. Базовый тип значения `T` не может соответствовать типу значения, допускающему значение NULL.

> **NOTE**
> В C# 8.0 появилась возможность использования ссылочных типов, допускающих значение NULL. Дополнительные сведения см. в статье Ссылочные типы, допускающие значение NULL. Типы значений, допускающие значение NULL, доступны начиная с C# 2.

Типы, допускающие значение NULL, представляют собой экземпляры универсальной структуры System.Nullable<T>. Вы можете ссылаться на тип значения, допускающий значение NULL, с базовым типом `T` в любой из следующих взаимозаменяемых форм: `Nullable<T>` или `T?`.

Тип значения, допускающий значение NULL, следует использовать, когда нужно представить неопределенное значение его базового типа. Например, логическая переменная (или `bool`) может иметь только значения `true` или `false`. Однако в некоторых приложениях значение переменной может быть неопределенным или отсутствовать. Например, поле базы данных может содержать значение `true` или `false` либо вообще никакого значения, то есть `NULL`. В этом сценарии можно использовать тип `bool?`.

Назначение и объявление

Так как тип значения можно неявно преобразовать в соответствующий тип значения, допускающий значение NULL, вы назначаете значение переменной такого типа значения так же, как для базового типа значения. Вы можете также присвоить значение `null`. Пример:

```
double? pi = 3.14;
char? letter = 'a';

int m2 = 10;
int? m = m2;

bool? flag = null;

// An array of a nullable value type:
int?[] arr = new int?[10];
```

Значение по умолчанию для типа значения, допускающего значение NULL, представляет `null`, то есть это экземпляр, свойство Nullable<T>.HasValue которого возвращает `false`.

Проверка экземпляра типа значения, допускающего значение NULL

Начиная с версии C# 7.0 можно использовать оператор `is` с шаблоном типа как для проверки

экземпляра типа, допускающего значение NULL, для `null`, так и для извлечения значения базового типа:

```
int? a = 42;
if (a is int valueOfA)
{
    Console.WriteLine($"a is {valueOfA}");
}
else
{
    Console.WriteLine("a does not have a value");
}
// Output:
// a is 42
```

Вы всегда можете использовать следующие свойства только для чтения, чтобы проверить и получить значение переменной типа, допускающего значение NULL:

- Nullable<T>.HasValue указывает, имеет ли экземпляр типа, допускающего значение NULL, значение своего базового типа.
- Nullable<T>.Value возвращает значение базового типа, если HasValue имеет значение `true`. Если HasValue имеет значение `false`, свойство Value выдает исключение InvalidOperationException.

В следующем примере используется свойство `HasValue`, чтобы проверить, содержит ли переменная значение, перед его отображением:

```
int? b = 10;
if (b.HasValue)
{
    Console.WriteLine($"b is {b.Value}");
}
else
{
    Console.WriteLine("b does not have a value");
}
// Output:
// b is 10
```

Можно также сравнить переменную типа значения, допускающего значение NULL, с `null` вместо использования свойства `HasValue`, как показано в следующем примере:

```
int? c = 7;
if (c != null)
{
    Console.WriteLine($"c is {c.Value}");
}
else
{
    Console.WriteLine("c does not have a value");
}
// Output:
// c is 7
```

Преобразование из типа значения, допускающего значение NULL, в базовый тип

Если необходимо присвоить значение типа, допускающего значение NULL, переменной типа значения, не допускающего значения NULL, может потребоваться указать значение, назначаемое вместо `null`. Для

этого используйте оператор объединения со значением NULL `??` (можно также применить метод Nullable<T>.GetValueOrDefault(T) для той же цели):

```
int? a = 28;
int b = a ?? -1;
Console.WriteLine($"b is {b}");  // output: b is 28

int? c = null;
int d = c ?? -1;
Console.WriteLine($"d is {d}");  // output: d is -1
```

Если вы хотите использовать значение по умолчанию базового типа значения вместо `null`, воспользуйтесь методом Nullable<T>.GetValueOrDefault().

Вы можете также явно привести тип значения, допускающий значение NULL, к типу, не допускающему значение NULL, как показано в примере ниже.

```
int? n = null;

//int m1 = n;    // Doesn't compile
int n2 = (int)n; // Compiles, but throws an exception if n is null
```

Во время выполнения, если значение типа значения, допускающего значение NULL, равно `null`, явное приведение вызывает исключение InvalidOperationException.

Тип `T`, не допускающий значение NULL, неявно преобразуется в соответствующий тип, допускающий значение NULL, `T?`.

Операторы с нулификацией

Предопределенные унарные и бинарные операторы или любые перегруженные операторы, поддерживаемые типом значения `T`, также поддерживаются соответствующим типом значения, допускающим значение NULL, т. е. `T?`. Эти операторы, также называемые *операторами с нулификацией*, возвращают значение `null`, если один или оба операнда имеют значение `null`. В противном случае оператор использует содержащиеся значения операндов для вычисления результата. Пример:

```
int? a = 10;
int? b = null;
int? c = 10;

a++;        // a is 11
a = a * c;  // a is 110
a = a + b;  // a is null
```

NOTE

Для типа `bool?` предопределенные операторы `&` и `|` не следуют правилам, описанным в этом разделе: результат вычисления оператора может быть отличным от NULL, даже если один из операндов имеет значение `null`. См. подробнее о логических операторах, поддерживающих значение NULL в описании логических операторов.

Для операторов сравнения `<`, `>`, `<=` и `>=`, если один или оба операнда равны `null`, результат будет равен `false`. В противном случае сравниваются содержащиеся значения операндов. Тут важно не полагать, что если какая-то операция сравнения (например, `<=`) возвращает `false`, то

противоположное сравнение (`>`) обязательно вернет `true`. В следующем примере показано, что 10

- не больше и не равно значению `null`,
- не меньше чем `null`.

```
int? a = 10;
Console.WriteLine($"{a} >= null is {a >= null}");
Console.WriteLine($"{a} < null is {a < null}");
Console.WriteLine($"{a} == null is {a == null}");
// Output:
// 10 >= null is False
// 10 < null is False
// 10 == null is False

int? b = null;
int? c = null;
Console.WriteLine($"null >= null is {b >= c}");
Console.WriteLine($"null == null is {b == c}");
// Output:
// null >= null is False
// null == null is True
```

Для оператора равенства `==`, если оба операнда равны `null`, результат будет равен `true`. Если один из операндов равен `null`, результат будет равен `false`. В противном случае сравниваются содержащиеся значения операндов.

Для оператора неравенства `!=`, если оба операнда равны `null`, результат будет равен `false`. Если один из операндов равен `null`, результат будет равен `true`. В противном случае сравниваются содержащиеся значения операндов.

Если между двумя типами данных определено пользовательское преобразование типов, то это же преобразование можно также использовать между соответствующими типами, допускающими значение NULL.

Упаковка-преобразование и распаковка-преобразование

Экземпляр типа значения, допускающего значение NULL, `T?` упакован следующим образом:

- Если HasValue возвращает `false`, создается пустая ссылка.
- Если HasValue возвращает `true`, упаковывается соответствующее значение базового типа `T`, а не экземпляр Nullable<T>.

Можно распаковать упакованный тип значения `T` в соответствующий тип, допускающий значение NULL, `T?`, как показано в следующем примере:

```
int a = 41;
object aBoxed = a;
int? aNullable = (int?)aBoxed;
Console.WriteLine($"Value of aNullable: {aNullable}");

object aNullableBoxed = aNullable;
if (aNullableBoxed is int valueOfA)
{
    Console.WriteLine($"aNullableBoxed is boxed int: {valueOfA}");
}
// Output:
// Value of aNullable: 41
// aNullableBoxed is boxed int: 41
```

Идентификация типа значений, допускающего значение NULL

В следующем примере показано, как определить, представляет ли экземпляр System.Type сконструированный тип значений, допускающий значение NULL, т. е. тип System.Nullable<T> с указанным параметром типа `T`:

```
Console.WriteLine($"int? is {(IsNullable(typeof(int?)) ? "nullable" : "non nullable")} value type");
Console.WriteLine($"int is {(IsNullable(typeof(int)) ? "nullable" : "non-nullable")} value type");

bool IsNullable(Type type) => Nullable.GetUnderlyingType(type) != null;

// Output:
// int? is nullable value type
// int is non-nullable value type
```

Как показано в примере, при помощи оператора typeof можно создать экземпляр System.Type.

Если вы хотите определить, принадлежит ли экземпляр к типу значений, допускающему значение NULL, не используйте метод Object.GetType для получения экземпляра Type для тестирования с помощью приведенного выше кода. При вызове метода Object.GetType в экземпляре типа значений, допускающего значение NULL, экземпляр упаковывается в Object. Так как упаковка экземпляра типа значений, допускающего значение NULL, значение которого отлично от NULL, эквивалентна упаковке значения базового типа, GetType возвращается экземпляр Type, представляющий базовый тип типа значений, допускающего значение NULL:

```
int? a = 17;
Type typeOfA = a.GetType();
Console.WriteLine(typeOfA.FullName);
// Output:
// System.Int32
```

Кроме того, не используйте оператор is, чтобы определить, принадлежит ли экземпляр к типу значений, допускающему значение NULL. Как показано в следующем примере, вы не можете отличить типы экземпляра типа значений, допускающего значение NULL, и его экземпляра базового типа с помощью оператора `is`:

```
int? a = 14;
if (a is int)
{
    Console.WriteLine("int? instance is compatible with int");
}

int b = 17;
if (b is int?)
{
    Console.WriteLine("int instance is compatible with int?");
}
// Output:
// int? instance is compatible with int
// int instance is compatible with int?
```

Чтобы определить, принадлежит ли экземпляр типу значений, допускающему значение NULL, можно использовать код, представленный в следующем примере:

```
int? a = 14;
Console.WriteLine(IsOfNullableType(a));  // output: True

int b = 17;
Console.WriteLine(IsOfNullableType(b));  // output: False

bool IsOfNullableType<T>(T o)
{
    var type = typeof(T);
    return Nullable.GetUnderlyingType(type) != null;
}
```

> **NOTE**
>
> Методы, описанные в этом разделе, неприменимы в случае ссылочных типов, допускающих значения NULL.

Спецификация языка C#

Дополнительные сведения см. в следующих разделах статьи Спецификация языка C#:

- Типы, допускающие значения NULL
- Операторы с нулификацией
- Неявные преобразования, допускающие значения NULL
- Явные преобразования, допускающие значения NULL
- Операторы преобразования с нулификацией

См. также

- справочник по C#
- What exactly does 'lifted' mean? (Что означает термин "расширенные"?)
- System.Nullable<T>
- System.Nullable
- Nullable.GetUnderlyingType
- Ссылочные типы, допускающие значение NULL

Ссылочные типы (справочник по C#)

18.12.2021 • 2 minutes to read

В C# существуют две разновидности типов: ссылочные типы и типы значений. В переменных ссылочных типов хранятся ссылки на их данные (объекты), а переменные типа значений содержат свои данные непосредственно. Две переменные ссылочного типа могут ссылаться на один и тот же объект, поэтому операции над одной переменной могут затрагивать объект, на который ссылается другая переменная. Каждая переменная типа значения имеет собственную копию данных, и операции над одной переменной не могут затрагивать другую (за исключением переменных параметров in, ref и out; см. описание модификатора параметров in, ref и out).

Для объявления ссылочных типов используются следующие ключевые слова:

- class
- interface
- delegate
- record

В C# также предусмотрены следующие встроенные ссылочные типы:

- dynamic
- object
- строка

См. также

- Справочник по C#
- Ключевые слова в C#
- Типы указателей
- Типы значений

Встроенные ссылочные типы (справочник по C#)

18.12.2021 • 6 minutes to read

В C# есть ряд встроенных ссылочных типов. У них есть ключевые слова или операторы, которые являются синонимами типа в библиотеке .NET.

Тип object

Тип `object` является псевдонимом System.Object в .NET. В унифицированной системе типов C# все типы, стандартные и определяемые пользователем, ссылочные типы и типы значений напрямую или косвенно наследуются из System.Object. Переменным типа `object` можно назначать значения любого типа. Любой переменной `object` можно назначить значение по умолчанию с помощью литерала `null`. Если переменная типа значения преобразуется в объект, она считается *упакованной*. Если переменная типа `object` преобразуется в тип значения, она считается *распакованной*. Дополнительные сведения см. в разделе Упаковка-преобразование и распаковка-преобразование.

Тип string

Тип `string` представляет последовательность, состоящую из нуля или более символов в кодировке Юникод. `string` является псевдонимом для System.String в .NET.

Несмотря на то что `string` представляет собой ссылочный тип, операторы равенства `==` и `!=` по определению сравнивают не ссылки, а значения объектов `string`. Это делает проверку равенства строк более интуитивно понятной. Пример:

```
string a = "hello";
string b = "h";
// Append to contents of 'b'
b += "ello";
Console.WriteLine(a == b);
Console.WriteLine(object.ReferenceEquals(a, b));
```

Отображается значение True, а затем False, поскольку содержимое строк эквивалентно, однако `a` и `b` не относятся к одному и тому же экземпляру строки.

Оператор + объединяет строки:

```
string a = "good " + "morning";
```

При этом создается строковый объект, содержащий слова "good morning".

Строки *неизменяемы*, т. е. содержимое созданного строкового объекта изменить нельзя, хотя синтаксис выглядит так, будто это возможно. Например, при написании кода компилятор фактически создает новый строковый объект для хранения новой последовательности символов, а затем этот новый объект назначается `b`. Память, выделенная для `b` (если он содержит строку h), затем доступна для сборки мусора.

```
string b = "h";
b += "ello";
```

Оператор `[]` позволяет предоставить к отдельным символам строки доступ только на чтение. Допустимые значения индекса начинаются с `0` и должны быть меньше, чем длина строки:

```
string str = "test";
char x = str[2];  // x = 's';
```

Также оператор `[]` можно использовать для итерации каждого символа в строке:

```
string str = "test";

for (int i = 0; i < str.Length; i++)
{
  Console.Write(str[i] + " ");
}
// Output: t e s t
```

Строковые литералы имеют тип `string` и могут быть записаны в двух формах: в кавычках (") или в кавычках с префиксом `@`. Строковые литералы в кавычках заключаются в двойные кавычки ("):

```
"good morning"  // a string literal
```

Строковые литералы могут содержать любые символьные литералы. Escape-последовательности включены. В следующем примере escape-последовательность `\\` используется для получения обратной косой черты, `\u0066` — для получения буквы f, и `\n` — для получения новой строки.

```
string a = "\\\u0066\n F";
Console.WriteLine(a);
// Output:
// \f
//  F
```

NOTE

Escape-код `\udddd` (где `dddd` состоит из четырех цифр) представляет символ Юникода U+`dddd`. Также распознаются восьмизначные escape-коды Юникода: `\Uddddddddd`.

Буквальные строковые литералы начинаются с `@` и также заключаются в двойные кавычки. Пример:

```
@"good morning"  // a string literal
```

Преимущество буквальных строк состоит в том, что escape-последовательности *не* обрабатываются, что позволяет легко написать, например, полное имя файла Windows:

```
@"c:\Docs\Source\a.txt"  // rather than "c:\\Docs\\Source\\a.txt"
```

Чтобы добавить в строку @-quoted двойные кавычки, продублируйте этот символ:

```
@"""Ahoy!"" cried the captain." // "Ahoy!" cried the captain.
```

Тип delegate

Объявление типа делегата аналогично сигнатуре метода. Оно имеет возвращаемое значение и любое число параметров любого типа:

```
public delegate void MessageDelegate(string message);
public delegate int AnotherDelegate(MyType m, long num);
```

В .NET типы `System.Action` и `System.Func` предоставляют общие определения для многих распространенных делегатов. Скорее всего, не нужно определять новые пользовательские типы делегата. Вместо этого можно создать экземпляры предоставленных универсальных типов.

Ключевое слово `delegate` имеет ссылочный тип, который можно использовать для инкапсуляции именованного или анонимного метода. Делегаты аналогичны используемым в языке C++ указателям функций, но являются типобезопасными и безопасными. Сведения о применении делегатов см. в разделах Делегаты и Универсальные делегаты. Делегаты являются основой событий. Экземпляры делегата могут создаваться путем его связывания с именованным или анонимным методом.

Делегат должен быть создан при помощи метода или лямбда-выражения, имеющего совместимые возвращаемый тип и входные параметры. Дополнительные сведения о допустимой степени вариации сигнатур методов см. в разделе Вариативность в делегатах. Для использования с анонимными методами делегат и код, который должен быть связан с ним, должны быть объявлены вместе.

Комбинирование и удаление делегата завершаются ошибкой из-за исключения среды выполнения, когда типы делегатов, задействованные во время выполнения, отличаются из-за преобразования вариантов. В следующем примере показана ситуация, в которой происходит сбой:

```
Action<string> stringAction = str => {};
Action<object> objectAction = obj => {};

// Valid due to implicit reference conversion of
// objectAction to Action<string>, but may fail
// at run time.
Action<string> combination = stringAction + objectAction;
```

Вы можете создать делегат с правильным типом среды выполнения, создав новый объект делегата. В следующем примере показано, как это решение можно применить к предыдущему примеру.

```
Action<string> stringAction = str => {};
Action<object> objectAction = obj => {};

// Creates a new delegate instance with a runtime type of Action<string>.
Action<string> wrappedObjectAction = new Action<string>(objectAction);

// The two Action<string> delegate instances can now be combined.
Action<string> combination = stringAction + wrappedObjectAction;
```

Начиная с C# 9, можно объявить *указатели функций*, которые используют аналогичный синтаксис. Указатель функции использует инструкцию `calli` вместо создания экземпляра типа делегата и вызова виртуального метода `Invoke`.

Тип dynamic

Тип `dynamic` указывает, что использование переменной и ссылок на ее члены обходит проверку типа во время компиляции. Такие операции разрешаются во время выполнения. Тип `dynamic` упрощает доступ к API COM, таким как API автоматизации Office, к динамическим API, таким как библиотеки IronPython, и к HTML-модели DOM.

Тип `dynamic` в большинстве случаев ведет себя как тип `object`. В частности, можно преобразовать любое выражение, отличное от NULL, в тип `dynamic`. Тип `dynamic` отличается от `object` тем, что операции, которые содержат выражения типа `dynamic`, не разрешаются или тип проверяется компилятором. Компилятор объединяет сведения об операции, которые впоследствии будут использоваться для оценки этой операции во время выполнения. В рамках этого процесса переменные типа `dynamic` компилируются в переменные типа `object`. Таким образом, тип `dynamic` существует только во время компиляции, но не во время выполнения.

В следующем примере переменной типа `dynamic` противопоставляется переменная типа `object`. Чтобы проверить тип каждой переменной во время компиляции, наведите указатель мыши на `dyn` или `obj` в операторах `WriteLine`. Скопируйте следующий код в редактор, где доступен IntelliSense. IntelliSense отображает **dynamic** для `dyn` и **object** для `obj`.

```
class Program
{
    static void Main(string[] args)
    {
        dynamic dyn = 1;
        object obj = 1;

        // Rest the mouse pointer over dyn and obj to see their
        // types at compile time.
        System.Console.WriteLine(dyn.GetType());
        System.Console.WriteLine(obj.GetType());
    }
}
```

Операторы WriteLine отображают типы времени выполнения `dyn` и `obj`. На этом этапе оба имеют один и тот же тип — целое число. Выводятся следующие результаты.

```
System.Int32
System.Int32
```

Чтобы увидеть разницу между `dyn` и `obj` во время компиляции, добавьте между объявлениями и операторами `WriteLine` в предыдущем примере следующие две строки:

```
dyn = dyn + 3;
obj = obj + 3;
```

При попытке добавления целого числа и объекта в выражение `obj + 3` выдается ошибка компилятора. При этом для `dyn + 3` ошибка не возникает. Выражение, которое содержит `dyn`, не проверяется во время компиляции, поскольку тип `dyn` имеет значение `dynamic`.

В следующем примере `dynamic` используется в нескольких объявлениях. Метод `Main` также противопоставляет проверку типов во время компиляции.

```
using System;

namespace DynamicExamples
{
    class Program
    {
        static void Main(string[] args)
        {
            ExampleClass ec = new ExampleClass();
            Console.WriteLine(ec.exampleMethod(10));
            Console.WriteLine(ec.exampleMethod("value"));

            // The following line causes a compiler error because exampleMethod
            // takes only one argument.
            //Console.WriteLine(ec.exampleMethod(10, 4));

            dynamic dynamic_ec = new ExampleClass();
            Console.WriteLine(dynamic_ec.exampleMethod(10));

            // Because dynamic_ec is dynamic, the following call to exampleMethod
            // with two arguments does not produce an error at compile time.
            // However, it does cause a run-time error.
            //Console.WriteLine(dynamic_ec.exampleMethod(10, 4));
        }
    }

    class ExampleClass
    {
        static dynamic field;
        dynamic prop { get; set; }

        public dynamic exampleMethod(dynamic d)
        {
            dynamic local = "Local variable";
            int two = 2;

            if (d is int)
            {
                return local;
            }
            else
            {
                return two;
            }
        }
    }
}
// Results:
// Local variable
// 2
// Local variable
```

См. также

- Справочник по C#
- Ключевые слова в C#
- События
- Использование типа dynamic
- Рекомендации по использованию строк
- Базовые операции со строками в .NET Framework
- Создание строк
- Операторы приведения и тестирования типов
- Практическое руководство. Безопасное приведение с помощью сопоставления шаблонов, а также

операторов is и as

- Пошаговое руководство. Создание и использование динамических объектов (C# и Visual Basic)
- System.Object
- System.String
- System.Dynamic.DynamicObject

Записи (справочник по C#)

18.12.2021 • 16 minutes to read

Начиная с C# версии 9 вы можете использовать ключевое слово `record` для определения ссылочного типа, который предоставляет встроенные возможности для инкапсуляции данных. Вы можете создавать типы записей с неизменяемыми свойствами, используя позиционные параметры или стандартный синтаксис свойств:

```
public record Person(string FirstName, string LastName);
```

```
public record Person
{
    public string FirstName { get; init; } = default!;
    public string LastName { get; init; } = default!;
};
```

Кроме того, можно создавать типы записей с изменяемыми свойствами и полями:

```
public record Person
{
    public string FirstName { get; set; } = default!;
    public string LastName { get; set; } = default!;
};
```

Несмотря на поддержку изменения, записи предназначены в первую очередь для неизменяемых моделей данных. Тип записи предоставляет следующие возможности:

- Краткий синтаксис для создания ссылочного типа с неизменяемыми свойствами.
- Встроенное поведение, полезное для ссылочного типа, ориентированного на данные:
  - Равенство значений
  - Краткий синтаксис для обратимого изменения.
  - Встроенное форматирование для отображения.
- Поддержка иерархий наследования.

Вы также можете на основе типов структур создать типы, ориентированные на данные, которые поддерживают равенство значений и почти не определяют поведение. В C# 10 и более поздних версиях можно определить типы `record struct` с помощью либо параметров позиционирования, либо синтаксиса стандартного свойства:

```
public readonly record struct Point(double X, double Y, double Z);
```

```
public record struct Point
{
    public double X {  get; init; }
    public double Y {  get; init; }
    public double Z {  get; init; }
}
```

Структуры записей также могут быть изменяемыми, как позиционированные структуры записей, так и

структуры записей без параметров позиционирования:

```
public record struct DataMeasurement(DateTime TakenAt, double Measurement);
```

```
public record struct Point
{
    public double X { get; set; }
    public double Y { get; set; }
    public double Z { get; set; }
}
```

В предыдущих примерах показаны некоторые различия между записями, которые являются ссылочными типами, и записями, которые являются типами значений:

- Объект `record` или `record class` объявляет ссылочный тип. Ключевое слово `class` является необязательным, но может добавить ясность для читателей. `record struct` объявляет тип значения.
- Свойства позиционирования являются *неизменяемыми* в `record class` и `readonly record struct`. Они являются *изменяемыми* в `record struct`.

В оставшейся части этой статьи обсуждаются типы `record class` и `record struct`. Различия подробно описаны в каждом разделе. Выберите между `record class` и `record struct`, как вы выбираете между `class` и `struct`. Термин *запись* используется для описания поведения, которое применяется ко всем типам записей. `record struct` или `record class` используется для описания поведения, которое применяется только к типам структур или классов соответственно.

Позиционный синтаксис для определения свойств

Позиционные параметры позволяют объявить свойства записи и инициализировать значения свойств при создании экземпляра:

```
public record Person(string FirstName, string LastName);

public static void Main()
{
    Person person = new("Nancy", "Davolio");
    Console.WriteLine(person);
    // output: Person { FirstName = Nancy, LastName = Davolio }
}
```

При использовании позиционного синтаксиса для определения свойства компилятор создает следующие элементы:

- Открытое автоматически реализуемое свойство "только init" создается для каждого позиционного параметра, предоставленного в объявлении записи.
  - Для типов `record` и `readonly record struct`: свойство только для инициализации может быть задано только в конструкторе или с помощью инициализатора свойств.
  - Для типов `record struct`: свойство для чтения и записи, которое может быть задано в конструкторе, инициализаторе свойства или назначении после создания.
- Основной конструктор, параметры которого соответствуют позиционным параметрам в объявлении записи.
- Для типов структуры записей конструктор без параметров присваивает каждому полю значение по умолчанию.
- Метод `Deconstruct` с параметром `out` создается для каждого позиционного параметра, предоставленного в объявлении записи. Этот метод предоставляется только в том случае, если

предоставлено два или более позиционных параметров. Этот метод деконструирует свойства, определенные с помощью позиционного синтаксиса, и игнорирует любые свойства, определенные с помощью стандартного синтаксиса.

Вы можете добавить атрибуты в любой из этих элементов, создаваемых компилятором из определения записи. Вы можете добавить *целевой объект* к любому атрибуту, который применяется к свойствам позиционной записи. В следующем примере System.Text.Json.Serialization.JsonPropertyNameAttribute применяется к каждому свойству записи `Person`. Целевой объект `property:` указывает, что атрибут применяется к свойству, созданному компилятором. Другие значения — `field:` для применения атрибута к полю и `param:` для применения атрибута к параметру.

```
/// <summary>
/// Person record type
/// </summary>
/// <param name="FirstName">First Name</param>
/// <param name="LastName">Last Name</param>
/// <remarks>
/// The person type is a positional record containing the
/// properties for the first and last name. Those properties
/// map to the JSON elements "firstName" and "lastName" when
/// serialized or deserialized.
/// </remarks>
public record Person([property: JsonPropertyName("firstName")]string FirstName,
    [property: JsonPropertyName("lastName")]string LastName);
```

В примере выше также показано, как создавать для записи комментарии XML-документации. Вы можете указать тег `<param>`, чтобы добавить документацию для параметров первичного конструктора.

Если вам не подходит созданное определение автоматически реализуемого свойства, вы можете определить собственное свойство с тем же именем. В этом случае созданные конструктор и деконструктор будут использовать предоставленное вами определение свойства. Например, в следующем примере объявляются свойства `FirstName` и `LastName` для позиционной записи `public`, но параметр позиционирования `Id` ограничен до `internal`. Этот синтаксис можно использовать для типов записей и структур записей. Необходимо добавить явное присвоение объявленного свойства соответствующему параметру позиционирования.

```
public record Person(string FirstName, string LastName, string Id)
{
    internal string Id { get; init; } = Id;
}

public static void Main()
{
    Person person = new("Nancy", "Davolio", "12345");
    Console.WriteLine(person.FirstName); //output: Nancy

}
```

В типе записи не обязательно объявлять позиционные свойства. Вы можете объявить запись без позиционных свойств или даже объявить другие поля и свойства, как показано в следующем примере:

```
public record Person(string FirstName, string LastName)
{
    public string[] PhoneNumbers { get; init; } = Array.Empty<string>();
};
```

Если вы определите свойства с использованием стандартного синтаксиса свойств, но опустите

модификатор доступа, эти свойства неявно становятся `private`.

Неизменяемость

Позиционная запись и *структура позиционной записи только для чтения* объявляют свойства только для инициализации. *Структура позиционированной записи* объявляет свойства, доступные для чтения и записи. Можно переопределить любое из этих значений по умолчанию, как показано в предыдущем разделе.

Неизменяемость может быть полезной, если требуется обеспечить потокобезопасность для типа, ориентированного на данные, или существует необходимость сохранять хэш-код в неизменном виде в хэш-таблице. Но все же неизменяемость пригодна не для всех сценариев работы с данными. Например, Entity Framework Core не поддерживает обновление с неизменяемыми типами сущностей.

Свойства только для инициализации, созданные на основе позиционных параметров (`record class` и `readonly record struct`) или путем указания методов доступа `init`, имеют *неполную неизменяемость*. После инициализации вы не сможете изменить значения свойств с типом значения или ссылки на свойства ссылочного типа. Но вы можете изменить сами данные, на которые ссылается свойство ссылочного типа. В следующем примере показано, что содержимое неизменяемого свойства ссылочного типа (в нашем примере это массив) является по сути изменяемым:

```
public record Person(string FirstName, string LastName, string[] PhoneNumbers);

public static void Main()
{
    Person person = new("Nancy", "Davolio", new string[1] { "555-1234" });
    Console.WriteLine(person.PhoneNumbers[0]); // output: 555-1234

    person.PhoneNumbers[0] = "555-6789";
    Console.WriteLine(person.PhoneNumbers[0]); // output: 555-6789
}
```

Возможности, уникальные для типов записей, реализуются синтезированными компилятором методами, ни один из которых не нарушает неизменяемость путем изменения состояния объекта. Если не указано иное, синтезированные методы создаются для объявлений `record`, `record struct` и `readonly record struct`.

Равенство значений

Для любого определяемого типа можно переопределить Object.Equals(Object) и перегрузить `operator ==`. Если не переопределять `Equals` или не перегружать `operator ==`, объявляемый тип указывает, как определяется равенство:

- Для типов `class` два объекта равны, если они ссылаются на один и тот же объект в памяти.
- Для типов `struct` два объекта равны, если хранят одинаковые значения и имеют один и тот же тип.
- Для типов `record`, включая `record struct` и `readonly record struct`, два объекта равны, если хранят одинаковые значения и имеют один и тот же тип.

Определение равенства для `record struct` совпадает с `struct`. Разница заключается в том, что для `struct` реализация находится в ValueType.Equals(Object) и зависит от отражения. Для записей реализация синтезируется компилятором и использует объявленные члены данных.

Для некоторых моделей данных требуется ссылочное равенство. Например, Entity Framework Core использует ссылочное равенство, чтобы гарантировать использование только одного экземпляра типа сущности в том случае, когда разные объекты концептуально являются одной сущностью. По этой причине записи и структуры записей не подходят для использования в качестве типов сущностей в Entity

Framework Core.

Следующий пример демонстрирует равенство значений для типов записей:

```
public record Person(string FirstName, string LastName, string[] PhoneNumbers);

public static void Main()
{
    var phoneNumbers = new string[2];
    Person person1 = new("Nancy", "Davolio", phoneNumbers);
    Person person2 = new("Nancy", "Davolio", phoneNumbers);
    Console.WriteLine(person1 == person2); // output: True

    person1.PhoneNumbers[0] = "555-1234";
    Console.WriteLine(person1 == person2); // output: True

    Console.WriteLine(ReferenceEquals(person1, person2)); // output: False
}
```

Чтобы реализовать равенство значений, компилятор синтезирует следующие методы:

- Переопределение Object.Equals(Object).

  Этот метод используется как основа для статического метода Object.Equals(Object, Object), если оба параметра имеют отличное от NULL значение.

- Виртуальный метод `Equals`, параметр которого является типом записи. Этот метод реализует IEquatable<T>.
- Переопределение Object.GetHashCode().
- Переопределения операторов `==` и `!=`.

Вы можете написать собственные реализации и заменить ими любой из синтезированных методов. Если тип записи имеет метод, совпадающий с сигнатурой какого-либо синтезируемого метода, компилятор будет его синтезировать.

Если вы предоставляете в типе записи собственную реализацию `Equals`, предоставьте также реализацию `GetHashCode`.

Обратимое изменение

Если нужно копировать экземпляр с изменениями, вы можете с помощью выражения `with` выполнить *обратимое изменение*. Выражение `with` создает новый экземпляр записи, который является копией существующего экземпляра записи, и изменяет в этой копии указанные свойства и поля. Для указания требуемых изменений используется синтаксис инициализатора объектов, как показано в следующем примере:

```
public record Person(string FirstName, string LastName)
{
    public string[] PhoneNumbers { get; init; }
}

public static void Main()
{
    Person person1 = new("Nancy", "Davolio") { PhoneNumbers = new string[1] };
    Console.WriteLine(person1);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers = System.String[] }

    Person person2 = person1 with { FirstName = "John" };
    Console.WriteLine(person2);
    // output: Person { FirstName = John, LastName = Davolio, PhoneNumbers = System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { PhoneNumbers = new string[1] };
    Console.WriteLine(person2);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers = System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { };
    Console.WriteLine(person1 == person2); // output: True
}
```

Выражение `with` может задавать позиционные свойства или свойства, созданные с помощью стандартного синтаксиса свойств. Непозиционные свойства должны иметь метод доступа `init` или `set`, чтобы их можно было изменять с помощью выражения `with`.

Результатом выражения `with` является *неполная копия*, то есть для ссылочных свойств в нее копируется только ссылка на экземпляр. В итоге исходная запись и ее копия указывают на один и тот же экземпляр.

Для реализации такой возможности для типов `record class` компилятор синтезирует метод клонирования и конструктор копии. Метод виртуального клонирования возвращает новую запись, инициализированную конструктором копий. При использовании выражения `with` компилятор создает код, который вызывает метод клона, а затем устанавливает указанные в выражении `with` свойства.

Если требуется другое поведение копирования, вы можете написать собственный конструктор копирования в `record class`. В этом случае компилятор не будет синтезировать конструктор. Присвойте конструктору атрибут `private`, если запись является `sealed`, или `protected` в противном случае. Компилятор не выполняет синтезирование конструктора копии для типов `record struct`. Можно написать его, но компилятор не будет создавать вызовы для выражений `with`. Вместо этого компилятор использует присваивание.

Вы не можете переопределить метод клонирования или создать другой элемент с именем `Clone` в любом типе записи. Фактическое имя метода клонирования создается компилятором.

Встроенное форматирование для отображения

Типы записей имеют создаваемый компилятором метод ToString, который отображает имена и значения открытых свойств и полей. Метод `ToString` возвращает строку в следующем формате:

```
<record type name> { <property name> = <value>, <property name> = <value>, ...}
```

Для ссылочных типов вместо значения свойства отображается имя типа того объекта, на который ссылается это свойство. В следующем примере массив имеет ссылочный тип, поэтому отображается `System.String[]` вместо фактических значений элементов массива:

```
Person { FirstName = Nancy, LastName = Davolio, ChildNames = System.String[] }
```

Для реализации этой функциональной возможности в типах `record class` компилятор синтезирует виртуальный метод `PrintMembers` и переопределение ToString. В типах `record struct` этот член является `private`. Переопределение `ToString` создает объект StringBuilder с именем типа, за которым следует открывающая квадратная скобка. Затем оно вызывает метод `PrintMembers`, который добавляет имена и значения свойств, а затем добавляет закрывающую скобку. В следующем примере показан код, аналогичный синтезированному переопределению:

```
public override string ToString()
{
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.Append("Teacher"); // type name
    stringBuilder.Append(" { ");
    if (PrintMembers(stringBuilder))
    {
        stringBuilder.Append(" ");
    }
    stringBuilder.Append("}");
    return stringBuilder.ToString();
}
```

Вы можете предоставить собственную реализацию `PrintMembers` или переопределения `ToString`. Примеры можно изучить в разделе Форматирование `PrintMembers` в производных записях далее в этой статье. В C# 10 и более поздних версиях ваша реализация `ToString` может включать модификатор `sealed`, который не позволяет компилятору создавать реализацию `ToString` для любых производных записей. Фактически это означает, что выходные данные `ToString` не будут содержать сведения о типе среды выполнения. (Отображаются все элементы и значения, поскольку в производных записях по-прежнему будет присутствовать метод PrintMembers.)

Наследование

Этот раздел относится только к типам `record class`.

Запись может наследовать от другой записи. Но запись не может наследовать от класса, а класс не может наследовать от записи.

Позиционные параметры в производных типах записей

Производная запись объявляет позиционные параметры для всех параметров, определенных в основном конструкторе базовой записи. Базовая запись объявляет и инициализирует эти свойства. Производная запись не скрывает их, но создает и инициализирует только свойства для параметров, которые не объявлены в базовой записи.

Следующий пример демонстрирует наследование с использованием синтаксиса позиционных свойств:

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, Grade = 3 }
}
```

Равенство в иерархиях наследования

Этот раздел относится к типам `record class`, но не к типам `record struct`. Чтобы две переменные записи считались равными, у них должен совпадать тип времени выполнения. При этом типы содержащихся в них переменных могут отличаться. В следующем примере кода показано сравнение на унаследованное равенство:

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Person student = new Student("Nancy", "Davolio", 3);
    Console.WriteLine(teacher == student); // output: False

    Student student2 = new Student("Nancy", "Davolio", 3);
    Console.WriteLine(student2 == student); // output: True
}
```

В этом примере все переменные объявляются как `Person`, даже если экземпляр является производным типом `Student` или `Teacher`. Все экземпляры имеют одинаковые свойства и одинаковые значения этих свойств. Но `student == teacher` возвращает `False`, хотя обе переменные имеют тип `Person`, а `student == student2` возвращает `True`, хотя одна переменная имеет тип `Person`, а другая — `Student`. Проверка на равенство зависит от типа фактического объекта в среде выполнения, а не от объявленного типа переменной.

Чтобы реализовать такое поведение, компилятор синтезирует свойство `EqualityContract`, которое возвращает объект Type, соответствующий типу записи. `EqualityContract` позволяет методам равенства учитывать тип времени выполнения при проверке объектов на равенство. Если запись имеет базовый тип `object`, это свойство получает атрибут `virtual`. Если базовый тип имеет другой тип записи, это свойство становится переопределением. Если тип записи имеет атрибут `sealed`, свойство также получает атрибут `sealed`.

При сравнении двух экземпляров производного типа синтезированные методы равенства проверяют на равенство все свойства базового и производного типов. Синтезированный метод `GetHashCode` использует метод `GetHashCode` всех свойств и полей, объявленных в базовом типе и в производном типе записи.

Выражения `with` в производных записях

Результат выражения `with` имеет тот же тип среды выполнения, что и операнд выражения. Копируются все свойства с типом времени выполнения, но изменять вы можете только свойства с типом времени компиляции, как показано в следующем примере:

```
public record Point(int X, int Y)
{
    public int Zbase { get; set; }
};
public record NamedPoint(string Name, int X, int Y) : Point(X, Y)
{
    public int Zderived { get; set; }
};

public static void Main()
{
    Point p1 = new NamedPoint("A", 1, 2) { Zbase = 3, Zderived = 4 };

    Point p2 = p1 with { X = 5, Y = 6, Zbase = 7 }; // Can't set Name or Zderived
    Console.WriteLine(p2 is NamedPoint);  // output: True
    Console.WriteLine(p2);
    // output: NamedPoint { X = 5, Y = 6, Zbase = 7, Name = A, Zderived = 4 }

    Point p3 = (NamedPoint)p1 with { Name = "B", X = 5, Y = 6, Zbase = 7, Zderived = 8 };
    Console.WriteLine(p3);
    // output: NamedPoint { X = 5, Y = 6, Zbase = 7, Name = B, Zderived = 8 }
}
```

Форматирование `PrintMembers` в производных записях

Синтезированный метод `PrintMembers` из производного типа записи вызывает базовую реализацию. Это означает, что в выходные данные `ToString` включаются все свойства и поля с атрибутом public, как в производных, так и базовых типах, как показано в следующем примере:

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);

public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, Grade = 3 }
}
```

Вы можете предоставить собственную реализацию метода `PrintMembers`. В этом случае используйте следующую сигнатуру:

- Для записи `sealed`, которая является производной от `object` (не объявляет базовую запись): `private bool PrintMembers(StringBuilder builder)`.
- Для записи `sealed`, которая является производной от другой записи: `protected sealed override bool PrintMembers(StringBuilder builder)`.
- Для записи, которая не является `sealed` и наследует от объекта: `protected virtual bool PrintMembers(StringBuilder builder);`.
- Для записи, которая не является `sealed` и наследует от другой записи: `protected override bool PrintMembers(StringBuilder builder);`.

Ниже приведен пример кода, который заменяет синтезированные методы `PrintMembers`: один пример для типа записи, которая наследует от объекта, и другой для типа записи, которая наследует от другой записи:

```
public abstract record Person(string FirstName, string LastName, string[] PhoneNumbers)
{
    protected virtual bool PrintMembers(StringBuilder stringBuilder)
    {
        stringBuilder.Append($"FirstName = {FirstName}, LastName = {LastName}, ");
        stringBuilder.Append($"PhoneNumber1 = {PhoneNumbers[0]}, PhoneNumber2 = {PhoneNumbers[1]}");
        return true;
    }
}

public record Teacher(string FirstName, string LastName, string[] PhoneNumbers, int Grade)
    : Person(FirstName, LastName, PhoneNumbers)
{
    protected override bool PrintMembers(StringBuilder stringBuilder)
    {
        if (base.PrintMembers(stringBuilder))
        {
            stringBuilder.Append(", ");
        };
        stringBuilder.Append($"Grade = {Grade}");
        return true;
    }
};

public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", new string[2] { "555-1234", "555-6789" }, 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, PhoneNumber1 = 555-1234, PhoneNumber2 = 555-
6789, Grade = 3 }
}
```

NOTE

В C# 10 и более поздних версиях компилятор будет синтезировать `PrintMembers`, если базовая запись запечатала метод `ToString`. Вы также можете создать собственную реализацию `PrintMembers`.

Поведение деконструктора в производных записях

Метод `Deconstruct` производной записи возвращает значения всех позиционных свойств с типом времени компиляции. Если переменная имеет тип базовой записи, деконструкция выполняется только для свойств базовой записи, если объект не приведен к производному типу. Следующий пример демонстрирует вызов деконструктора для производной записи.

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);

public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    var (firstName, lastName) = teacher; // Doesn't deconstruct Grade
    Console.WriteLine($"{firstName}, {lastName}");// output: Nancy, Davolio

    var (fName, lName, grade) = (Teacher)teacher;
    Console.WriteLine($"{fName}, {lName}, {grade}");// output: Nancy, Davolio, 3
}
```

Общие ограничения

Не существует общих ограничений, по которым тип обязан являться записью. Записи соответствуют ограничению `class` или `struct`. Чтобы задать ограничение для конкретной иерархии типов записей, задайте это ограничение для базовой записи, как обычно для базового класса. Дополнительные сведения см. в статье Ограничения параметров типа.

Спецификация языка C#

Дополнительные сведения см. в разделе о классах в спецификации языка C#.

Дополнительные сведения о функциях, появившихся в C# версии 9 и более поздних, см. в следующих заметках о функциях:

- Записи
- Методы задания init
- Ковариантные возвращаемые значения

См. также раздел

- справочник по C#
- Рекомендации по проектированию — выбор между классом и структурой
- Рекомендации по проектированию — проектирование структуры
- Система типов C#
- Выражение `with`

класс (Справочник по C#)

18.12.2021 • 2 minutes to read

Классы объявляются с помощью ключевого слова `class`, как показано в следующем примере:

```
class TestClass
{
    // Methods, properties, fields, events, delegates
    // and nested classes go here.
}
```

Комментарии

В C# допускается только одиночное наследование. Другими словами, класс может наследовать реализацию только от одного базового класса. Однако класс может реализовывать несколько интерфейсов. В таблице ниже приведены примеры наследования класса и реализации интерфейса.

НАСЛЕДОВАНИЕ	ПРИМЕР
Нет	`class ClassA { }`
Single	`class DerivedClass : BaseClass { }`
Отсутствует, реализует два интерфейса	`class ImplClass : IFace1, IFace2 { }`
Одиночное, реализует один интерфейс	`class ImplDerivedClass : BaseClass, IFace1 { }`

Классы, объявленные непосредственно в пространстве имен и не вложенные в другие классы, могут быть открытыми или внутренними. По умолчанию классы являются `internal`.

Члены класса, включая вложенные классы, могут объявляться с типом доступа public, protected internal, protected, internal, private или private protected. По умолчанию члены имеют тип доступа `private`.

Дополнительные сведения см. в статье Модификаторы доступа.

Можно объявить универсальные классы, имеющие параметры типа. Дополнительные сведения см. в разделе Универсальные классы.

Класс может содержать объявления следующих членов:

- Конструкторы
- Константы
- Fields
- Методы завершения
- Методы
- Свойства
- Индексаторы

- Инструкции
- События
- Делегаты
- Классы
- Интерфейсы
- Типы структур
- Типы перечислений

Пример

В приведенном ниже примере показано объявление полей, конструкторов и методов класса. В нем также демонстрируется создание экземпляра объекта и печать данных экземпляра. В этом примере объявляются два класса. Первый класс, `Child`, содержит два частных поля (`name` и `age`), два общих конструктора и один общий метод. Второй класс, `StringTest`, используется для хранения `Main`.

```
class Child
{
    private int age;
    private string name;

    // Default constructor:
    public Child()
    {
        name = "N/A";
    }

    // Constructor:
    public Child(string name, int age)
    {
        this.name = name;
        this.age = age;
    }

    // Printing method:
    public void PrintChild()
    {
        Console.WriteLine("{0}, {1} years old.", name, age);
    }
}

class StringTest
{
    static void Main()
    {
        // Create objects by using the new operator:
        Child child1 = new Child("Craig", 11);
        Child child2 = new Child("Sally", 10);

        // Create an object using the default constructor:
        Child child3 = new Child();

        // Display results:
        Console.Write("Child #1: ");
        child1.PrintChild();
        Console.Write("Child #2: ");
        child2.PrintChild();
        Console.Write("Child #3: ");
        child3.PrintChild();
    }
}
/* Output:
    Child #1: Craig, 11 years old.
    Child #2: Sally, 10 years old.
    Child #3: N/A, 0 years old.
*/
```

Комментарии

Обратите внимание, что в предыдущем примере доступ к частным полям (`name` и `age`) возможен только с помощью общих методов класса `Child` . Например, имя ребенка нельзя напечатать из метода `Main` с помощью следующего оператора:

```
Console.Write(child1.name);   // Error
```

Получить доступ к закрытым членам класса `Child` из метода `Main` можно было бы лишь в том случае, если бы `Main` был членом класса.

Типы, объявленные в классе без модификатора доступа, по умолчанию являются `private`, поэтому члены данных в этом примере останутся `private`, если ключевые слова будут удалены.

Наконец, обратите внимание, что для объекта, созданного с помощью конструктора без параметров (`child3`), поле `age` по умолчанию инициализировано с нулевым значением.

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Ссылочные типы

interface (справочник по C#)

18.12.2021 • 2 minutes to read

Интерфейс определяет контракт. Любой `class` или `struct`, реализующий этот контракт, должен предоставлять реализацию для членов, определенных в интерфейсе. Начиная с C# 8.0, интерфейс может определять реализацию по умолчанию для членов. Он также может определять члены `static`, чтобы обеспечить единую реализацию для общих функциональных возможностей.

В следующем примере класс `ImplementationClass` должен реализовать метод с именем `SampleMethod`, не имеющий параметров и возвращающий значение `void`.

Дополнительные сведения и примеры см. в разделе Интерфейсы.

Пример интерфейса

```
interface ISampleInterface
{
    void SampleMethod();
}

class ImplementationClass : ISampleInterface
{
    // Explicit interface member implementation:
    void ISampleInterface.SampleMethod()
    {
        // Method implementation.
    }

    static void Main()
    {
        // Declare an interface instance.
        ISampleInterface obj = new ImplementationClass();

        // Call the member.
        obj.SampleMethod();
    }
}
```

Интерфейс может быть членом пространства имен или класса. Объявление интерфейса может содержать объявления (сигнатуры без реализации) следующих членов.

- Методы
- Свойства
- Индексаторы
- События

Эти предыдущие объявления членов обычно не содержат тело. Начиная с C# 8.0, член интерфейса может объявлять тело. Этот называется *реализацией по умолчанию*. Члены с телом позволяют интерфейсу предоставлять реализацию по умолчанию для классов и структур, которые не предоставляют реализацию с переопределением. Кроме того, начиная с C# 8.0, интерфейс может включать следующее.

- Константы
- Операторы

- Статический конструктор
- Вложенные типы
- Статические поля, методы, свойства, индексаторы и события
- Объявления членов с использованием синтаксиса явной реализации интерфейса.
- Явные модификаторы доступа (доступ по умолчанию — `public`).

Интерфейсы не могут содержать состояние экземпляра. Хотя статические поля теперь разрешены, поля экземпляров в интерфейсах запрещены. Автосвойства экземпляра не поддерживаются в интерфейсах, так как они неявно объявляют скрытое поле. Это правило оказывает незначительное воздействие на объявления свойств. В объявлении интерфейса следующий код не объявляет автоматически реализуемое свойство, как в `class` или `struct`. Вместо этого он объявляет свойство, которое не имеет реализации по умолчанию, но должно быть реализовано в любом типе, реализующем интерфейс.

```
public interface INamed
{
  public string Name {get; set;}
}
```

Интерфейс может наследовать от одного или нескольких базовых интерфейсов. Когда интерфейс переопределяет метод, реализованный в базовом интерфейсе, он должен использовать синтаксис явной реализации интерфейса.

Если список базовых типов содержит базовый класс и интерфейсы, базовый класс должен стоять первым в списке.

Класс, реализующий интерфейс, может явно реализовывать члены этого интерфейса. При явной реализации члена к нему можно получить доступ только через экземпляр интерфейса, но не через экземпляр класса. Кроме того, обращение к членам интерфейса по умолчанию можно осуществлять только через экземпляр интерфейса.

Дополнительные сведения о явной реализации интерфейса см. в статье Явная реализация интерфейса.

Пример реализации интерфейса

В следующем примере показана реализация интерфейса. В этом примере интерфейс содержит объявление свойства, а класс содержит реализацию. Любой экземпляр класса, который реализует `IPoint`, имеет целочисленные свойства `x` и `y`.

```
interface IPoint
{
   // Property signatures:
   int X
   {
      get;
      set;
   }

   int Y
   {
      get;
      set;
   }

   double Distance
   {
       get;
   }
}

class Point : IPoint
{
   // Constructor:
   public Point(int x, int y)
   {
      X = x;
      Y = y;
   }

   // Property implementation:
   public int X { get; set; }

   public int Y { get; set; }

   // Property implementation
   public double Distance =>
      Math.Sqrt(X * X + Y * Y);

}

class MainClass
{
   static void PrintPoint(IPoint p)
   {
      Console.WriteLine("x={0}, y={1}", p.X, p.Y);
   }

   static void Main()
   {
      IPoint p = new Point(2, 3);
      Console.Write("My Point: ");
      PrintPoint(p);
   }
}
// Output: My Point: x=2, y=3
```

Спецификация языка C#

Дополнительные сведения см. в разделе Интерфейсы статьи Спецификация языка C#, а также в спецификации функций для элементов интерфейса по умолчанию — C# 8.0.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Ссылочные типы
- Интерфейсы
- Использование свойств
- Использование индексаторов

Ссылочные типы, допускающие значение NULL (справочник по C#)

18.12.2021 • 5 minutes to read

> **NOTE**
> В этой статье рассматриваются ссылочные типы, допускающие значение NULL. Вы также можете объявить типы значений, допускающие значение NULL.

Ссылочные типы, допускающие значение NULL, доступны начиная с C# 8.0, в коде, который дал явное согласие на *контекст, поддерживающий значение NULL*. Ссылочные типы, допускающие значение NULL, предупреждения о значении NULL при статическом анализе и оператор, опускающий NULL, являются необязательными функциями языка. По умолчанию все они отключены. *Контекст, допускающий значение NULL*, контролируется на уровне проекта с помощью параметров сборки или в коде с помощью директив pragma.

В контексте, поддерживающем значение NULL:

- переменная ссылочного типа `T` должна быть инициализирована со значением, отличным от NULL, и ей невозможно присвоить значение, которое может быть равно `null`;
- переменная ссылочного типа `T?` может быть инициализирована с помощью `null`, или ей может быть присвоено значение `null`, но она должна быть проверена на `null` перед разыменованием;
- переменная `m` типа `T?` считается не равной NULL при применении оператора, опускающего NULL, как в `m!`.

Различия между ссылочным типом, не допускающим значение NULL, `T` и ссылочным типом, допускающим значение NULL, `T?` проявляются при интерпретации предыдущих правил компилятором. Переменная типа `T` и переменная типа `T?` представлены одним и тем же типом .NET. В следующем примере объявляется строка, не допускающая значение NULL, и строка, допускающая значение NULL, а затем используется оператор, опускающий NULL, для присваивания значения строке, не допускающей значение NULL:

```
string notNull = "Hello";
string? nullable = default;
notNull = nullable!; // null forgiveness
```

Переменные `notNull` и `nullable` представлены типом String. Так как типы, допускающие и не допускающие значение NULL, хранятся в виде одного типа, существует несколько мест, где использование ссылочного типа, допускающего значение NULL, не допускается. Как правило, ссылочный тип, допускающий значение NULL, запрещено использовать в качестве базового класса или реализованного интерфейса. Ссылочный тип, допускающий значение NULL, не может использоваться в выражении проверки типа или создания объекта. Ссылочный тип, допускающий значение NULL, не может быть типом выражения доступа к члену. Эти конструкции показаны в следующих примерах:

```
public MyClass : System.Object? // not allowed
{
}

var nullEmpty = System.String?.Empty; // Not allowed
var maybeObject = new object?(); // Not allowed
try
{
    if (thing is string? nullableString) // not allowed
        Console.WriteLine(nullableString);
} catch (Exception? e) // Not Allowed
{
    Console.WriteLine("error");
}
```

Ссылки, допускающие значение NULL, и статический анализ

Примеры в предыдущем разделе иллюстрируют природу ссылочных типов, допускающих значение NULL. Ссылочные типы, допускающие значение NULL, не являются новыми типами классов, а обозначены заметками для существующих ссылочных типов. Компилятор использует эти заметки, чтобы помочь найти потенциальные ошибки для пустых ссылок в коде. Во время выполнения нет никакой разницы между ссылочным типом, не допускающим значение NULL, и ссылочным типом, допускающим значение NULL. Компилятор не добавляет никакую проверку для ссылочных типов, не допускающих значение NULL, во время выполнения. Преимущества заключаются в анализе времени компиляции. Компилятор создает предупреждения, помогающие находить и исправлять потенциальные ошибки со значениями NULL в коде. Вы объявляете свое намерение, и компилятор предупреждает вас, если код нарушает его.

В контексте, допускающем значение NULL, компилятор выполняет статический анализ для переменных любого ссылочного типа, как допускающего, так и не допускающего значение NULL. Компилятор отслеживает *состояние NULL* каждой ссылочной переменной в виде *не равно NULL* или *может быть NULL*. Состоянием по умолчанию для ссылки, не допускающей значение NULL, является *не равно NULL*. Состоянием по умолчанию для ссылки, допускающей значение NULL, является *может быть NULL*.

Ссылочные типы, не допускающие значение NULL, всегда должны быть безопасными для разыменования, так как их *состоянием NULL* является *не равно NULL*. Чтобы применить это правило, компилятор выдает предупреждения, если ссылочный тип, не допускающий значение NULL, не инициализируется со значением, отличным от NULL. Локальные переменные должны присваиваться там же, где они объявляются. Каждому полю должно быть присвоено значение, *не равное NULL*, в инициализаторе поля или в каждом конструкторе. Компилятор выдает предупреждения, если ссылка, не допускающая значение NULL, присваивается ссылке с состоянием *может быть NULL*. В целом, так как ссылка, не допускающая значение NULL, имеет состояние *не равно NULL*, при разыменовании этих переменных предупреждения не выдаются.

> **NOTE**
> При назначении выражения с состоянием *может быть NULL* для ссылочного типа, не допускающего значения NULL, компилятор создает предупреждения. Компилятор будет создавать предупреждения для этой переменной до тех пор, пока она не будет назначена выражению со значением *не равно NULL*.

Ссылочные типы, допускающие значение NULL, могут быть инициализированы или присвоены `null`. Таким образом, статический анализ должен определить, что переменная имеет состояние *не равно NULL*, до ее разыменования. Если ссылка, допускающая значение NULL, определяется как *может быть NULL*, при ее присвоении ссылочной переменной, не допускающей значение NULL, создается предупреждение компилятора. В следующем классе показаны примеры этих предупреждений:

```
public class ProductDescription
{
    private string shortDescription;
    private string? detailedDescription;

    public ProductDescription() // Warning! shortDescription not initialized.
    {
    }

    public ProductDescription(string productDescription) =>
        this.shortDescription = productDescription;

    public void SetDescriptions(string productDescription, string? details=null)
    {
        shortDescription = productDescription;
        detailedDescription = details;
    }

    public string GetDescription()
    {
        if (detailedDescription.Length == 0) // Warning! dereference possible null
        {
            return shortDescription;
        }
        else
        {
            return $"{shortDescription}\n{detailedDescription}";
        }
    }

    public string FullDescription()
    {
        if (detailedDescription == null)
        {
            return shortDescription;
        }
        else if (detailedDescription.Length > 0) // OK, detailedDescription can't be null.
        {
            return $"{shortDescription}\n{detailedDescription}";
        }
        return shortDescription;
    }
}
```

В следующем фрагменте кода показано, где компилятор выдает предупреждения при использовании этого класса:

```
string shortDescription = default; // Warning! non-nullable set to null;
var product = new ProductDescription(shortDescription); // Warning! static analysis knows shortDescription
maybe null.

string description = "widget";
var item = new ProductDescription(description);

item.SetDescriptions(description, "These widgets will do everything.");
```

В предыдущих примерах показан статический анализ компилятора для определения *состояния NULL* ссылочных переменных. Компилятор применяет правила языка для проверок и присваиваний, чтобы получить сведения для анализа. Компилятор не может делать предположения о семантике методов или свойств. При вызове методов, выполняющих проверки значений NULL, компилятор не может понять, что эти методы влияют на *состояние NULL* переменной. Существует ряд атрибутов, которые можно добавить в API, чтобы сообщить компилятору о семантике аргументов и возвращаемых значений. Эти атрибуты

применялись для многих общих API в библиотеках .NET Core. Например, IsNullOrEmpty был обновлен, и компилятор правильно интерпретирует этот метод как проверку значения NULL. Дополнительные сведения об атрибутах, применяемых для статического анализа *состояния NULL*, см. в статье Атрибуты, допускающие значение NULL.

Задание контекста, допускающего значение NULL

Существует два способа управления контекстом, допускающим значение NULL. На уровне проекта можно добавить параметр `<Nullable>enable</Nullable>`. В одном файле исходного кода C# можно добавить директиву pragma `#nullable enable`, чтобы включить контекст, допускающий значение NULL. См. статью о настройке стратегии, допускающей значение NULL. В версиях, предшествовавших версии .NET 6, в новых проектах используется значение по умолчанию `<Nullable>disable</Nullable>`. Начиная с .NET 6 все файлы новых проектов содержат элемент `<Nullable>enable</Nullable>`.

Спецификация языка C#

Дополнительные сведения см. в следующих предложениях для спецификации языка C#:

- Ссылочные типы, допускающие значение NULL
- Ссылочные типы, допускающие значение NULL. Черновик спецификации

См. также

- справочник по C#
- Типы значений, допускающие значение NULL

Справочник по C#. void

18.12.2021 • 2 minutes to read

`void` можно использовать в качестве возвращаемого типа метода (или локальной функции) для определения того, что метод не возвращает значение.

```
public static void Display(IEnumerable<int> numbers)
{
    if (numbers is null)
    {
        return;
    }

    Console.WriteLine(string.Join(" ", numbers));
}
```

Вы также можете использовать `void` как ссылочный тип для объявления того, что тип указателя неизвестен. Дополнительные сведения см. в разделе Типы указателей.

Нельзя использовать `void` в качестве типа переменной.

См. также раздел

- справочник по C#
- System.Void

var (справочник по C#)

18.12.2021 • 2 minutes to read

Начиная с Visual C# 3 переменные, объявленные в области действия метода, получают неявный "тип" `var`. Неявно типизированные локальные переменные является строго типизированными, как если бы вы объявили тип самостоятельно, однако его определяет компилятор. Следующие два объявления `i` функционально эквивалентны:

```
var i = 10; // Implicitly typed.
int i = 10; // Explicitly typed.
```

IMPORTANT

При использовании `var` с включенными ссылочными типами, допускающими значение NULL, всегда подразумевается ссылочный тип, допускающий значение NULL, даже если тип выражения не допускает значения NULL. Анализ состояния NULL в компиляторе обеспечивает защиту от разыменования возможного значения `null`. Если переменная никогда не назначается выражению, которое может иметь значение NULL, компилятор не выдаст никаких предупреждений. Если переменная присваивается выражению, которое может иметь значение NULL, необходимо проверить, что оно не равно NULL, перед разыменованием, чтобы избежать предупреждений.

Ключевое слово `var` обычно используется с выражениями вызова конструктора. Использование `var` позволяет вам не повторять имя типа при объявлении переменной и создании экземпляра объекта, как показано в следующем примере:

```
var xs = new List<int>();
```

Начиная с C# 9.0 вы можете использовать выражение `new` с целевым типом в качестве альтернативы:

```
List<int> xs = new();
List<int>? ys = new();
```

При сопоставлении шаблонов в шаблоне `var` используется ключевое слово `var`.

Пример

В следующем примере показаны два выражения запросов. В первом выражении использование `var` разрешено, но не является обязательным, поскольку тип результата запроса может быть задан явно как `IEnumerable<string>`. Однако во втором выражении благодаря `var` результат может быть коллекцией анонимных типов, и имя этого типа доступно только для компилятора. Использование `var` делает создание нового класса для результата необязательным. Обратите внимание на то, что во втором примере переменная итерации `foreach``item` также типизирована неявно.

```
// Example #1: var is optional when
// the select clause specifies a string
string[] words = { "apple", "strawberry", "grape", "peach", "banana" };
var wordQuery = from word in words
                where word[0] == 'g'
                select word;

// Because each element in the sequence is a string,
// not an anonymous type, var is optional here also.
foreach (string s in wordQuery)
{
    Console.WriteLine(s);
}

// Example #2: var is required because
// the select clause specifies an anonymous type
var custQuery = from cust in customers
                where cust.City == "Phoenix"
                select new { cust.Name, cust.Phone };

// var must be used because each item
// in the sequence is an anonymous type
foreach (var item in custQuery)
{
    Console.WriteLine("Name={0}, Phone={1}", item.Name, item.Phone);
}
```

См. также

- справочник по C#
- Неявно типизированные локальные переменные
- Связи типов в операциях запросов LINQ

Встроенные типы (справочник по C#)

18.12.2021 • 2 minutes to read

В следующей таблице приведены встроенные типы значений языка C#:

КЛЮЧЕВОЕ СЛОВО ТИПА C#	ТИП .NET
`bool`	System.Boolean
`byte`	System.Byte
`sbyte`	System.SByte
`char`	System.Char
`decimal`	System.Decimal
`double`	System.Double
`float`	System.Single
`int`	System.Int32
`uint`	System.UInt32
`nint`	System.IntPtr
`nuint`	System.UIntPtr
`long`	System.Int64
`ulong`	System.UInt64
`short`	System.Int16
`ushort`	System.UInt16

В следующей таблице приведены встроенные ссылочные типы языка C#:

КЛЮЧЕВОЕ СЛОВО ТИПА C#	ТИП .NET
`object`	System.Object
`string`	System.String
`dynamic`	System.Object

В таблицах выше каждое ключевое слово типа C# в левом столбце (кроме nint, nuint и dynamic) является

псевдонимом для соответствующего типа .NET. Они взаимозаменяемые. Например, следующие объявления объявляют переменные одного типа:

```
int a = 123;
System.Int32 b = 123;
```

Типы `nint` и `nuint` являются целыми числами собственного размера. В коде такие числа представлены определенными типами .NET, но в каждом случае ключевое слово и тип .NET не взаимозаменяемы. Для `nint` и `nuint` компилятор предоставляет преобразования и операции, как для целочисленных типов. Они отличаются от преобразований и операций для типов указателей `System.IntPtr` и `System.UIntPtr`. Дополнительные сведения см. в статье о типах `nint` и `nuint`.

Ключевое слово `void` представляет отсутствие типа. Оно используется в качестве возвращаемого типа метода, который не возвращает значение.

См. также

- справочник по C#
- Значения по умолчанию типов C#

Справочник по C#. Неуправляемые типы

18.12.2021 • 2 minutes to read

Тип является **неуправляемым типом**, если он принадлежит к одному из следующих типов:

- `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal` или `bool`;
- любой тип перечисления;
- любой тип указателя.
- Любой определенный пользователем тип структуры, который содержит поля только неуправляемых типов в C# 7.3 и более ранних версиях, не является сконструированным типом (тип, который включает по крайней мере один аргумент типа)

Начиная с C# 7.3 можно использовать ограничение `unmanaged`, чтобы указать, что параметр типа является отличным от указателя неуправляемым типом, не допускающим значения NULL.

Начиная с C# 8.0, *сконструированный* тип структуры, содержащий поля только неуправляемых типов, также является неуправляемым, как показано в следующем примере:

```
using System;

public struct Coords<T>
{
    public T X;
    public T Y;
}

public class UnmanagedTypes
{
    public static void Main()
    {
        DisplaySize<Coords<int>>();
        DisplaySize<Coords<double>>();
    }

    private unsafe static void DisplaySize<T>() where T : unmanaged
    {
        Console.WriteLine($"{typeof(T)} is unmanaged and its size is {sizeof(T)} bytes");
    }
}
// Output:
// Coords`1[System.Int32] is unmanaged and its size is 8 bytes
// Coords`1[System.Double] is unmanaged and its size is 16 bytes
```

Универсальная структура может быть источником как для неуправляемых сконструированных типов, так и для сконструированных типов, отличных от неуправляемых. В предыдущем примере определяется универсальная структура `Coords<T>`, а также представлены примеры неуправляемых сконструированных типов. Примером типа, отличного от неуправляемого, является `Coords<object>`. Он не является неуправляемым, так как содержит поля типа `object`, которые являются отличными от неуправляемых. Если необходимо, чтобы *все* сконструированные типы были неуправляемыми, используйте ограничение `unmanaged` в определении универсальной структуры:

```
public struct Coords<T> where T : unmanaged
{
    public T X;
    public T Y;
}
```

Спецификация языка C#

Дополнительные сведения см. в разделе Типы указателей в статье Спецификации языка C#.

См. также

- справочник по C#
- Типы указателей
- Типы, связанные с памятью и диапазонами
- Оператор sizeof
- stackalloc

Значения по умолчанию типов C# (справка по C#)

18.12.2021 • 2 minutes to read

В следующей таблице показаны значения по умолчанию для типов C#:

TYPE	ЗНАЧЕНИЕ ПО УМОЛЧАНИЮ
Любой ссылочный тип	`null`
Любой встроенный целочисленный тип	Ноль (0)
Любой встроенный тип с плавающей запятой	Ноль (0)
bool	`false`
char	`'\0'` (U+0000)
enum	Значение, создаваемое выражением `(E)0`, где `E` — это идентификатор перечисления.
struct	Значение, создаваемое путем установки значений по умолчанию для всех полей с типами значений и значений `null` для всех полей ссылочного типа.
Любой тип значения, допускающий значение NULL	Экземпляр, свойство `false` которого имеет значение HasValue, а свойство Value не определено. Это значение по умолчанию также называется значением *NULL* типа значения, допускающего значение NULL.

Выражения значения по умолчанию

Используйте оператор `default`, чтобы получить значение по умолчанию для типа, как показано в следующем примере:

```
int a = default(int);
```

Начиная с C# 7.1 вы можете использовать литерал `default` для инициализации переменной со значением по умолчанию для ее типа:

```
int a = default;
```

Конструктор без параметров для типа значения

Для типа значения *неявный* конструктор без параметров также создает значение по умолчанию для типа, как показано в следующем примере:

```
var n = new System.Numerics.Complex();
Console.WriteLine(n);  // output: (0, 0)
```

Если экземпляр System.Type представляет тип значения, во время выполнения можно использовать метод Activator.CreateInstance(Type), чтобы вызвать конструктор без параметров и получить значение типа по умолчанию.

> **NOTE**
> В C# 10 и более поздних версиях тип структуры (который является типом значения) может иметь явный конструктор без параметров, который может возвращать значение типа, отличное от значения по умолчанию. Рекомендуется использовать оператор `default` или литерал `default` для создания значения по умолчанию для типа.

Спецификация языка C#

Дополнительные сведения см. в следующих разделах статьи Спецификация языка C#:

- Значения по умолчанию
- Конструкторы по умолчанию

См. также

- справочник по C#
- Конструкторы

Ключевые слова C#

18.12.2021 • 2 minutes to read

Ключевые слова — это предварительно определенные зарезервированные идентификаторы, которые имеют специальные значения для компилятора. Их нельзя использовать как идентификаторы в программах без префикса `@`. Например, допустимым идентификатором является `@if`, но не `if`, поскольку `if` является ключевым словом.

В первой таблице этой статьи перечислены ключевые слова, которые нельзя использовать как идентификаторы в любой части программы C#. Во второй таблице этой статьи перечислены контекстные ключевые слова C#. Контекстные ключевые слова имеют особое значение только в определенном контексте программы, а за пределами этого контекста могут использоваться в качестве идентификаторов. Как правило, новые ключевые слова добавляются в язык C# в качестве контекстных ключевых слов, чтобы не нарушать работу программ, созданных в предыдущих версиях.

abstract
as
base
bool
break
byte
case
catch
char
checked
class
const
continue
decimal
default
delegate
do
double
else
enum

event
explicit
extern
false
finally
fixed
float
for
foreach
goto
if
implicit
in
int

interface
internal
is
lock
long

namespace
new
null
object
operator
out
override
params
private
protected
public
readonly
ref
return
sbyte
sealed
short
sizeof
stackalloc

static
string
struct
switch
this
throw
true
try
typeof
uint
ulong
unchecked
unsafe
ushort
using
virtual
void
volatile
while

Контекстные ключевые слова

Контекстное ключевое слово используется для предоставления конкретного значения в коде, но оно не является зарезервированным словом в C#. Некоторые контекстные ключевые слова, например `partial` и `where`, имеют особое значение в двух или более контекстах.

add

and
alias
ascending
async
await
by
descending
dynamic
equals
from

get
global
group
init
into
join
let
managed (соглашение о вызовах указателей функций)
nameof
nint
not (не);

notnull
nuint
on
или диспетчер конфигурации служб
orderby
partial (тип)
partial (метод)
record
remove
select

set
unmanaged (соглашение о вызовах указателей функций)
unmanaged (ограничение универсального типа)
value
var
when (условие фильтра)
where (ограничение универсального типа)
where (предложение запроса)
with
yield

См. также

- справочник по C#

Модификаторы доступа (Справочник по C#)

18.12.2021 • 2 minutes to read

Модификаторы доступа — это ключевые слова, которые задают объявленный уровень доступности члена или типа. В этом разделе описываются четыре модификатора доступа:

- `public`
- `protected`
- `internal`
- `private`

С помощью этих модификаторов можно задать следующие шесть уровней доступа:

- `public`: Неограниченный доступ.
- `protected`: Доступ ограничен содержащим классом или типами, которые являются производными от содержащего класса.
- `internal`: Доступ ограничен текущей сборкой.
- `protected internal`: Доступ ограничен текущей сборкой или типами, которые являются производными от содержащего класса.
- `private`: Доступ ограничен содержащим типом.
- `private protected`: Доступ ограничен содержащим классом или типами, которые являются производными от содержащего класса в текущей сборке.

В этом разделе также представлена следующая информация:

- Уровни доступности: с помощью четырех модификаторов доступа можно объявить шесть уровней доступности.
- Домен доступности: определяет, в каких разделах программы может присутствовать ссылка на этот член.
- Ограничения на использование уровней доступности: общие сведения об ограничениях на использование объявленных уровней доступности.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Модификаторы доступа
- Ключевые слова доступа
- Модификаторы

Уровни доступности (Справочник по C#)

18.12.2021 • 2 minutes to read

Модификаторы доступа `public`, `protected`, `internal` или `private` используются для указания одного из следующих объявленных уровней доступности к членам.

ОБЪЯВЛЕННАЯ ДОСТУПНОСТЬ	ЗНАЧЕНИЕ
`public`	Неограниченный доступ.
`protected`	Доступ ограничен содержащим классом или типами, которые являются производными от содержащего класса.
`internal`	Доступ ограничен текущей сборкой.
`protected internal`	Доступ ограничен текущей сборкой или типами, которые являются производными от содержащего класса.
`private`	Доступ ограничен содержащим типом.
`private protected`	Доступ ограничен содержащим классом или типами, которые являются производными от содержащего класса в текущей сборке. Доступно с версии C# 7.2.

Вы можете указать для члена или типа только один модификатор доступа, за исключением случаев использования сочетаний `protected internal` или `private protected`.

Модификаторы доступа не могут быть указаны для пространств имен. Пространства имен не имеют ограничений доступа.

В зависимости от контекста, в котором производится объявление члена, допускаются только некоторые объявленные уровни доступности. Если модификатор доступа не указывается в объявлении члена, используется доступность по умолчанию.

Типы верхнего уровня, не вложенные в другие типы, могут иметь только уровень доступности `internal` или `public`. Для этих типов уровнем доступности по умолчанию является `internal`.

Вложенные типы, которые являются членами других типов, могут иметь объявленные уровни доступности, как указано в следующей таблице.

ЧЛЕНЫ ТИПА	УРОВЕНЬ ДОСТУПНОСТИ ЧЛЕНОВ ПО УМОЛЧАНИЮ	ДОПУСТИМЫЕ ОБЪЯВЛЕННЫЕ УРОВНИ ДОСТУПНОСТИ ЧЛЕНА
`enum`	`public`	None

ЧЛЕНЫ ТИПА	УРОВЕНЬ ДОСТУПНОСТИ ЧЛЕНОВ ПО УМОЛЧАНИЮ	ДОПУСТИМЫЕ ОБЪЯВЛЕННЫЕ УРОВНИ ДОСТУПНОСТИ ЧЛЕНА

`class`	`private`	`public`
		`protected`
		`internal`
		`private`
		`protected internal`
		`private protected`

`interface`	`public`	`public`
		`protected`
		`internal`
		`private` *
		`protected internal`
		`private protected`

`struct`	`private`	`public`
		`internal`
		`private`

* Элемент `interface` с доступностью типа `private` должен иметь реализацию по умолчанию.

Доступность вложенного типа зависит от домена доступности, который определяется объявленным уровнем доступности члена и доменом доступности непосредственно вмещающего его типа. Однако домен доступности вложенного типа не может выходить за границы домена доступности содержащего его типа.

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Модификаторы доступа
- Домен доступности
- Ограничения на использование уровней доступности
- Модификаторы доступа
- public
- private

- protected
- internal

Область доступности (Справочник по C#)

18.12.2021 • 2 minutes to read

Область доступности члена определяет, в каких разделах программы может присутствовать ссылка на этот член. Если член вложен в другой тип, его область доступности определяется как уровнем доступности самого члена, так и областью доступности типа, непосредственно содержащего вложенный тип.

Область доступности типа верхнего уровня — это по крайней мере текст программы проекта, в котором он объявлен. То есть область включает в себя все исходные файлы данного проекта. Область доступности вложенного типа — это по крайней мере текст программы типа, в котором он объявлен. Таким образом, областью является тело типа, включающее все вложенные типы. Область доступности вложенного типа никогда не выходит за пределы области доступности содержащего его типа. Эти принципы продемонстрированы в приведенном ниже примере.

Пример

Этот пример содержит тип верхнего уровня `T1` и два вложенных класса: `M1` и `M2`. Классы содержат поля, имеющие различную объявленную доступность. В методе `Main` после каждого оператора следует комментарий, указывающий область доступности для каждого члена. Обратите внимание, что операторы, ссылающиеся на недоступные члены, закомментированы. Если необходимо просмотреть сообщения об ошибках, выдаваемые компилятором при попытке обращения к недоступному члену, удаляйте комментарии по одному.

```
public class T1
{
    public static int publicInt;
    internal static int internalInt;
    private static int privateInt = 0;

    static T1()
    {
        // T1 can access public or internal members
        // in a public or private (or internal) nested class.
        M1.publicInt = 1;
        M1.internalInt = 2;
        M2.publicInt = 3;
        M2.internalInt = 4;

        // Cannot access the private member privateInt
        // in either class:
        // M1.privateInt = 2; //CS0122
    }

    public class M1
    {
        public static int publicInt;
        internal static int internalInt;
        private static int privateInt = 0;
    }

    private class M2
    {
        public static int publicInt = 0;
        internal static int internalInt = 0;
        private static int privateInt = 0;
    }
```

```
}

class MainClass
{
    static void Main()
    {
        // Access is unlimited.
        T1.publicInt = 1;

        // Accessible only in current assembly.
        T1.internalInt = 2;

        // Error CS0122: inaccessible outside T1.
        // T1.privateInt = 3;

        // Access is unlimited.
        T1.M1.publicInt = 1;

        // Accessible only in current assembly.
        T1.M1.internalInt = 2;

        // Error CS0122: inaccessible outside M1.
        //    T1.M1.privateInt = 3;

        // Error CS0122: inaccessible outside T1.
        //    T1.M2.publicInt = 1;

        // Error CS0122: inaccessible outside T1.
        //    T1.M2.internalInt = 2;

        // Error CS0122: inaccessible outside M2.
        //    T1.M2.privateInt = 3;

        // Keep the console open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
```

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Модификаторы доступа
- Уровни доступности
- Ограничения на использование уровней доступности
- Модификаторы доступа
- public
- private
- protected
- internal

Ограничения на использование уровней доступности (справочник по C#)

18.12.2021 • 2 minutes to read

При задании типа в объявлении необходимо проверить, зависит ли уровень доступности типа от уровня доступности члена или другого типа. Например, прямой базовый класс должен иметь по крайней мере такой же уровень доступности, как и производный класс. Следующие объявления вызывают ошибку компиляции, так как базовый класс `BaseClass` менее доступен, чем `MyClass`:

```
class BaseClass {...}
public class MyClass: BaseClass {...} // Error
```

В таблице ниже приведены все ограничения на объявленные уровни доступности.

КОНТЕКСТ	КОММЕНТАРИИ
Классы	Прямой базовый класс для типа класса должен иметь по крайней мере такой же уровень доступности, как и сам тип класса.
Интерфейсы	Явные базовые интерфейсы для типа интерфейса должны иметь по крайней мере такой же уровень доступности, как и сам тип интерфейса.
Делегаты	Тип возвращаемого значения и типы параметров для типа делегата должны иметь по крайней мере такой же уровень доступности, как и сам тип делегата.
Константы	Тип константы должен иметь по крайней мере такой же уровень доступности, как и сама константа.
Fields	Тип поля должен иметь по крайней мере такой же уровень доступности, как и само поле.
Методы	Тип возвращаемого значения и типы параметров для метода должны иметь по крайней мере такой же уровень доступности, как и сам метод.
Свойства	Тип свойства должен иметь по крайней мере такой же уровень доступности, как и само свойство.
События	Тип события должен иметь по крайней мере такой же уровень доступности, как и само событие.
Индексаторы	Тип и типы параметров для индексатора должны иметь по крайней мере такой же уровень доступности, как и сам индексатор.
Операторы	Тип возвращаемого значения и типы параметров для оператора должны иметь по крайней мере такой же уровень доступности, как и сам оператор.

КОНТЕКСТ	КОММЕНТАРИИ
Конструкторы	Типы параметров для конструктора должны иметь по крайней мере такой же уровень доступности, как и сам конструктор.

Пример

В приведенном ниже примере содержатся ошибочные объявления различных типов. В комментарии после каждого объявления указывается предполагаемая ошибка компиляции.

```
// Restrictions on Using Accessibility Levels
// CS0052 expected as well as CS0053, CS0056, and CS0057
// To make the program work, change access level of both class B
// and MyPrivateMethod() to public.

using System;

// A delegate:
delegate int MyDelegate();

class B
{
    // A private method:
    static int MyPrivateMethod()
    {
        return 0;
    }
}

public class A
{
    // Error: The type B is less accessible than the field A.myField.
    public B myField = new B();

    // Error: The type B is less accessible
    // than the constant A.myConst.
    public readonly B myConst = new B();

    public B MyMethod()
    {
        // Error: The type B is less accessible
        // than the method A.MyMethod.
        return new B();
    }

    // Error: The type B is less accessible than the property A.MyProp
    public B MyProp
    {
        set
        {
        }
    }

    MyDelegate d = new MyDelegate(B.MyPrivateMethod);
    // Even when B is declared public, you still get the error:
    // "The parameter B.MyPrivateMethod is not accessible due to
    // protection level."

    public static B operator +(A m1, B m2)
    {
        // Error: The type B is less accessible
        // than the operator A.operator +(A,B)
        return new B();
    }

    static void Main()
    {
        Console.Write("Compiled successfully");
    }
}
```

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Модификаторы доступа
- Домен доступности
- Уровни доступности
- Модификаторы доступа
- public
- private
- protected
- internal

internal (Справочник по C#)

18.12.2021 • 2 minutes to read

Ключевое слово `internal` является модификатором доступа для типов и членов типов.

> Эта страница содержит доступ `internal`. Ключевое слово `internal` также является частью модификатора доступа `protected internal`.

Внутренние типы или члены доступны только внутри файлов в той же сборке, как в следующем примере:

```
public class BaseClass
{
    // Only accessible within the same assembly.
    internal static int x = 0;
}
```

Сравнение модификатора `internal` с другими модификаторами доступа см. в разделах Уровни доступности и Модификаторы доступа.

См. дополнительные сведения о сборках в .NET.

Обычно доступ к внутренним компонентам используется в разработке на основе компонентов, так как он позволяет группе компонентов взаимодействовать в закрытой форме, без их предоставления остальной части кода приложения. Например, платформа для создания графических интерфейсов пользователя может предоставлять классы `Control` и `Form`, взаимодействующие с помощью членов с внутренним доступом. Поскольку эти члены являются внутренними, они не предоставляются коду, использующему платформу.

Будет ошибкой ссылаться на тип или член с внутренним доступом из-за пределов сборки, в которой он определен.

Пример 1

Этот пример содержит два файла, `Assembly1.cs` и `Assembly1_a.cs`. Первый файл содержит внутренний базовый класс `BaseClass`. Во втором файле попытка создать экземпляр `BaseClass` приведет к ошибке.

```
// Assembly1.cs
// Compile with: /target:library
internal class BaseClass
{
   public static int intM = 0;
}
```

```
// Assembly1_a.cs
// Compile with: /reference:Assembly1.dll
class TestAccess
{
   static void Main()
   {
      var myBase = new BaseClass();   // CS0122
   }
}
```

Пример 2

В этом примере используйте те же файлы, которые использовались в примере 1, однако измените уровень доступности `BaseClass` на `public`. Кроме того, измените уровень доступности члена `intM` на `internal`. В этом случае можно создать экземпляр класса, но нельзя получить доступ к внутреннему члену.

```
// Assembly2.cs
// Compile with: /target:library
public class BaseClass
{
   internal static int intM = 0;
}
```

```
// Assembly2_a.cs
// Compile with: /reference:Assembly2.dll
public class TestAccess
{
   static void Main()
   {
      var myBase = new BaseClass();   // Ok.
      BaseClass.intM = 444;    // CS0117
   }
}
```

Спецификация языка C#

Дополнительные сведения см. в разделе Объявленная доступность в Спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Модификаторы доступа
- Уровни доступности
- Модификаторы
- public
- private
- protected

private (Справочник по C#)

18.12.2021 • 2 minutes to read

Ключевое слово `private` является модификатором доступа к члену.

> Эта страница содержит доступ `private`. Ключевое слово `private` также является частью модификатора доступа `private protected`.

Закрытый доступ является уровнем доступа с минимальными правами. Доступ к закрытым членам можно получить только внутри тела класса или структуры, в которой они объявлены, как показано в следующем примере:

```
class Employee
{
    private int _i;
    double _d;   // private access by default
}
```

Вложенные типы в том же теле могут также обращаться к таким закрытым членам.

Ошибка времени компиляции возникнет в том случае, если создать ссылку на закрытый член за пределами класса или структуры, в которой он объявлен.

Сравнение модификатора `private` с другими модификаторами доступа см. в разделах Уровни доступности и Модификаторы доступа.

Пример

В этом примере класс `Employee` содержит два закрытых элемента данных — `_name` и `_salary`. Как к закрытым членам, к ним нельзя получить доступ, кроме как через методы членов. Для получения управляемого доступа к закрытым членам можно использовать открытые методы `GetName` и `Salary`. Доступ к члену `_name` можно поучить через открытый метод, а к члену `_salary` — через открытое свойство только для чтения. (Дополнительные сведения см. в разделе Свойства.)

```
class Employee2
{
    private readonly string _name = "FirstName, LastName";
    private readonly double _salary = 100.0;

    public string GetName()
    {
        return _name;
    }

    public double Salary
    {
        get { return _salary; }
    }
}

class PrivateTest
{
    static void Main()
    {
        var e = new Employee2();

        // The data members are inaccessible (private), so
        // they can't be accessed like this:
        //    string n = e._name;
        //    double s = e._salary;

        // '_name' is indirectly accessed via method:
        string n = e.GetName();

        // '_salary' is indirectly accessed via property
        double s = e.Salary;
    }
}
```

Спецификация языка C#

Дополнительные сведения см. в разделе Объявленная доступность в Спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Модификаторы доступа
- Уровни доступности
- Модификаторы
- public
- protected
- internal

protected (справочник по C#)

18.12.2021 • 2 minutes to read

Ключевое слово `protected` является модификатором доступа к члену.

> **NOTE**
> Эта страница содержит доступ `protected`. Ключевое слово `protected` также является частью модификаторов доступа `protected internal` и `private protected`.

Доступ к защищенному элементу может быть получен из соответствующего класса, а также экземплярами производных классов.

Сравнение модификатора `protected` с другими модификаторами доступа см. в разделе Уровни доступности.

Пример 1

Доступ к защищенному элементу базового класса может быть получен в производном классе, только если доступ осуществляется через тип производного класса. Для примера рассмотрим следующий сегмент кода:

```
class A
{
    protected int x = 123;
}

class B : A
{
    static void Main()
    {
        var a = new A();
        var b = new B();

        // Error CS1540, because x can only be accessed by
        // classes derived from A.
        // a.x = 10;

        // OK, because this class derives from A.
        b.x = 10;
    }
}
```

Оператор `a.x = 10` вызывает ошибку, поскольку выполняется в статическом методе Main, а не в экземпляре класса Б.

Элементы структуры защитить нельзя, поскольку структура не может наследоваться.

Пример 2

В этом примере класс `DerivedPoint` является производным от класса `Point`. В связи с этим доступ к защищенным элементам базового класса можно получить напрямую из производного класса.

```
class Point
{
    protected int x;
    protected int y;
}

class DerivedPoint: Point
{
    static void Main()
    {
        var dpoint = new DerivedPoint();

        // Direct access to protected members.
        dpoint.x = 10;
        dpoint.y = 15;
        Console.WriteLine($"x = {dpoint.x}, y = {dpoint.y}");
    }
}
// Output: x = 10, y = 15
```

Если изменить уровни доступа `x` и `y` на private, компилятор выдаст сообщения об ошибках:

```
'Point.y' is inaccessible due to its protection level.
```

```
'Point.x' is inaccessible due to its protection level.
```

Спецификация языка C#

Дополнительные сведения см. в разделе Объявленная доступность в Спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Модификаторы доступа
- Уровни доступности
- Модификаторы
- public
- private
- internal
- Вопросы безопасности, связанные с использованием ключевых слов internal virtual

public (справочник по C#)

18.12.2021 • 2 minutes to read

Ключевое слово `public` является модификатором доступа для типов и членов типов. Общий доступ является уровнем доступа с максимальными правами. Ограничений доступа к общим членам не существует, как показано в следующем примере:

```
class SampleClass
{
    public int x; // No access restrictions.
}
```

Дополнительные сведения см. в разделах Модификаторы доступа и Уровни доступности.

Пример

В следующем примере объявляются два класса: `PointTest` и `Program`. Доступ к открытым членам `x` и `y` класса `PointTest` осуществляется непосредственно из класса `Program`.

```
class PointTest
{
    public int x;
    public int y;
}

class Program
{
    static void Main()
    {
        var p = new PointTest();
        // Direct access to public members.
        p.x = 10;
        p.y = 15;
        Console.WriteLine($"x = {p.x}, y = {p.y}");
    }
}
// Output: x = 10, y = 15
```

Если уровень доступа `public` изменить на private или protected, будет выводится следующее сообщение об ошибке:

"PointTest.y" недоступен из-за его уровня защиты.

Спецификация языка C#

Дополнительные сведения см. в разделе Объявленная доступность в Спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#

- Модификаторы доступа
- Ключевые слова в C#
- Модификаторы доступа
- Уровни доступности
- Модификаторы
- private
- protected
- internal

protected internal (справочник по C#)

18.12.2021 • 2 minutes to read

Комбинация ключевых слов `protected internal` является модификатором доступа к члену. Доступ к членам с модификатором доступа protected internal может осуществляться из текущей сборки или типов, которые являются производными от содержащего класса. Сравнение модификатора `protected internal` с другими модификаторами доступа см. в разделе Уровни доступности.

Пример

Член базового класса с модификатором доступа protected internal доступен из любого типа в пределах содержащей сборки. Он также доступен в производном классе, расположенном в другой сборке, только в том случае, если доступ осуществляется через переменную типа производного класса. Для примера рассмотрим следующий сегмент кода:

```
// Assembly1.cs
// Compile with: /target:library
public class BaseClass
{
   protected internal int myValue = 0;
}

class TestAccess
{
    void Access()
    {
        var baseObject = new BaseClass();
        baseObject.myValue = 5;
    }
}
```

```
// Assembly2.cs
// Compile with: /reference:Assembly1.dll
class DerivedClass : BaseClass
{
    static void Main()
    {
        var baseObject = new BaseClass();
        var derivedObject = new DerivedClass();

        // Error CS1540, because myValue can only be accessed by
        // classes derived from BaseClass.
        // baseObject.myValue = 10;

        // OK, because this class derives from BaseClass.
        derivedObject.myValue = 10;
    }
}
```

Этот пример содержит два файла, `Assembly1.cs` и `Assembly2.cs`. Первый файл содержит открытый базовый класс, `BaseClass`, и еще один класс, `TestAccess`. `BaseClass` владеет членом protected internal, `myValue`, доступ к которому осуществляется типом `TestAccess`. Во втором файле попытка получить доступ к `myValue` через экземпляр `BaseClass` приведет к ошибке во время доступа к этому члену через экземпляр производного класса. `DerivedClass` гарантирует успешное выполнение.

Элементы структуры не могут иметь модификатор `protected internal`, поскольку структура не может наследоваться.

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Модификаторы доступа
- Уровни доступности
- Модификаторы
- public
- private
- internal
- Вопросы безопасности, связанные с использованием ключевых слов internal virtual

private protected (справочник по C#)

18.12.2021 • 2 minutes to read

Комбинация ключевых слов `private protected` является модификатором доступа к члену. К члену private protected имеют доступ типы, производные от содержащего класса, но только в пределах содержащей сборки. Сравнение модификатора `private protected` с другими модификаторами доступа см. в разделе Уровни доступности.

> **NOTE**
> Модификатор доступа `private protected` допустим в C# 7.2 и более поздних версий.

Пример

Член базового класса private protected доступен из производных типов в содержащей сборке только в том случае, если статический тип переменной является типом производного класса. Для примера рассмотрим следующий сегмент кода:

```
public class BaseClass
{
    private protected int myValue = 0;
}

public class DerivedClass1 : BaseClass
{
    void Access()
    {
        var baseObject = new BaseClass();

        // Error CS1540, because myValue can only be accessed by
        // classes derived from BaseClass.
        // baseObject.myValue = 5;

        // OK, accessed through the current derived class instance
        myValue = 5;
    }
}
```

```
// Assembly2.cs
// Compile with: /reference:Assembly1.dll
class DerivedClass2 : BaseClass
{
    void Access()
    {
        // Error CS0122, because myValue can only be
        // accessed by types in Assembly1
        // myValue = 10;
    }
}
```

Этот пример содержит два файла, `Assembly1.cs` и `Assembly2.cs`. Первый файл содержит открытый базовый класс, `BaseClass`, и производный от него тип, `DerivedClass1`. `BaseClass` владеет членом private protected, `myValue`, к которому `DerivedClass1` пытается получить доступ двумя способами. Первая

попытка доступа к `myValue` через экземпляр `BaseClass` приведет к ошибке. Однако попытка использовать его в качестве наследуемого члена в `DerivedClass1` завершится успешно.

Во втором файле попытка получить доступ к `myValue` в качестве наследуемого члена `DerivedClass2` приведет к ошибке, поскольку он доступен только для производных типов в Assembly1.

Если `Assembly1.cs` содержит InternalsVisibleToAttribute с именем `Assembly2`, производный класс `DerivedClass2` будет иметь доступ к членам `private protected`, объявленным в `BaseClass`. `InternalsVisibleTo` делает члены `private protected` видимыми для производных классов в других сборках.

Элементы структуры не могут иметь модификатор `private protected`, поскольку структура не может наследоваться.

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Модификаторы доступа
- Уровни доступности
- Модификаторы
- public
- private
- internal
- Вопросы безопасности, связанные с использованием ключевых слов internal virtual

abstract (Справочник по C#)

18.12.2021 • 3 minutes to read

Модификатор `abstract` указывает, что изменяемый элемент имеет отсутствующую или неполную реализацию. Модификатор abstract можно использовать с классами, методами, свойствами, индексаторами и событиями. Используйте модификатор `abstract` в объявлении класса, чтобы указать, что класс предназначен только для использования в качестве базового класса для других классов и не должен быть создан сам по себе. Элементы с пометкой abstract должны быть реализованы не абстрактными классами, производными от абстрактного класса.

Пример 1

В этом примере класс `Square` должен обеспечивать реализацию `GetArea`, поскольку является производным от класса `Shape`:

```
abstract class Shape
{
    public abstract int GetArea();
}

class Square : Shape
{
    private int _side;

    public Square(int n) => _side = n;

    // GetArea method is required to avoid a compile-time error.
    public override int GetArea() => _side * _side;

    static void Main()
    {
        var sq = new Square(12);
        Console.WriteLine($"Area of the square = {sq.GetArea()}");
    }
}
// Output: Area of the square = 144
```

Абстрактные классы предоставляют следующие возможности:

- Создавать экземпляры абстрактного класса нельзя.
- Абстрактный класс может содержать абстрактные методы и методы доступа.
- Изменить абстрактный класс с модификатором sealed нельзя, так как два этих модификатора имеют взаимоисключающие значения. Модификатор `sealed` запрещает наследование класса, в то время как модификатор `abstract` указывает, что класс обязан иметь производные классы.
- Неабстрактный класс, производный от абстрактного класса, должен включать фактические реализации всех наследуемых абстрактных методов и методов доступа.

Модификатор `abstract` в объявлении метода или свойства позволяет указать, что этот метод или свойство не содержат реализации.

Абстрактные методы предоставляют следующие возможности:

- Абстрактный метод неявно представляет собой виртуальный метод.

- Объявления абстрактных методов допускаются только в абстрактных классах.
- Поскольку объявление абстрактного метода не предоставляет фактической реализации, тело метода отсутствует, а объявление метода заканчивается точкой с запятой, и фигурных скобок ({ }) после подписи нет. Пример:

  ```
  public abstract void MyMethod();
  ```

  Реализация предоставляется методом override, который является членом неабстрактного класса.
- В объявлении абстрактного метода нельзя использовать статические или виртуальные модификаторы.

Действие абстрактных свойств аналогично абстрактным методам, за исключением отличий в синтаксисе объявлений и вызовов.

- Использование модификатора `abstract` в статическом свойстве является недопустимым.
- Абстрактное наследуемое свойство можно переопределить в производном классе, включив объявление свойства, которое использует модификатор override.

Дополнительные сведения об абстрактных классах см. в разделе Абстрактные и запечатанные классы и члены классов.

Абстрактный класс должен предоставлять реализацию для всех членов интерфейса.

Абстрактный класс, реализующий интерфейс, может сопоставлять методы интерфейса с абстрактными методами. Пример:

```
interface I
{
    void M();
}

abstract class C : I
{
    public abstract void M();
}
```

Пример 2

В следующем примере класс `DerivedClass` является производным от абстрактного класса `BaseClass`. Абстрактный класс содержит абстрактный метод, `AbstractMethod`, и два абстрактных свойства, `X` и `Y`.

```
// Abstract class
abstract class BaseClass
{
    protected int _x = 100;
    protected int _y = 150;

    // Abstract method
    public abstract void AbstractMethod();

    // Abstract properties
    public abstract int X { get; }
    public abstract int Y { get; }
}

class DerivedClass : BaseClass
{
    public override void AbstractMethod()
    {
        _x++;
        _y++;
    }

    public override int X   // overriding property
    {
        get
        {
            return _x + 10;
        }
    }

    public override int Y   // overriding property
    {
        get
        {
            return _y + 10;
        }
    }

    static void Main()
    {
        var o = new DerivedClass();
        o.AbstractMethod();
        Console.WriteLine($"x = {o.X}, y = {o.Y}");
    }
}
// Output: x = 111, y = 161
```

В предыдущем примере при попытке создать экземпляр абстрактного класса с помощью оператора вида:

```
BaseClass bc = new BaseClass();   // Error
```

Выдается сообщение об ошибке, указывающее, что компилятор не может создать экземпляр абстрактного класса BaseClass.

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#

- Руководство по программированию на C#
- Модификаторы
- virtual
- override
- Ключевые слова в C#

async (справочник по C#)

18.12.2021 • 4 minutes to read

Модификатор `async` позволяет указать, что метод, лямбда-выражение или анонимный метод является асинхронным. Если этот модификатор используется в методе или выражении, они называются *асинхронными методами*. Ниже приводится пример асинхронного метода с именем `ExampleMethodAsync`:

```
public async Task<int> ExampleMethodAsync()
{
    //...
}
```

Если вы только начали заниматься асинхронным программированием или не понимаете, как в асинхронном методе используется оператор `await` для выполнения потенциально долгих операций без блокировки потока вызывающего объекта, ознакомьтесь с общими сведениями в статье Асинхронное программирование с использованием ключевых слов async и await. Следующий код размещается внутри асинхронного метода и вызывает метод HttpClient.GetStringAsync:

```
string contents = await httpClient.GetStringAsync(requestUrl);
```

Асинхронный метод выполняется синхронным образом до тех пор, пока не будет достигнуто первое выражение `await`, после чего метод приостанавливается, пока не будет завершена ожидаемая задача. В то же время управление возвращается в вызывающий объект метода, как показано в примере в следующем разделе.

Если метод, который изменяется ключевым словом `async`, не содержит выражения или оператора `await`, метод выполняется синхронно. Компилятор выводит предупреждения обо всех асинхронных методах, не содержащих операторы `await`, поскольку такая ситуация может указывать на ошибку. См. раздел Предупреждение компилятора (уровень 1) CS4014.

Ключевое слово `async` — это контекстно-зависимо ключевое слово, которое является ключевым словом, когда оно изменяет метод, лямбда-выражение или анонимный метод. Во всех других контекстах он интерпретируется как идентификатор.

Пример

В следующем примере показана структура и поток управления между обработчиком асинхронных событий `StartButton_Click` и асинхронным методом `ExampleMethodAsync`. В результате выполнения этого асинхронного метода возвращается число символов на веб-странице. Код подходит для приложения Windows Presentation Foundation (WPF) или приложения для Магазина Windows Presentation Foundation (WPF), которые создаются в Visual Studio; см. комментарии к коду для настройки приложения.

Этот код можно выполнить в Visual Studio как приложение Windows Presentation Foundation (WPF) или приложение Магазина Windows. Вам понадобятся элементы управления типа "Кнопка" (`StartButton`) и "Текстовое поле" (`ResultsTextBox`). Не забудьте задать имена и обработчик, чтобы получить код следующего вида:

```
<Button Content="Button" HorizontalAlignment="Left" Margin="88,77,0,0" VerticalAlignment="Top" Width="75"
        Click="StartButton_Click" Name="StartButton"/>
<TextBox HorizontalAlignment="Left" Height="137" Margin="88,140,0,0" TextWrapping="Wrap"
         Text="&lt;Enter a URL&gt;" VerticalAlignment="Top" Width="310" Name="ResultsTextBox"/>
```

Выполнение кода в виде приложения WPF:

- Вставьте этот код в класс `MainWindow` в MainWindow.xaml.cs.
- Добавьте ссылку на System.Net.Http.
- Добавьте директиву `using` для System.Net.Http.

Выполнение кода в виде приложения Магазина Windows:

- Вставьте этот код в класс `MainPage` в MainPage.xaml.cs.
- Добавьте директивы using для System.Net.Http и System.Threading.Tasks.

```
private async void StartButton_Click(object sender, RoutedEventArgs e)
{
    // ExampleMethodAsync returns a Task<int>, which means that the method
    // eventually produces an int result. However, ExampleMethodAsync returns
    // the Task<int> value as soon as it reaches an await.
    ResultsTextBox.Text += "\n";

    try
    {
        int length = await ExampleMethodAsync();
        // Note that you could put "await ExampleMethodAsync()" in the next line where
        // "length" is, but due to when '+=' fetches the value of ResultsTextBox, you
        // would not see the global side effect of ExampleMethodAsync setting the text.
        ResultsTextBox.Text += String.Format("Length: {0:N0}\n", length);
    }
    catch (Exception)
    {
        // Process the exception if one occurs.
    }
}

public async Task<int> ExampleMethodAsync()
{
    var httpClient = new HttpClient();
    int exampleInt = (await httpClient.GetStringAsync("http://msdn.microsoft.com")).Length;
    ResultsTextBox.Text += "Preparing to finish ExampleMethodAsync.\n";
    // After the following return statement, any method that's awaiting
    // ExampleMethodAsync (in this case, StartButton_Click) can get the
    // integer result.
    return exampleInt;
}
// The example displays the following output:
// Preparing to finish ExampleMethodAsync.
// Length: 53292
```

IMPORTANT

Дополнительные сведения о задачах и коде, который выполняется во время ожидания задачи, см. в разделе Асинхронное программирование с использованием ключевых слов async и await. Полный пример консоли, в котором используются похожие элементы, см. в разделе Обработка асинхронных задач по мере их завершения (C#).

Типы возвращаемых значений

Асинхронные методы могут иметь следующие типы возвращаемых значений:

- Task
- Task<TResult>
- void. Методы `async void` обычно рекомендуются для кода, отличного от обработчиков событий, поскольку вызывающие объекты не могут `await` эти методы и должны реализовать другой механизм уведомления об успешном завершении или ошибках.
- Начиная с версии 7.0 в языке C# поддерживаются любые типы с доступным методом `GetAwaiter`. Одной из таких реализаций является тип `System.Threading.Tasks.ValueTask<TResult>`. Доступ к нему осуществляется посредством пакета NuGet `System.Threading.Tasks.Extensions`.

Асинхронный метод не может объявлять параметры in, ref или out, а также иметь ссылочное возвращаемое значение, но он может вызывать методы с такими параметрами.

`Task<TResult>` указывается в качестве возвращаемого типа асинхронного метода, если оператор return метода задает операнд типа `TResult`. Класс `Task` используется при отсутствии содержательного значения, возвращаемого методом при его завершении. То есть вызов метода возвращает `Task`, однако когда `Task` завершен, любое выражение `await`, которое ожидает `Task`, возвращает значение `void`.

Возвращаемый тип `void` используется в основном для определения обработчиков событий, которые требуют этого возвращаемого типа. Вызывающий объект асинхронного метода, возвращающего `void`, не может ожидать его и перехватывать создаваемые методом исключения.

Начиная с версии C# 7.0, возвращается другой тип, обычно тип значения, с методом `GetAwaiter`, что позволяет свести к минимуму операции выделения памяти в разделах кода с критическими требованиями к производительности.

Дополнительные сведения и примеры см. в разделе Асинхронные типы возвращаемых значений.

См. также

- AsyncStateMachineAttribute
- await
- Асинхронное программирование с использованием ключевых слов async и await
- Обработка асинхронных задач по мере завершения

const (Справочник по C#)

18.12.2021 • 2 minutes to read

Для объявления константного поля или константной локальной используется ключевое слово `const`. Константные поля и локальные не являются переменными и не могут быть изменены. Константы могут быть числами, логическими значениями, строками или нулевыми ссылками. Не создавайте константу для предоставления сведений, которые могут измениться в любое время. Например, не используйте константное поле для хранения цены услуги, номера версии продукта или торгового названия компании. Эти значения могут со временем измениться, а поскольку константы распространяются компиляторами, для отражения изменений потребуется повторная компиляция остальных кодов, скомпилированных с использованием ваших библиотек. См. также описание ключевого слова readonly. Пример:

```
const int X = 0;
public const double GravitationalConstant = 6.673e-11;
private const string ProductName = "Visual C#";
```

Начиная с C# 10, интерполированные строки могут быть константами, если все используемые выражения также являются константными строками. Эта функция может улучшить код, создающий константные строки:

```
const string Language = "C#";
const string Platform = ".NET";
const string Version = "10.0";
const string FullProductName = $"{Platform} - Language: {Language} Version: {Version}";
```

Remarks

Тип объявления константы указывает на тип членов, которые вводятся объявлением. Инициализатор локальной константы или константного поля должен быть выражением константы, поддерживающим неявное преобразование в конечный тип.

Выражение константы — это выражение, которое можно полностью вычислить во время компиляции. Таким образом, единственно возможными значениями для констант типов ссылок являются `string` и нулевые ссылки.

Объявление константы может объявлять несколько констант, например:

```
public const double X = 1.0, Y = 2.0, Z = 3.0;
```

Модификатор `static` в объявлении константы не допускается.

Константа может участвовать в выражении константы следующим образом:

```
public const int C1 = 5;
public const int C2 = C1 + 100;
```

NOTE

Ключевое слово readonly отличается от ключевого слова `const`. Поле `const` может быть инициализировано только при объявлении поля. Поле `readonly` может быть инициализировано при объявлении или в конструкторе. Таким образом, поля `readonly` могут иметь разные значения в зависимости от использованного конструктора. Также, несмотря на то, что поле `const` является константой во время компиляции, поле `readonly` можно использовать для констант во время выполнения, как в следующей строке:
`public static readonly uint l1 = (uint)DateTime.Now.Ticks;`

Примеры

```
public class ConstTest
{
    class SampleClass
    {
        public int x;
        public int y;
        public const int C1 = 5;
        public const int C2 = C1 + 5;

        public SampleClass(int p1, int p2)
        {
            x = p1;
            y = p2;
        }
    }

    static void Main()
    {
        var mC = new SampleClass(11, 22);
        Console.WriteLine($"x = {mC.x}, y = {mC.y}");
        Console.WriteLine($"C1 = {SampleClass.C1}, C2 = {SampleClass.C2}");
    }
}
/* Output
    x = 11, y = 22
    C1 = 5, C2 = 10
*/
```

В этом примере показан способ использования констант в качестве локальных переменных.

```
public class SealedTest
{
    static void Main()
    {
        const int C = 707;
        Console.WriteLine($"My local constant = {C}");
    }
}
// Output: My local constant = 707
```

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Модификаторы
- readonly

Справочник по C#. event

18.12.2021 • 2 minutes to read

Ключевое слово `event` используется для объявления события в классе Publisher.

Пример

Следующий пример демонстрирует объявление и вызов события, которое использует EventHandler как базовый тип делегата. Полный пример кода, демонстрирующий использование универсального типа делегата EventHandler<TEventArgs>, создание подписки на событие и создание метода обработчика событий, см. в статье Публикация событий, соответствующих рекомендациям .NET.

```
public class SampleEventArgs
{
    public SampleEventArgs(string text) { Text = text; }
    public string Text { get; } // readonly
}

public class Publisher
{
    // Declare the delegate (if using non-generic pattern).
    public delegate void SampleEventHandler(object sender, SampleEventArgs e);

    // Declare the event.
    public event SampleEventHandler SampleEvent;

    // Wrap the event in a protected virtual method
    // to enable derived classes to raise the event.
    protected virtual void RaiseSampleEvent()
    {
        // Raise the event in a thread-safe manner using the ?. operator.
        SampleEvent?.Invoke(this, new SampleEventArgs("Hello"));
    }
}
```

События представляют собой специальный вид многоадресного делегата, который можно вызвать только из класса или структуры, в которых он объявлен (класс Publisher). Если другие классы или структуры подписываются на событие, их методы обработчиков событий будут вызываться, когда класс Publisher будет вызывать событие. Дополнительные сведения и примеры кода см. в разделах События и Делегаты.

События могут иметь пометку public, private, protected, internal, protected internal или private protected. Эти модификаторы доступа определяют, каким образом пользователи класса смогут получать доступ к событию. Дополнительные сведения см. в статье Модификаторы доступа.

Ключевые слова и события

Следующие ключевые слова применяются к событиям.

КЛЮЧЕВОЕ СЛОВО	ОПИСАНИЕ	ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ

КЛЮЧЕВОЕ СЛОВО	ОПИСАНИЕ	ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ
static	Делает событие доступным для вызывающих объектов в любое время, даже если экземпляр класса не существует.	Статические классы и члены статических классов
virtual	Позволяет производным классам переопределять поведение события с помощью ключевого слова override.	Наследование
sealed	Указывает, что для производных классов оно больше не является виртуальным.	
abstract	Компилятор не будет создавать блоки доступа к событиям `add` и `remove`, и поэтому производные классы должны предоставлять собственную реализацию.	

Событие может быть объявлено как статическое событие с помощью ключевого слова static. Это делает событие доступным для вызывающих объектов в любое время, даже если экземпляр класса не существует. Дополнительные сведения см. в статье Статические классы и члены статических классов.

Событие может быть помечено как виртуальное событие с помощью ключевого слова virtual. Это позволяет производным классам переопределять поведение события с помощью ключевого слова override. Дополнительные сведения см. в разделе Наследование. Событие, переопределяющее виртуальное событие, также может быть запечатанным (sealed), что указывает, что для производных классов оно больше не является виртуальным. И наконец, можно объявить событие абстрактным (abstract), что означает, что компилятор не будет создавать блоки доступа к событиям `add` и `remove`. Поэтому производные классы должны предоставлять собственную реализацию.

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- add
- remove
- Модификаторы
- Практическое руководство. Объединение делегатов (многоадресные делегаты)

extern (справочник по C#)

18.12.2021 • 2 minutes to read

Модификатор `extern` используется для объявления метода с внешней реализацией. При применении служб взаимодействия для вызова неуправляемого кода модификатор `extern` обычно используется с атрибутом `DllImport`. В этом случае также необходимо объявить метод как `static` в соответствии со следующим примером:

```
[DllImport("avifil32.dll")]
private static extern void AVIFileInit();
```

Ключевое слово `extern` может также определять внешний псевдоним сборки, который позволяет ссылаться на разные версии одного компонента из одной сборки. Дополнительные сведения см. в разделе Псевдоним extern.

Совместное использование модификаторов abstract и `extern` для изменения одного члена недопустимо. Использование модификатора `extern` означает, что метод реализуется вне кода C#, а применение модификатора `abstract` указывает на то, что в данном классе реализация метода не обеспечивается.

В C# ключевое слово extern имеет более ограниченное применение, чем в C++. Сравнительные характеристики использования этого ключевого слова в C# и в C++ см. в разделе "Использование extern для указания компоновки" Справочника по языку C++.

Пример 1

В этом примере программа получает от пользователя строку и отображает ее в окне сообщения. В этой программе используется метод `MessageBox`, импортированный из библиотеки User32.dll.

```
//using System.Runtime.InteropServices;
class ExternTest
{
    [DllImport("User32.dll", CharSet=CharSet.Unicode)]
    public static extern int MessageBox(IntPtr h, string m, string c, int type);

    static int Main()
    {
        string myString;
        Console.Write("Enter your message: ");
        myString = Console.ReadLine();
        return MessageBox((IntPtr)0, myString, "My Message Box", 0);
    }
}
```

Пример 2

В этом примере показана программа C#, в которой вызывается библиотека C (собственная библиотека DLL).

1. Создайте следующий файл C и назовите его `cmdll.c`.

```
// cmdll.c
// Compile with: -LD
int __declspec(dllexport) SampleMethod(int i)
{
  return i*10;
}
```

2. Откройте из каталога установки Visual Studio окно командной строки Visual Studio x64 (или x32) Native Tools и скомпилируйте файл `cmdll.c`, введя в командной строке **cl -LD cmdll.c**.

3. В том же каталоге создайте следующий файл C# и назовите его `cm.cs`.

```
// cm.cs
using System;
using System.Runtime.InteropServices;
public class MainClass
{
    [DllImport("Cmdll.dll")]
      public static extern int SampleMethod(int x);

    static void Main()
    {
        Console.WriteLine("SampleMethod() returns {0}.", SampleMethod(5));
    }
}
```

4. Откройте из каталога установки Visual Studio окно командной строки Visual Studio x64 (или x32) Native Tools и скомпилируйте файл `cm.cs`, введя:

   csc cm.cs (для командной строки x64) — или — **csc -platform:x86 cm.cs** (для командной строки x32)

   При этом будет создан исполняемый файл `cm.exe`.

5. Запустите `cm.exe`. Метод `SampleMethod` передает значение 5 в файл DLL, который возвращает значение, умноженное на 10. Программа выдает следующие результаты.

```
SampleMethod() returns 50.
```

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- System.Runtime.InteropServices.DllImportAttribute
- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Модификаторы

in (универсальный модификатор) (Справочник по C#)

18.12.2021 • 2 minutes to read

Для параметров универсального типа ключевое слово `in` указывает, что параметр типа является контравариантным. Ключевое слово `in` может применяться в универсальных интерфейсах и делегатах.

Контравариантность позволяет использовать производные типы со степенью наследования меньше, чем у типа, заданного универсальным параметром. Благодаря этому можно осуществлять неявное преобразование классов, реализующих контравариантные интерфейсы, и неявное преобразование типов делегатов. Ковариантность и контравариантность поддерживаются для ссылочных типов, но не для типов значений.

Тип может быть объявлен контравариантным в универсальном интерфейсе или делегате, только если он определяет тип параметров метода, но не тип значения, возвращаемого методом. Параметры `In`, `ref` и `out` должны быть инвариантными, то есть не являться ковариантными или контравариантными.

Интерфейс с параметром контравариантного типа позволяет своим методам принимать аргументы производных типов, степень наследования у которых меньше, чем у параметра типа интерфейса. Например, в интерфейсе IComparer<T> тип T является ковариантным, поэтому можно присвоить объект типа `IComparer<Person>` объекту типа `IComparer<Employee>` без применения каких-либо специальных методов преобразования, если `Employee` наследует `Person`.

Контравариантный делегат может быть назначен другому делегату того же типа, но с производным параметром универсального типа меньшей степени.

Дополнительные сведения см. в разделе Ковариация и контравариантность.

Контравариантный универсальный интерфейс

В следующем примере показано, как объявить, расширить и реализовать контравариантный универсальный интерфейс. В нем также показано, как можно использовать неявное преобразование для классов, реализующих этот интерфейс.

```
// Contravariant interface.
interface IContravariant<in A> { }

// Extending contravariant interface.
interface IExtContravariant<in A> : IContravariant<A> { }

// Implementing contravariant interface.
class Sample<A> : IContravariant<A> { }

class Program
{
    static void Test()
    {
        IContravariant<Object> iobj = new Sample<Object>();
        IContravariant<String> istr = new Sample<String>();

        // You can assign iobj to istr because
        // the IContravariant interface is contravariant.
        istr = iobj;
    }
}
```

Контравариантный универсальный делегат

В следующем примере кода показано, как объявить контравариантный универсальный делегат. В нем также показано, как можно выполнить неявное преобразование типа делегата.

```
// Contravariant delegate.
public delegate void DContravariant<in A>(A argument);

// Methods that match the delegate signature.
public static void SampleControl(Control control)
{ }
public static void SampleButton(Button button)
{ }

public void Test()
{

    // Instantiating the delegates with the methods.
    DContravariant<Control> dControl = SampleControl;
    DContravariant<Button> dButton = SampleButton;

    // You can assign dControl to dButton
    // because the DContravariant delegate is contravariant.
    dButton = dControl;

    // Invoke the delegate.
    dButton(new Button());
}
```

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- out
- Ковариация и контрвариантность

- Модификаторы

Модификатор new (справочник по C#)

18.12.2021 • 3 minutes to read

При использовании в качестве модификатора объявления ключевое слово `new` явным образом скрывает члены, унаследованные от базового класса. При скрытии унаследованного члена его производная версия заменяет версию базового класса. Предполагается, что версия базового класса элемента является видимой, так как она уже была бы скрыта, если бы была помечена как `private` или в некоторых случаях как `internal`. Хотя элементы `public` или `protected` можно скрывать без использования модификатора `new`, в этом случае выводится предупреждение компилятора. При использовании `new` для явного скрытия члена, предупреждение не появляется.

Ключевое слово `new` можно также использовать для создания экземпляра типа или как ограничение универсального типа.

Чтобы скрыть унаследованный член, объявите его в производном классе с использованием такого же имени члена и измените с помощью ключевого слова `new`. Пример:

```
public class BaseC
{
    public int x;
    public void Invoke() { }
}
public class DerivedC : BaseC
{
    new public void Invoke() { }
}
```

В этом примере `BaseC.Invoke` скрывается с помощью `DerivedC.Invoke`. Поле `x` не затрагивается, поскольку оно не скрыто таким же именем.

Скрытие имен через наследование принимает одну из следующих форм.

- Как правило, константы, поля, свойства и типы, которые вводятся в классе или структуре, скрывают все члены базового класса с таким же именем. Однако существуют особые случаи. Например, если объявить новое поле с именем `N`, чтобы сделать тип невызываемым, в то время как в базовом типе был объявлен метод `N`, новое поле не скрывает базовое объявление в синтаксисе вызова. Дополнительные сведения см. в разделе Поиск членов в спецификации языка C#.

- Метод, введенный в классе или структуре, скрывает свойства, поля и типы с тем же именем в базовом классе. Кроме того, он также скрывает все методы базового класса, имеющие такую же сигнатуру.

- Индексатор, представленный в классе или структуре, скрывает все индексаторы базового класса, имеющие одинаковую сигнатуру.

Совместное использование модификаторов `new` и override в одном члене является недопустимым, так как эти два модификатора имеют взаимоисключающие значения. Модификатор `new` создает новый член с таким же именем и приводит к скрытию исходного члена. Модификатор `override` расширяет реализацию для наследуемого члена.

При использовании модификатора `new` в объявлении, которое не скрывает наследуемый член, возникает предупреждение.

Примеры

В этом примере базовый класс `BaseC` и производный класс `DerivedC` используют одно и то же имя поля `x`, которое скрывает значение унаследованного поля. В примере показано использование модификатора `new`. Здесь также показано обращение к скрытым членам базового класса с помощью их полных имен.

```
public class BaseC
{
    public static int x = 55;
    public static int y = 22;
}

public class DerivedC : BaseC
{
    // Hide field 'x'.
    new public static int x = 100;

    static void Main()
    {
        // Display the new value of x:
        Console.WriteLine(x);

        // Display the hidden value of x:
        Console.WriteLine(BaseC.x);

        // Display the unhidden member y:
        Console.WriteLine(y);
    }
}
/*
Output:
100
55
22
*/
```

В этом примере вложенный класс скрывает класс, имеющий такое же имя в базовом классе. Здесь показано использование модификатора `new` для исключения предупреждений, а также обращение к членам скрытого класса с помощью их полных имен.

```
public class BaseC
{
    public class NestedC
    {
        public int x = 200;
        public int y;
    }
}

public class DerivedC : BaseC
{
    // Nested type hiding the base type members.
    new public class NestedC
    {
        public int x = 100;
        public int y;
        public int z;
    }

    static void Main()
    {
        // Creating an object from the overlapping class:
        NestedC c1  = new NestedC();

        // Creating an object from the hidden class:
        BaseC.NestedC c2 = new BaseC.NestedC();

        Console.WriteLine(c1.x);
        Console.WriteLine(c2.x);
    }
}
/*
Output:
100
200
*/
```

В случае удаления модификатора `new` программа продолжит компиляцию и выполнение, однако появится следующее предупреждение.

```
The keyword new is required on 'MyDerivedC.x' because it hides inherited member 'MyBaseC.x'.
```

Спецификация языка C#

Дополнительные сведения см. в разделе Модификатор new в спецификации языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Модификаторы
- Управление версиями с помощью ключевых слов Override и New
- Использование ключевых слов Override и New

out (универсальный модификатор) (Справочник по C#)

18.12.2021 • 2 minutes to read

Для параметров универсального типа ключевое слово `out` указывает, что параметр типа является ковариантным. Ключевое слово `out` может применяться в универсальных интерфейсах и делегатах.

Ковариация позволяет использовать производные типы со степенью наследования больше, нежели у типа, заданного универсальным параметром. Благодаря этому можно осуществлять неявное преобразование классов, реализующих ковариантные интерфейсы, и неявное преобразование типов делегатов. Ковариация и контравариантность поддерживаются для ссылочных типов, но не для типов значений.

Интерфейс с параметром ковариантного типа позволяет своим методам возвращать аргументы производных типов, степень наследования у которых больше, чем у параметра типа. Например, так как в .NET Framework 4 в IEnumerable<T> тип T является ковариантным, можно назначить объект типа `IEnumerable(Of String)` объекту типа `IEnumerable(Of Object)` без применения каких-либо специальных методов преобразования.

Ковариантный делегат может быть назначен другому делегату того же типа, но с производным параметром универсального типа большей степени.

Дополнительные сведения см. в разделе Ковариация и контравариантность.

Пример — ковариантный универсальный интерфейс

В следующем примере показано, как объявить, расширить и реализовать ковариантный универсальный интерфейс. В нем также показано, как использовать неявное преобразование для классов, реализующих ковариантный интерфейс.

```
// Covariant interface.
interface ICovariant<out R> { }

// Extending covariant interface.
interface IExtCovariant<out R> : ICovariant<R> { }

// Implementing covariant interface.
class Sample<R> : ICovariant<R> { }

class Program
{
    static void Test()
    {
        ICovariant<Object> iobj = new Sample<Object>();
        ICovariant<String> istr = new Sample<String>();

        // You can assign istr to iobj because
        // the ICovariant interface is covariant.
        iobj = istr;
    }
}
```

В универсальном интерфейсе параметр типа может быть объявлен ковариантным, если он удовлетворяет следующим условиям:

- Параметр типа используется только в качестве типа значения, возвращаемого методами интерфейса, и не используется в качестве типа аргументов метода.

  > **NOTE**
  > Существует одно исключение из данного правила. Если в ковариантном интерфейсе в качестве параметра метода используется контравариантный универсальный делегат, ковариантный тип можно использовать в качестве параметра универсального типа для этого делегата. Дополнительные сведения о ковариантных и контравариантных универсальных методах-делегатах см. в разделе Вариативность в делегатах и Использование вариативности в универсальных методах-делегатах Func и Action.

- Параметр типа не используется в качестве универсального ограничения для методов интерфейса.

Пример — ковариантный универсальный метод-делегат

В следующем примере кода показано, как объявить ковариантный универсальный метод-делегат. В нем также показано, как выполнить неявное преобразование типов делегатов.

```
// Covariant delegate.
public delegate R DCovariant<out R>();

// Methods that match the delegate signature.
public static Control SampleControl()
{ return new Control(); }

public static Button SampleButton()
{ return new Button(); }

public void Test()
{
    // Instantiate the delegates with the methods.
    DCovariant<Control> dControl = SampleControl;
    DCovariant<Button> dButton = SampleButton;

    // You can assign dButton to dControl
    // because the DCovariant delegate is covariant.
    dControl = dButton;

    // Invoke the delegate.
    dControl();
}
```

В универсальном методе-делегате тип может быть объявлен ковариантным, если он используется только как тип значения, возвращаемого методом, и не используется для аргументов метода.

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Вариативность в универсальных интерфейсах
- in
- Модификаторы

override (справочник по C#)

18.12.2021 • 2 minutes to read

Модификатор `override` требуется для расширения или изменения абстрактной или виртуальной реализации унаследованного метода, свойства, индексатора или события.

В следующем примере класс `Square` должен предоставить переопределенную реализацию `GetArea`, так как `GetArea` является унаследованным от абстрактного класса `Shape`:

```
abstract class Shape
{
    public abstract int GetArea();
}

class Square : Shape
{
    private int _side;

    public Square(int n) => _side = n;

    // GetArea method is required to avoid a compile-time error.
    public override int GetArea() => _side * _side;

    static void Main()
    {
        var sq = new Square(12);
        Console.WriteLine($"Area of the square = {sq.GetArea()}");
    }
}
// Output: Area of the square = 144
```

Метод `override` предоставляет новую реализацию метода, унаследованного от базового класса. Метод, переопределенный объявлением `override`, называется переопределенным базовым методом. Метод `override` должен иметь ту же сигнатуру, что и переопределенный базовый метод. Начиная с версии C# 9.0, методы `override` поддерживают ковариантные типы возвращаемых значений. В частности, тип возвращаемого значения метода `override` может быть производным от типа возвращаемого значения соответствующего базового метода. В C# 8.0 и более ранних версий типы возвращаемых значений метода `override` и переопределенного базового метода должны быть одинаковыми.

Невиртуальный или статический метод нельзя переопределить. Переопределенный базовый метод должен иметь тип `virtual`, `abstract` или `override`.

Объявление `override` не может изменить доступность метода `virtual`. Методы `override` и `virtual` должны иметь одинаковый модификатор уровня доступа.

Модификаторы `new`, `static` и `virtual` нельзя использовать для изменения метода `override`.

Переопределяющее объявление свойства должно задавать такие же модификатор уровня доступа, тип и имя, которые имеются у унаследованного свойства. Начиная с версии C# 9.0, переопределяющие свойства только для чтения поддерживают ковариантные типы возвращаемых значений. Переопределенное свойство должно быть `virtual`, `abstract` или `override`.

Дополнительные сведения об использовании ключевого слова `override` см. в разделах Управление версиями с помощью ключевых слов Override и New и Использование ключевых слов Override и New. Дополнительные сведения о наследовании см. в разделе Наследование.

Пример

В этом примере определяется базовый класс с именем `Employee` и производный класс с именем `SalesEmployee`. Класс `SalesEmployee` включает дополнительное поле `salesbonus`, для использования которого переопределяется метод `CalculatePay`.

```
class TestOverride
{
    public class Employee
    {
        public string Name { get; }

        // Basepay is defined as protected, so that it may be
        // accessed only by this class and derived classes.
        protected decimal _basepay;

        // Constructor to set the name and basepay values.
        public Employee(string name, decimal basepay)
        {
            Name = name;
            _basepay = basepay;
        }

        // Declared virtual so it can be overridden.
        public virtual decimal CalculatePay()
        {
            return _basepay;
        }
    }

    // Derive a new class from Employee.
    public class SalesEmployee : Employee
    {
        // New field that will affect the base pay.
        private decimal _salesbonus;

        // The constructor calls the base-class version, and
        // initializes the salesbonus field.
        public SalesEmployee(string name, decimal basepay, decimal salesbonus)
            : base(name, basepay)
        {
            _salesbonus = salesbonus;
        }

        // Override the CalculatePay method
        // to take bonus into account.
        public override decimal CalculatePay()
        {
            return _basepay + _salesbonus;
        }
    }

    static void Main()
    {
        // Create some new employees.
        var employee1 = new SalesEmployee("Alice", 1000, 500);
        var employee2 = new Employee("Bob", 1200);

        Console.WriteLine($"Employee1 {employee1.Name} earned: {employee1.CalculatePay()}");
        Console.WriteLine($"Employee2 {employee2.Name} earned: {employee2.CalculatePay()}");
    }
}
/*
    Output:
    Employee1 Alice earned: 1500
    Employee2 Bob earned: 1200
*/
```

Спецификация языка C#

Дополнительные сведения см. в разделе Методы переопределения статьи Спецификация языка C#.

Дополнительные сведения о ковариантных типах возвращаемых значений см. в примечании к функциям.

См. также

- справочник по C#
- Наследование
- Ключевые слова C#
- Модификаторы
- abstract
- virtual
- new (модификатор)
- Полиморфизм

readonly (Справочник по C#)

18.12.2021 • 3 minutes to read

Ключевое слово `readonly` — это модификатор, который может использоваться в четырех контекстах:

- В объявлении поля `readonly` указывает на то, что присвоение значения полю может происходить только при объявлении или в конструкторе этого класса. Полю только для чтения можно несколько раз назначить значения в объявлении поля и в конструкторе.

  Поле `readonly` нельзя изменять после выхода из конструктора. Это правило влечет за собой разные последствия для типов значений и ссылочных типов:

  - Поскольку типы значений содержат данные, поле `readonly` с типом значения является неизменяемым.
  - Ссылочные типы содержат только ссылку на соответствующие данные, а значит поле `readonly` ссылочного типа будет всегда ссылаться на один объект. Но сам этот объект не является неизменяемым. Модификатор `readonly` запрещает замену поля другим экземпляром ссылочного типа. Но этот модификатор не препятствует изменению данных экземпляра, на которое ссылается поле только для чтения, в том числе через это поле.

  > **WARNING**
  > Видимый извне тип, который содержит видимое извне и доступное только для чтения поле с изменяемым ссылочным типом, может представлять уязвимость и приводить к предупреждению CA2104: Не объявляйте изменяющиеся ссылочные типы только для чтения.

- В определении типа `readonly struct` объект `readonly` указывает на то, что тип структуры является неизменяемым. Дополнительные сведения см. в описании структуры `readonly` в статье Типы структур.
- В объявлении члена экземпляра в типе структуры `readonly` указывает на то, что член экземпляра не изменяет состояние структуры. Дополнительные сведения см. в разделе о членах экземпляров `readonly` в статье Типы структур.
- В возврате метода `ref readonly` модификатор `readonly` указывает, что метод возвращает ссылку, и записи для этой ссылки не допускаются.

Контексты `readonly struct` и `ref readonly` были добавлены в C# 7.2. Члены структуры `readonly` добавлены в C# 8.0

Пример поля только для чтения

В этом примере значение поля `year` нельзя изменить в методе `ChangeYear`, несмотря на то, что в конструкторе класса ему присваивается значение:

```
class Age
{
    private readonly int _year;
    Age(int year)
    {
        _year = year;
    }
    void ChangeYear()
    {
        //_year = 1967; // Compile error if uncommented.
    }
}
```

Можно присвоить значение полю `readonly` только в следующих контекстах:

- Когда переменная инициализируется в объявлении, например:

  ```
  public readonly int y = 5;
  ```

- В конструкторе экземпляра класса, содержащего объявление поля экземпляра.
- В статическом конструкторе класса, содержащего объявление статического поля.

Эти контексты конструктора являются единственными, в которых можно передавать поле `readonly` в качестве параметра out или ref.

NOTE

Ключевое слово `readonly` отличается от ключевого слова const. Поле `const` может быть инициализировано только при объявлении поля. Поле `readonly` может быть назначено несколько раз в объявлении поля и в любом конструкторе. Таким образом, поля `readonly` могут иметь разные значения в зависимости от использованного конструктора. К тому же, поскольку поле `const` является константой времени компиляции, поле `readonly` можно использовать для констант времени выполнения, как в следующем примере:

```
public static readonly uint timeStamp = (uint)DateTime.Now.Ticks;
```

```
public class SamplePoint
{
    public int x;
    // Initialize a readonly field
    public readonly int y = 25;
    public readonly int z;

    public SamplePoint()
    {
        // Initialize a readonly instance field
        z = 24;
    }

    public SamplePoint(int p1, int p2, int p3)
    {
        x = p1;
        y = p2;
        z = p3;
    }

    public static void Main()
    {
        SamplePoint p1 = new SamplePoint(11, 21, 32);   // OK
        Console.WriteLine($"p1: x={p1.x}, y={p1.y}, z={p1.z}");
        SamplePoint p2 = new SamplePoint();
        p2.x = 55;   // OK
        Console.WriteLine($"p2: x={p2.x}, y={p2.y}, z={p2.z}");
    }
    /*
     Output:
        p1: x=11, y=21, z=32
        p2: x=55, y=25, z=24
    */
}
```

В предыдущем примере при использовании такого оператора:

```
p2.y = 66;        // Error
```

будет отображено сообщение об ошибке компилятора:

Присваивание значений доступному только для чтения полю допускается только в конструкторе и в инициализаторе переменных.

Пример возвращаемой ссылки только для чтения

Модификатор `readonly` в `ref return` указывает, что возвращаемую ссылку нельзя изменить. Следующий пример возвращает ссылку на источник. Он использует модификатор `readonly`, чтобы указать, что вызывающие объекты не могут изменять источник:

```
private static readonly SamplePoint s_origin = new SamplePoint(0, 0, 0);
public static ref readonly SamplePoint Origin => ref s_origin;
```

Необязательно должен возвращаться тип `readonly struct`. Любой тип, возвращаемый из `ref`, может возвращаться из `ref readonly`.

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является

предписывающим источником информации о синтаксисе и использовании языка C#.

Вы также можете ознакомиться с предложениями языковых спецификаций:

- Ссылка и структура readonly
- Члены структуры readonly

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Модификаторы
- const
- Поля

sealed (Справочник по C#)

18.12.2021 • 2 minutes to read

При применении к классу модификатор `sealed` запрещает другим классам наследовать от этого класса. В следующем примере класс `B` наследует от класса `A`, но никакие классы не могут наследовать от класса `B`.

```
class A {}
sealed class B : A {}
```

Модификатор `sealed` можно использовать для метода или свойства, которое переопределяет виртуальный метод или свойство в базовом классе. Это позволяет классам наследовать от вашего класса, запрещая им при этом переопределять определенные виртуальные методы или свойства.

Пример

В следующем примере класс `Z` наследует от класса `Y`, но `Z` не может переопределить виртуальную функцию `F`, которая объявлена в классе `X` и запечатана в классе `Y`.

```
class X
{
    protected virtual void F() { Console.WriteLine("X.F"); }
    protected virtual void F2() { Console.WriteLine("X.F2"); }
}

class Y : X
{
    sealed protected override void F() { Console.WriteLine("Y.F"); }
    protected override void F2() { Console.WriteLine("Y.F2"); }
}

class Z : Y
{
    // Attempting to override F causes compiler error CS0239.
    // protected override void F() { Console.WriteLine("Z.F"); }

    // Overriding F2 is allowed.
    protected override void F2() { Console.WriteLine("Z.F2"); }
}
```

Чтобы предотвратить переопределение производных классов при определении новых методов или свойств, не назначайте их в качестве виртуальных (virtual).

Нельзя использовать модификатор abstract с запечатанным классом, поскольку абстрактный класс должен наследоваться классом, реализующим абстрактные методы или свойства.

При применении к методу или свойству модификатор `sealed` всегда следует использовать с override.

Поскольку структуры неявно запечатаны, их нельзя наследовать.

Дополнительные сведения см. в разделе Наследование.

Дополнительные примеры см. в разделе Абстрактные и запечатанные классы и члены классов.

```
sealed class SealedClass
{
    public int x;
    public int y;
}

class SealedTest2
{
    static void Main()
    {
        var sc = new SealedClass();
        sc.x = 110;
        sc.y = 150;
        Console.WriteLine($"x = {sc.x}, y = {sc.y}");
    }
}
// Output: x = 110, y = 150
```

Чтобы наследовать от запечатанного класса, можно применить следующую инструкцию в приведенном выше примере:

```
class MyDerivedC: SealedClass {} // Error
```

В результате выдается сообщение об ошибке:

```
'MyDerivedC': cannot derive from sealed type 'SealedClass'
```

Комментарии

Чтобы определить, нужно ли запечатывать класс, метод или свойство, имейте в виду следующее:

- потенциальные преимущества, которые производные классы могут получить от возможности настраивать ваш класс;
- вероятность того, что производные классы могут корректировать ваши классы, препятствуя их нормальной работе.

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Статические классы и члены статических классов
- Абстрактные и запечатанные классы и члены классов
- Модификаторы доступа
- Модификаторы
- override
- virtual

static (Справочник по C#)

18.12.2021 • 3 minutes to read

На этой странице приводятся сведения о ключевом слове модификатора `static`. Ключевое слово `static` также является частью директивы `using static`.

Модификатор `static` используется для объявления статического члена, принадлежащего собственно типу, а не конкретному объекту. Модификатор `static` можно использовать для объявления классов `static`. В классах, интерфейсах и структурах вы можете добавить модификатор `static` к полям, методам, свойствам, операторам, событиям и конструкторам. Модификатор `static` запрещено использовать с индексаторами или методами завершения. Дополнительные сведения см. в статье Статические классы и члены статических классов.

Начиная с C# 8.0 можно добавить модификатор `static` в локальную функцию. Статическая локальная функция не может сохранять локальные переменные или состояние экземпляра.

Начиная с C# 9.0 можно добавить модификатор `static` в лямбда-выражение или анонимный метод. Статическое лямбда-выражение или анонимный метод не могут сохранять локальные переменные или состояние экземпляра.

Пример: статический класс

Следующий класс объявляется как `static` и содержит только методы `static`:

```
static class CompanyEmployee
{
    public static void DoSomething() { /*...*/ }
    public static void DoSomethingElse() { /*...*/  }
}
```

Объявление константы или типа неявно является членом `static`. На член `static` невозможно ссылаться через экземпляр, а можно только через имя типа. Например, рассмотрим следующий класс.

```
public class MyBaseC
{
    public struct MyStruct
    {
        public static int x = 100;
    }
}
```

Чтобы обратиться к члену `static` `x`, воспользуйтесь полным именем — `MyBaseC.MyStruct.x` (если только член не доступен из той же области действия).

```
Console.WriteLine(MyBaseC.MyStruct.x);
```

Так как экземпляр класса содержит отдельную копию всех полей экземпляра класса, каждому полю `static` соответствует только одна копия.

Невозможно использовать `this` для ссылки на методы `static` или методы доступа к свойствам.

Если к классу применяется ключевое слово `static`, все члены этого класса должны быть `static`.

Классы, интерфейсы и классы `static` могут иметь конструкторы `static`. Конструктор `static` вызывается на определенном этапе между запуском программы и созданием экземпляра класса.

> **NOTE**
> Ключевое слово `static` имеет более ограниченное применение по сравнению с C++. Сведения о сравнении с ключевым словом C++ см. в статье Классы хранения (C++).

В качестве демонстрации членов `static` рассмотрим класс, представляющий сотрудника компании. Предположим, что этот класс содержит метод для подсчета сотрудников и поле для хранения их числа. И метод, и поле не принадлежат никакому экземпляру сотрудника. Они принадлежат всему классу сотрудников. В связи с этим они должны объявляться как члены `static` класса.

Пример: статическое поле и метод

В этом примере выполняется чтение имени и идентификатора нового сотрудника, увеличение счетчика сотрудников на единицу, а также отображение сведений о новом сотруднике и новом числе сотрудников. Эта программа считывает текущее число сотрудников с клавиатуры.

```
public class Employee4
{
    public string id;
    public string name;

    public Employee4()
    {
    }

    public Employee4(string name, string id)
    {
        this.name = name;
        this.id = id;
    }

    public static int employeeCounter;

    public static int AddEmployee()
    {
        return ++employeeCounter;
    }
}

class MainClass : Employee4
{
    static void Main()
    {
        Console.Write("Enter the employee's name: ");
        string name = Console.ReadLine();
        Console.Write("Enter the employee's ID: ");
        string id = Console.ReadLine();

        // Create and configure the employee object.
        Employee4 e = new Employee4(name, id);
        Console.Write("Enter the current number of employees: ");
        string n = Console.ReadLine();
        Employee4.employeeCounter = Int32.Parse(n);
        Employee4.AddEmployee();

        // Display the new information.
        Console.WriteLine($"Name: {e.name}");
        Console.WriteLine($"ID:   {e.id}");
        Console.WriteLine($"New Number of Employees: {Employee4.employeeCounter}");
    }
}
/*
Input:
Matthias Berndt
AF643G
15
 *
Sample Output:
Enter the employee's name: Matthias Berndt
Enter the employee's ID: AF643G
Enter the current number of employees: 15
Name: Matthias Berndt
ID:   AF643G
New Number of Employees: 16
*/
```

Пример: статическая инициализация

В этом примере показано, как можно инициализировать поле `static`, используя другое поле `static`, которое еще не объявлено. Результаты будут неопределенными до тех пор, пока вы явно не присвоите значение полю `static`.

```
class Test
{
    static int x = y;
    static int y = 5;

    static void Main()
    {
        Console.WriteLine(Test.x);
        Console.WriteLine(Test.y);

        Test.x = 99;
        Console.WriteLine(Test.x);
    }
}
/*
Output:
    0
    5
    99
*/
```

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Модификаторы
- Директива using static
- Статические классы и члены статических классов

unsafe (Справочник по C#)

18.12.2021 • 2 minutes to read

Ключевое слово `unsafe` обозначает небезопасный контекст, необходимый для выполнения любых операций с применением указателей. Дополнительные сведения см. в разделе Небезопасный код и указатели.

В объявлении типа или члена типа можно использовать модификатор `unsafe`. Все текстовое пространство типа или члена типа считается небезопасным контекстом. Например, следующий метод объявлен с модификатором `unsafe`:

```
unsafe static void FastCopy(byte[] src, byte[] dst, int count)
{
    // Unsafe context: can use pointers here.
}
```

Область небезопасного контекста простирается от списка параметров до конца метода, поэтому в списке параметров можно также использовать указатели:

```
unsafe static void FastCopy ( byte* ps, byte* pd, int count ) {...}
```

Кроме того, небезопасный блок позволяет добавлять небезопасный код в блок. Пример:

```
unsafe
{
    // Unsafe context: can use pointers here.
}
```

Для компиляции небезопасного кода нужно задать параметр **AllowUnsafeBlocks**. Небезопасный код не проверяется средой CLR.

Пример

```
// compile with: -unsafe
class UnsafeTest
{
    // Unsafe method: takes pointer to int.
    unsafe static void SquarePtrParam(int* p)
    {
        *p *= *p;
    }

    unsafe static void Main()
    {
        int i = 5;
        // Unsafe method: uses address-of operator (&).
        SquarePtrParam(&i);
        Console.WriteLine(i);
    }
}
// Output: 25
```

Спецификация языка C#

Дополнительные сведения см. в разделе Небезопасный код в Спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Оператор fixed
- Небезопасный код и указатели
- Буферы фиксированного размера

virtual (Справочник по C#)

18.12.2021 • 3 minutes to read

Ключевое слово `virtual` используется для изменения объявлений методов, свойств, индексаторов и событий и разрешения их переопределения в производном классе. Например, этот метод может быть переопределен любым наследующим его классом:

```
public virtual double Area()
{
    return x * y;
}
```

Реализацию виртуального члена можно изменить путем переопределения члена в производном классе. Дополнительные сведения об использовании ключевого слова `virtual` см. в разделах Управление версиями с помощью ключевых слов Override и New и Использование ключевых слов Override и New.

Remarks

При вызове виртуального метода тип времени выполнения объекта проверяется на переопределение члена. Вызывается переопределение члена в самом дальнем классе. Это может быть исходный член, если никакой производный класс не выполнял переопределение этого члена.

По умолчанию методы не являются виртуальными. Такой метод переопределить невозможно.

Нельзя использовать модификатор `virtual` с модификаторами `static`, `abstract`, `private`, или `override`. В следующем примере показано виртуальное свойство.

```
class MyBaseClass
{
    // virtual auto-implemented property. Overrides can only
    // provide specialized behavior if they implement get and set accessors.
    public virtual string Name { get; set; }

    // ordinary virtual property with backing field
    private int _num;
    public virtual int Number
    {
        get { return _num; }
        set { _num = value; }
    }
}

class MyDerivedClass : MyBaseClass
{
    private string _name;

    // Override auto-implemented property with ordinary property
    // to provide specialized accessor behavior.
    public override string Name
    {
        get
        {
            return _name;
        }
        set
        {
            if (!string.IsNullOrEmpty(value))
            {
                _name = value;
            }
            else
            {
                _name = "Unknown";
            }
        }
    }
}
```

Действие виртуальных свойств аналогично виртуальным методам, за исключением отличий в синтаксисе объявлений и вызовов.

- Использование модификатора `virtual` в статическом свойстве является недопустимым.
- Виртуальное наследуемое свойство может быть переопределено в производном классе путем включения объявления свойства, которое использует модификатор `override`.

Пример

В этом примере класс `Shape` содержит две координаты `x`, `y` и виртуальный метод `Area()`. Различные классы фигур, такие как `Circle`, `Cylinder` и `Sphere`, наследуют класс `Shape`, и для каждой фигуры вычисляется площадь поверхности. Каждый производный класс обладает собственной реализацией переопределения метода `Area()`.

Обратите внимание, что наследуемые классы `Circle`, `Sphere` и `Cylinder` используют конструкторы, которые инициализируют базовый класс, как показано в следующем объявлении.

```
public Cylinder(double r, double h): base(r, h) {}
```

Следующая программа вычисляет и отображает соответствующую область для каждой фигуры путем вызова нужной реализации метода `Area()` в соответствии с объектом, связанным с методом.

```
class TestClass
{
    public class Shape
    {
        public const double PI = Math.PI;
        protected double _x, _y;

        public Shape()
        {
        }

        public Shape(double x, double y)
        {
            _x = x;
            _y = y;
        }

        public virtual double Area()
        {
            return _x * _y;
        }
    }

    public class Circle : Shape
    {
        public Circle(double r) : base(r, 0)
        {
        }

        public override double Area()
        {
            return PI * _x * _x;
        }
    }

    public class Sphere : Shape
    {
        public Sphere(double r) : base(r, 0)
        {
        }

        public override double Area()
        {
            return 4 * PI * _x * _x;
        }
    }

    public class Cylinder : Shape
    {
        public Cylinder(double r, double h) : base(r, h)
        {
        }

        public override double Area()
        {
            return 2 * PI * _x * _x + 2 * PI * _x * _y;
        }
    }

    static void Main()
    {
        double r = 3.0, h = 5.0;
        Shape c = new Circle(r);
        Shape s = new Sphere(r);
        Shape l = new Cylinder(r, h);
```

```
        Shape l = new Cylinder(r, h);
        // Display results.
        Console.WriteLine("Area of Circle   = {0:F2}", c.Area());
        Console.WriteLine("Area of Sphere   = {0:F2}", s.Area());
        Console.WriteLine("Area of Cylinder = {0:F2}", l.Area());
    }
}
/*
Output:
Area of Circle   = 28.27
Area of Sphere   = 113.10
Area of Cylinder = 150.80
*/
```

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Полиморфизм
- abstract
- override
- new (модификатор)

volatile (Справочник по C#)

18.12.2021 • 2 minutes to read

Ключевое слово `volatile` означает, что поле может изменить несколько потоков, выполняемых одновременно. Компилятор, среда выполнения или даже аппаратное обеспечение могут изменять порядок операций чтения и записи в расположения в памяти для повышения производительности. Поля, которые объявлены `volatile`, исключаются из некоторых типов оптимизации. Нет никакой гарантии единого общего прядка временных записей во всех потоках выполнения. Дополнительные сведения см. в описании класса Volatile.

> **NOTE**
> В многопроцессорной системе переменные операции чтения не гарантируют получения последнего значения, записанного в эту область памяти любым из процессоров. Аналогичным образом, операция записи не гарантирует, что записываемое значение будет сразу отображаться для других процессоров.

Ключевое слово `volatile` может применяться к полям следующих типов:

- Ссылочные типы.
- Типы указателей (в небезопасном контексте). Несмотря на то, что сам указатель может быть изменяемым, объект, на который он указывает, должен быть постоянным. Другими словами, объявить указатель на изменяемый объект невозможно.
- Простые типы, например `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `char`, `float` и `bool`.
- Тип `enum` с одним из следующих базовых типов: `byte`, `sbyte`, `short`, `ushort`, `int` или `uint`.
- Параметры универсального типа называются ссылочными типами.
- IntPtr и UIntPtr.

Другие типы, включая `double` и `long`, нельзя снабдить модификатором `volatile`, потому что для них не гарантируется атомарность операций чтения и записи. Чтобы защитить многопотоковый доступ к полям таких типов, используйте члены класса Interlocked или защиту доступа с помощью инструкции `lock`.

Ключевое слово `volatile` можно применять только к полям `class` или `struct`. Локальные переменные не могут объявляться как `volatile`.

Пример

В следующем примере показано, как объявить переменную поля открытого типа `volatile`.

```
class VolatileTest
{
    public volatile int sharedStorage;

    public void Test(int i)
    {
        sharedStorage = i;
    }
}
```

Следующий пример демонстрирует создание вспомогательного или рабочего потока и его применение для выполнения обработки параллельно с обработкой основного потока. Дополнительные сведения о

многопоточности см. в разделе Управляемая поточность.

```
public class Worker
{
    // This method is called when the thread is started.
    public void DoWork()
    {
        bool work = false;
        while (!_shouldStop)
        {
            work = !work; // simulate some work
        }
        Console.WriteLine("Worker thread: terminating gracefully.");
    }
    public void RequestStop()
    {
        _shouldStop = true;
    }
    // Keyword volatile is used as a hint to the compiler that this data
    // member is accessed by multiple threads.
    private volatile bool _shouldStop;
}

public class WorkerThreadExample
{
    public static void Main()
    {
        // Create the worker thread object. This does not start the thread.
        Worker workerObject = new Worker();
        Thread workerThread = new Thread(workerObject.DoWork);

        // Start the worker thread.
        workerThread.Start();
        Console.WriteLine("Main thread: starting worker thread...");

        // Loop until the worker thread activates.
        while (!workerThread.IsAlive)
            ;

        // Put the main thread to sleep for 500 milliseconds to
        // allow the worker thread to do some work.
        Thread.Sleep(500);

        // Request that the worker thread stop itself.
        workerObject.RequestStop();

        // Use the Thread.Join method to block the current thread
        // until the object's thread terminates.
        workerThread.Join();
        Console.WriteLine("Main thread: worker thread has terminated.");
    }
    // Sample output:
    // Main thread: starting worker thread...
    // Worker thread: terminating gracefully.
    // Main thread: worker thread has terminated.
}
```

Добавив модификатор `volatile` к объявлению `_shouldStop`, вы всегда получите одинаковые результаты (как показано в приведенном выше фрагменте кода). Но если член `_shouldStop` не имеет этого модификатора, поведение будет непредсказуемым. Метод `DoWork` может оптимизировать доступ к членам, что приведет к чтению устаревших данных. В условиях многопоточного программирования невозможно прогнозировать число операций чтения устаревших данных. При каждом запуске программы результаты могут отличаться.

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Спецификация языка C#: ключевое слово volatile
- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Модификаторы
- Оператор lock
- Interlocked

Ключевые слова операторов (Справочник по C#)

18.12.2021 • 2 minutes to read

Операторы представляют собой инструкции для программы. За исключением случаев, которые описаны в разделах, приведенных в следующей таблице, операторы выполняются последовательно. В следующей таблице приводятся ключевые слова операторов в C#. Дополнительные сведения об операторах, которые не выражаются ключевыми словами, см. в разделе Операторы.

КАТЕГОРИЯ	КЛЮЧЕВЫЕ СЛОВА В C#
Операторы выбора	`if`, `switch`
Инструкции итерации	`do`, `for`, `foreach`, `while`
Операторы перехода	`break`, `continue`, `goto`, `return`
Операторы обработки исключений	throw, try-catch, try-finally, try-catch-finally
Инструкции checked и unchecked	checked, unchecked
Оператор fixed	fixed
Оператор lock	lock
Оператор yield	`yield`

См. также

- Справочник по C#
- Операторы
- Ключевые слова в C#

throw (Справочник по C#)

18.12.2021 • 2 minutes to read

Сообщает о возникновении исключения во время выполнения программы.

Remarks

Синтаксис `throw` выглядит следующим образом:

```
throw [e];
```

где `e` — это экземпляр класса, производного от System.Exception. В следующем примере используется оператор `throw` для создания IndexOutOfRangeException, если аргумент, переданный в метод с именем `GetNumber`, не соответствует допустимому индексу внутреннего массива.

```
using System;

namespace Throw2
{
public class NumberGenerator
{
   int[] numbers = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };

   public int GetNumber(int index)
   {
      if (index < 0 || index >= numbers.Length)
      {
         throw new IndexOutOfRangeException();
      }
      return numbers[index];
   }
}
```

Затем вызывающие объекты метода используют блок `try-catch` или `try-catch-finally` для обработки исключения. В следующем примере показана обработка исключения, созданного методом `GetNumber`.

```
using System;

public class Example
{
   public static void Main()
   {
      var gen = new NumberGenerator();
      int index = 10;
      try
      {
          int value = gen.GetNumber(index);
          Console.WriteLine($"Retrieved {value}");
      }
      catch (IndexOutOfRangeException e)
      {
         Console.WriteLine($"{e.GetType().Name}: {index} is outside the bounds of the array");
      }
   }
}
// The example displays the following output:
//        IndexOutOfRangeException: 10 is outside the bounds of the array
```

Повторный вызов исключения

`throw` можно также использовать в блоке `catch` для повторного создания исключения, обрабатываемого в блоке `catch`. В этом случае оператор `throw` не принимает операнд исключения. Это наиболее полезно, когда метод передает аргумент от вызывающего объекта в другой метод библиотеки, а метод библиотеки создает исключение, которое должно быть передано вызывающему объекту. Например, в следующем примере повторно создается исключение NullReferenceException, возникающее при попытке получить первый символ неинициализированной строки.

```
using System;

namespace Throw
{
public class Sentence
{
   public Sentence(string s)
   {
      Value = s;
   }

   public string Value { get; set; }

   public char GetFirstCharacter()
   {
      try
      {
         return Value[0];
      }
      catch (NullReferenceException e)
      {
         throw;
      }
   }
}

public class Example
{
   public static void Main()
   {
      var s = new Sentence(null);
      Console.WriteLine($"The first character is {s.GetFirstCharacter()}");
   }
}
// The example displays the following output:
//    Unhandled Exception: System.NullReferenceException: Object reference not set to an instance of an
object.
//       at Sentence.GetFirstCharacter()
//       at Example.Main()
```

IMPORTANT

Можно также использовать синтаксис `throw e` в блоке `catch`, чтобы создать исключение, которое будет передано вызывающему объекту. В этом случае трассировка стека исходного исключения, которое доступно из свойства StackTrace, не сохраняется.

Выражение `throw`

Начиная с C# 7.0 `throw` можно использовать в качестве выражения, а также как оператор. Это позволяет вызывать исключения в контекстах, которые ранее не поддерживались. Сюда входит следующее.

- Условный оператор. В следующем примере используется исключение `throw` для создания ArgumentException, если методу передается массив пустых строк. До выхода C# 7.0 эта логика должна была отображаться в операторе `if` / `else`.

```
private static void DisplayFirstNumber(string[] args)
{
   string arg = args.Length >= 1 ? args[0] :
                              throw new ArgumentException("You must supply an argument");
   if (Int64.TryParse(arg, out var number))
      Console.WriteLine($"You entered {number:F0}");
   else
      Console.WriteLine($"{arg} is not a number.");
}
```

- Оператор объединения с NULL. В следующем примере выражение `throw` используется с оператором, принимающим значение NULL, для создания исключения, если строка, назначенная свойству `Name` является `null`.

```
public string Name
{
    get => name;
    set => name = value ??
        throw new ArgumentNullException(paramName: nameof(value), message: "Name cannot be null");
}
```

- Метод или лямбда, воплощающие выражение. В следующем примере показан метод, воплощающий выражение, который создает InvalidCastException, так как преобразование в значение DateTime не поддерживается.

```
DateTime ToDateTime(IFormatProvider provider) =>
         throw new InvalidCastException("Conversion to a DateTime is not supported.");
```

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- try-catch
- Ключевые слова в C#
- Практическое руководство. Явное создание исключений

try-catch (Справочник по C#)

18.12.2021 • 8 minutes to read

Оператор try-catch состоит из блока `try`, за которым следует одно или несколько предложений `catch`, задающих обработчики для различных исключений.

При возникновении исключения общеязыковая среда выполнения (CLR) ищет оператор `catch`, который обрабатывает это исключение. Если текущий выполняемый метод не содержит такой блок `catch`, среда CLR выполняет поиск в методе, который вызвал текущий метод, и так далее вверх по стеку вызовов. Если блок `catch` не находится, то среда CLR отображает пользователю сообщение о необработанном исключении и останавливает выполнение программы.

Блок `try` содержит защищенный код, который может вызвать исключение. Этот блок выполняется, пока не возникнет исключение или пока он не будет успешно завершен. Например, следующая попытка приведения объекта `null` вызывает исключение NullReferenceException:

```
object o2 = null;
try
{
    int i2 = (int)o2;   // Error
}
```

Хотя предложение `catch` может использоваться без аргументов для перехвата любого типа исключения, такое использование не рекомендуется. В целом вы должны перехватывать только те исключения, после которых вы знаете, как выполнить восстановление. Поэтому всегда следует указывать аргумент объекта, производный от System.Exception. Тип исключения должен быть как можно более конкретным, чтобы избежать неправильного приема исключений, которые обработчик исключений не сможет разрешить. Таким образом, лучше использовать конкретные исключения вместо базового типа `Exception`. Вот несколько примеров.

```
catch (InvalidCastException e)
{
    // recover from exception
}
```

В одном блоке try-catch можно использовать несколько определенных предложений `catch`. В этом случае важен порядок предложений `catch`, поскольку предложения `catch` проверяются по порядку. Перехватывайте более конкретные исключения перед менее конкретными. Компилятор выдает ошибку, если вы расположили блоки catch в таком порядке, что последующий блок может быть никогда не достигнут.

Использование аргументов `catch` представляет один из способов фильтрации исключений, которые требуется обработать. Вы также можете использовать фильтр исключений, который дополнительно проверяет исключение, чтобы решить, следует ли его обрабатывать. Если фильтр исключений возвращает значение false, поиск обработчика продолжается.

```
catch (ArgumentException e) when (e.ParamName == "…")
{
    // recover from exception
}
```

Фильтры исключений предпочтительнее перехвата и повторного вызова (объясняется ниже), поскольку фильтры оставляют стек в целости и сохранности. Если последующий обработчик разгружает стек, вы можете увидеть, откуда изначально произошло исключение, а не только последнее место, в котором оно было повторно вызвано. Обычно выражения фильтра исключений используются для ведения журнала. Вы можете создать фильтр, который всегда возвращает значение false, а также записывает выходной результат в журнал, чтобы регистрировать исключения в журнале по мере их поступления без необходимости их обработки и повторного вызова.

Оператор throw, включенный в блок `catch`, позволяет заново вызвать исключение, перехваченное блоком `catch`. В следующем примере извлекаются сведения об источнике из исключения IOException, а затем это исключение вызывается для родительского метода.

```
catch (FileNotFoundException e)
{
    // FileNotFoundExceptions are handled here.
}
catch (IOException e)
{
    // Extract some information from this exception, and then
    // throw it to the parent method.
    if (e.Source != null)
        Console.WriteLine("IOException source: {0}", e.Source);
    throw;
}
```

Вы можете перехватывать одно исключение и вызывать другое исключение. При этом следует указать перехватываемое исключение как внутреннее, как показано в следующем примере.

```
catch (InvalidCastException e)
{
    // Perform some action here, and then throw a new exception.
    throw new YourCustomException("Put your error message here.", e);
}
```

Вы также можете повторно вызывать исключение при выполнении указанного условия, как показано в следующем примере.

```
catch (InvalidCastException e)
{
    if (e.Data == null)
    {
        throw;
    }
    else
    {
        // Take some action.
    }
}
```

NOTE

Вы можете использовать фильтр исключений, чтобы получить тот же результат, но проще (при этом не изменяя стек, как описано ранее в этом документе). В следующем примере показано такое же поведение для вызывающих объектов, как и в предыдущем примере. Эта функция отправляет исключение `InvalidCastException` обратно вызывающему объекту, если `e.Data` имеет значение `null`.

```
catch (InvalidCastException e) when (e.Data != null)
{
    // Take some action.
}
```

В блоке `try` инициализируйте только те переменные, которые в нем объявлены. В противном случае до завершения выполнения блока может возникнуть исключение. Например, в следующем примере кода переменная `n` инициализируется внутри блока `try`. Попытка использовать данную переменную вне этого блока `try` в инструкции `Write(n)` приведет к ошибке компилятора.

```
static void Main()
{
    int n;
    try
    {
        // Do not initialize this variable here.
        n = 123;
    }
    catch
    {
    }
    // Error: Use of unassigned local variable 'n'.
    Console.Write(n);
}
```

Дополнительные сведения о перехвате исключений см. в разделе try-catch-finally.

Исключения в асинхронных методах

Асинхронный метод помечается модификатором async и обычно содержит одно или несколько выражений или инструкций await. Выражение await применяет оператор await к Task или Task<TResult>.

Когда управление достигает `await` в асинхронном методе, выполнение метода приостанавливается до завершения выполнения ожидающей задачи. После завершения задачи выполнение в методе может быть возобновлено. Дополнительные сведения см. в разделе Асинхронное программирование с использованием ключевых слов async и await.

Завершенная задача, к которой применяется `await`, может находиться в состоянии сбоя из-за необработанного исключения в методе, который возвращает эту задачу. Ожидание задачи вызывает исключение. Задача также может завершиться в отмененном состоянии, если отменяется асинхронный процесс, возвращающий эту задачу. Ожидание отмененной задачи вызывает `OperationCanceledException`.

Для перехвата исключения ожидайте задачу в блоке `try` и перехватывайте это исключение в соответствующем блоке `catch`. См. пример в разделе Пример асинхронного метода.

Задача может быть в состоянии сбоя, если в ожидаемом асинхронном методе произошло несколько исключений. Например, задача может быть результатом вызова метода Task.WhenAll. При ожидании такой задачи перехватывается только одно из исключений и невозможно предсказать, какое исключение будет перехвачено. См. пример в разделе Пример Task.WhenAll.

Пример

В следующем примере блок `try` содержит вызов метода `ProcessString`, который может вызвать исключение. Предложение `catch` содержит обработчик исключений, который просто отображает сообщение на экране. Когда оператор `throw` вызывается из `ProcessString`, система осуществляет поиск оператора `catch` и отображает сообщение `Exception caught`.

```
class TryFinallyTest
{
    static void ProcessString(string s)
    {
        if (s == null)
        {
            throw new ArgumentNullException(paramName: nameof(s), message: "parameter can't be null.");
        }
    }

    public static void Main()
    {
        string s = null; // For demonstration purposes.

        try
        {
            ProcessString(s);
        }
        catch (Exception e)
        {
            Console.WriteLine("{0} Exception caught.", e);
        }
    }
}
/*
Output:
System.ArgumentNullException: Value cannot be null.
   at TryFinallyTest.Main() Exception caught.
 * */
```

Пример двух блоков catch

В следующем примере используются два блока catch и перехватывается наиболее конкретное исключение, поступившее первым.

Чтобы перехватить наименее конкретное исключение, можно заменить оператор throw в `ProcessString` следующим оператором: `throw new Exception()`.

Если в этом примере первым поместить блок catch для перехвата наименее конкретного исключения, то появится следующее сообщение об ошибке:
`A previous catch clause already catches all exceptions of this or a super type ('System.Exception')`.

```
class ThrowTest3
{
    static void ProcessString(string s)
    {
        if (s == null)
        {
            throw new ArgumentNullException(paramName: nameof(s), message: "Parameter can't be null");
        }
    }

    public static void Main()
    {
        try
        {
            string s = null;
            ProcessString(s);
        }
        // Most specific:
        catch (ArgumentNullException e)
        {
            Console.WriteLine("{0} First exception caught.", e);
        }
        // Least specific:
        catch (Exception e)
        {
            Console.WriteLine("{0} Second exception caught.", e);
        }
    }
}
/*
 Output:
 System.ArgumentNullException: Value cannot be null.
 at Test.ThrowTest3.ProcessString(String s) ... First exception caught.
*/
```

Пример асинхронного метода

В следующем примере демонстрируется обработка исключений для асинхронных методов. Для перехвата исключения, вызванного асинхронной задачей, поместите выражение `await` в блок `try` и перехватывайте это исключение в блоке `catch`.

Раскомментируйте строку `throw new Exception` в этом примере для демонстрации обработки исключений. Для свойства `IsFaulted` задачи установлено значение `True`, для свойства `Exception.InnerException` задачи установлено это исключение, а исключение перехватывается в блоке `catch`.

Раскомментируйте строку `throw new OperationCanceledException`, чтобы показать, что происходит при отмене асинхронного процесса. Для свойства `IsCanceled` задачи устанавливается значение `true`, и исключение перехватывается в блоке `catch`. В некоторых условиях, которые неприменимы в данном примере, для свойства `IsFaulted` задачи устанавливается значение `true`, а для `IsCanceled` устанавливается значение `false`.

```
public async Task DoSomethingAsync()
{
    Task<string> theTask = DelayAsync();

    try
    {
        string result = await theTask;
        Debug.WriteLine("Result: " + result);
    }
    catch (Exception ex)
    {
        Debug.WriteLine("Exception Message: " + ex.Message);
    }
    Debug.WriteLine("Task IsCanceled: " + theTask.IsCanceled);
    Debug.WriteLine("Task IsFaulted:  " + theTask.IsFaulted);
    if (theTask.Exception != null)
    {
        Debug.WriteLine("Task Exception Message: "
            + theTask.Exception.Message);
        Debug.WriteLine("Task Inner Exception Message: "
            + theTask.Exception.InnerException.Message);
    }
}

private async Task<string> DelayAsync()
{
    await Task.Delay(100);

    // Uncomment each of the following lines to
    // demonstrate exception handling.

    //throw new OperationCanceledException("canceled");
    //throw new Exception("Something happened.");
    return "Done";
}

// Output when no exception is thrown in the awaited method:
//   Result: Done
//   Task IsCanceled: False
//   Task IsFaulted:  False

// Output when an Exception is thrown in the awaited method:
//   Exception Message: Something happened.
//   Task IsCanceled: False
//   Task IsFaulted:  True
//   Task Exception Message: One or more errors occurred.
//   Task Inner Exception Message: Something happened.

// Output when a OperationCanceledException or TaskCanceledException
// is thrown in the awaited method:
//   Exception Message: canceled
//   Task IsCanceled: True
//   Task IsFaulted:  False
```

Пример Task.WhenAll

В следующем примере демонстрируется обработка исключений, когда несколько задач могут привести к нескольким исключениям. Блок `try` ожидает задачу, которая возвращается вызовом метода Task.WhenAll. Эта задача завершается после завершения трех задач, к которым применяется WhenAll.

Каждая из трех задач вызывает исключение. Блок `catch` выполняет итерацию по исключениям, которые обнаруживаются в свойстве `Exception.InnerExceptions` задачи, возвращенной методом Task.WhenAll.

```
public async Task DoMultipleAsync()
{
    Task theTask1 = ExcAsync(info: "First Task");
    Task theTask2 = ExcAsync(info: "Second Task");
    Task theTask3 = ExcAsync(info: "Third Task");

    Task allTasks = Task.WhenAll(theTask1, theTask2, theTask3);

    try
    {
        await allTasks;
    }
    catch (Exception ex)
    {
        Debug.WriteLine("Exception: " + ex.Message);
        Debug.WriteLine("Task IsFaulted: " + allTasks.IsFaulted);
        foreach (var inEx in allTasks.Exception.InnerExceptions)
        {
            Debug.WriteLine("Task Inner Exception: " + inEx.Message);
        }
    }
}

private async Task ExcAsync(string info)
{
    await Task.Delay(100);

    throw new Exception("Error-" + info);
}

// Output:
//   Exception: Error-First Task
//   Task IsFaulted: True
//   Task Inner Exception: Error-First Task
//   Task Inner Exception: Error-Second Task
//   Task Inner Exception: Error-Third Task
```

Спецификация языка C#

Дополнительные сведения см. в разделе Оператор try в документации Спецификация C# 6.0.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Операторы try, throw и catch (C++)
- throw
- try-finally
- Практическое руководство. Явное создание исключений

try-finally (Справочник по C#)

18.12.2021 • 3 minutes to read

С помощью блока `finally` можно выполнить очистку всех ресурсов, выделенных в блоке try, и запускать код даже при возникновении исключения в блоке `try`. Как правило, операторы блока `finally` выполняются, когда элемент управления покидает оператор `try`. Передача управления может возникать в результате нормального выполнения, выполнения операторов `break`, `continue`, `goto` или `return` или распространения исключения из оператора `try`.

Внутри обработанного исключения гарантируется выполнение связанного блока `finally`. Однако если исключение не обработано, то выполнение блока `finally` зависит от того, как запускается операция развертывания исключения. Это, в свою очередь, зависит от способа настройки компьютера. Единственные случаи, когда предложения `finally` не выполняются — немедленная остановка программы. Например, это может произойти при вызове исключения InvalidProgramException из-за повреждения инструкций IL. В большинстве операционных систем при остановке и выгрузке процесса будет выполняться обоснованная очистка ресурсов.

Как правило, когда необработанное исключение приводит к завершению работы приложения, выполнение блока `finally` не имеет значения. Однако если в блоке `finally` есть операторы, которые необходимо запускать даже в такой ситуации, одним из решений является добавление блока `catch` в оператор `try`-`finally`. Кроме того, можно перехватить исключение, которое может создаваться в блоке `try` оператора `try`-`finally` выше в стеке вызовов. То есть можно перехватить исключение в методе, который вызывает метод, содержащий оператор `try`-`finally`, или в методе, который вызывает этот метод, или в любом методе в стеке вызовов. Если исключение не перехвачено, выполнение блока `finally` зависит от того, активирует ли операционная система операцию развертывания исключения.

Пример

В следующем примере недопустимый оператор преобразования вызывает исключение `System.InvalidCastException`. Исключение не обрабатывается.

```
public class ThrowTestA
{
    public static void Main()
    {
        int i = 123;
        string s = "Some string";
        object obj = s;

        try
        {
            // Invalid conversion; obj contains a string, not a numeric type.
            i = (int)obj;

            // The following statement is not run.
            Console.WriteLine("WriteLine at the end of the try block.");
        }
        finally
        {
            // To run the program in Visual Studio, type CTRL+F5. Then
            // click Cancel in the error dialog.
            Console.WriteLine("\nExecution of the finally block after an unhandled\n" +
                "error depends on how the exception unwind operation is triggered.");
            Console.WriteLine("i = {0}", i);
        }
    }
    // Output:
    // Unhandled Exception: System.InvalidCastException: Specified cast is not valid.
    //
    // Execution of the finally block after an unhandled
    // error depends on how the exception unwind operation is triggered.
    // i = 123
}
```

В следующем примере исключение из метода `TryCast` перехватывается в методе выше в стеке вызовов.

```
public class ThrowTestB
{
    public static void Main()
    {
        try
        {
            // TryCast produces an unhandled exception.
            TryCast();
        }
        catch (Exception ex)
        {
            // Catch the exception that is unhandled in TryCast.
            Console.WriteLine
                ("Catching the {0} exception triggers the finally block.",
                ex.GetType());

            // Restore the original unhandled exception. You might not
            // know what exception to expect, or how to handle it, so pass
            // it on.
            throw;
        }
    }

    static void TryCast()
    {
        int i = 123;
        string s = "Some string";
        object obj = s;

        try
        {
            // Invalid conversion; obj contains a string, not a numeric type.
            i = (int)obj;

            // The following statement is not run.
            Console.WriteLine("WriteLine at the end of the try block.");
        }
        finally
        {
            // Report that the finally block is run, and show that the value of
            // i has not been changed.
            Console.WriteLine("\nIn the finally block in TryCast, i = {0}.\n", i);
        }
    }
    // Output:
    // In the finally block in TryCast, i = 123.

    // Catching the System.InvalidCastException exception triggers the finally block.

    // Unhandled Exception: System.InvalidCastException: Specified cast is not valid.
}
```

Дополнительные сведения о `finally` см. в разделе try-catch-finally.

C# также содержит оператор using, который предоставляет аналогичную функциональность для объектов IDisposable в удобном синтаксисе.

Спецификация языка C#

Дополнительные сведения см. в разделе Оператор try в документации Спецификация C# 6.0.

См. также

- Справочник по C#

- Руководство по программированию на C#
- Ключевые слова в C#
- Операторы try, throw и catch (C++)
- throw
- try-catch
- Практическое руководство. Явное создание исключений

try-catch-finally (Справочник по C#)

18.12.2021 • 2 minutes to read

Чаще всего `catch` и `finally` применяются вместе для получения и использования ресурсов в блоке `try`, устранения исключительных обстоятельств в блоке `catch` и освобождения ресурсов в блоке `finally`.

Дополнительные сведения и примеры повторного создания исключений см. в разделах try-catch и Порождение исключений. Дополнительные сведения о блоке `finally` см. в разделе try-finally.

Пример

```
public class EHClass
{
    void ReadFile(int index)
    {
        // To run this code, substitute a valid path from your local machine
        string path = @"c:\users\public\test.txt";
        System.IO.StreamReader file = new System.IO.StreamReader(path);
        char[] buffer = new char[10];
        try
        {
            file.ReadBlock(buffer, index, buffer.Length);
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine("Error reading from {0}. Message = {1}", path, e.Message);
        }
        finally
        {
            if (file != null)
            {
                file.Close();
            }
        }
        // Do something with buffer...
    }
}
```

Спецификация языка C#

Дополнительные сведения см. в разделе Оператор try в документации Спецификация C# 6.0.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Операторы try, throw и catch (C++)
- throw
- Практическое руководство. Явное создание исключений
- Оператор using

Checked и Unchecked (Справочник по C#)

18.12.2021 • 2 minutes to read

Строка кода C# может выполняться как в проверенном, так и в непроверенном контексте. В проверенном контексте арифметическое переполнение создает исключение. В непроверяемом контексте арифметическое переполнение игнорируется и результат усекается путем удаления старших разрядов, которые не помещаются в целевой тип данных.

- checked. Укажите проверенный контекст.
- unchecked. Укажите непроверенный контекст.

Следующие операции не затрагиваются проверкой переполнения.

- Выражения, использующие следующие предопределенные операторы на целочисленных типах:

  `++` , `--` ; унарные `-` , `+` , `-` , `*` , `/` .

- Явные числовые преобразования между целочисленными типами либо из `float` или `double` в целочисленный тип.

Если ни `checked` , ни `unchecked` не указаны, контекст по умолчанию для неконстантных выражений (выражения, вычисляемые во время выполнения) определяется по значению параметра компилятора **CheckForOverflowUnderflow**. По умолчанию значение этого параметра не задано, и арифметические операции выполняются в непроверенном контексте.

Для константных выражений (выражения, которые можно полностью вычислить во время компиляции) контекстом по умолчанию является проверяемый. Если только константное выражение явным образом не размещено в непроверенном контексте, переполнения, возникающие при вычислении выражения во время компиляции, вызывают ошибки времени компиляции.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Ключевые слова операторов

checked (Справочник по C#)

18.12.2021 • 2 minutes to read

Ключевое слово `checked` используется для явного включения проверки переполнения при выполнении арифметических операций и преобразований с данными целого типа.

По умолчанию выражение, содержащее только константные значения, вызывает ошибку компилятора в том случае, если результат его вычисления выходит за допустимые пределы значений конечного типа. Если выражение содержит одно или несколько неконстантных значений, компилятор не выполняет проверку переполнения. Вычисление выражения, присвоенного переменной `i2` в приведенном ниже примере, не вызывает ошибку компилятора.

```
// The following example causes compiler error CS0220 because 2147483647
// is the maximum value for integers.
//int i1 = 2147483647 + 10;

// The following example, which includes variable ten, does not cause
// a compiler error.
int ten = 10;
int i2 = 2147483647 + ten;

// By default, the overflow in the previous statement also does
// not cause a run-time exception. The following line displays
// -2,147,483,639 as the sum of 2,147,483,647 and 10.
Console.WriteLine(i2);
```

По умолчанию эти неконстантные выражения также не проверяются на переполнение во время выполнения, и они не создают исключений переполнения. В предыдущем примере в качестве суммы двух положительных целых чисел выводится значение -2 147 483 639.

Проверку переполнения можно включить посредством параметров компилятора, настройки среды или использования ключевого слова `checked`. В следующих примерах демонстрируется использование выражения `checked` или блока `checked` для обнаружения переполнения, возникающего в результате предыдущего сложения во время выполнения. В обоих примерах создается исключение переполнения.

```
// If the previous sum is attempted in a checked environment, an
// OverflowException error is raised.

// Checked expression.
Console.WriteLine(checked(2147483647 + ten));

// Checked block.
checked
{
    int i3 = 2147483647 + ten;
    Console.WriteLine(i3);
}
```

Для запрета проверки переполнения можно использовать ключевое слово unchecked.

Пример

В этом примере демонстрируется включение проверки переполнения во время выполнения посредством ключевого слова `checked`.

```
class OverFlowTest
{
    // Set maxIntValue to the maximum value for integers.
    static int maxIntValue = 2147483647;

    // Using a checked expression.
    static int CheckedMethod()
    {
        int z = 0;
        try
        {
            // The following line raises an exception because it is checked.
            z = checked(maxIntValue + 10);
        }
        catch (System.OverflowException e)
        {
            // The following line displays information about the error.
            Console.WriteLine("CHECKED and CAUGHT:  " + e.ToString());
        }
        // The value of z is still 0.
        return z;
    }

    // Using an unchecked expression.
    static int UncheckedMethod()
    {
        int z = 0;
        try
        {
            // The following calculation is unchecked and will not
            // raise an exception.
            z = maxIntValue + 10;
        }
        catch (System.OverflowException e)
        {
            // The following line will not be executed.
            Console.WriteLine("UNCHECKED and CAUGHT:  " + e.ToString());
        }
        // Because of the undetected overflow, the sum of 2147483647 + 10 is
        // returned as -2147483639.
        return z;
    }

    static void Main()
    {
        Console.WriteLine("\nCHECKED output value is: {0}",
                          CheckedMethod());
        Console.WriteLine("UNCHECKED output value is: {0}",
                          UncheckedMethod());
    }
    /*
   Output:
   CHECKED and CAUGHT:  System.OverflowException: Arithmetic operation resulted
   in an overflow.
      at ConsoleApplication1.OverFlowTest.CheckedMethod()

   CHECKED output value is: 0
   UNCHECKED output value is: -2147483639
 */
}
```

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Checked и Unchecked
- unchecked

unchecked (справочник по C#)

18.12.2021 • 2 minutes to read

Ключевое слово `unchecked` позволяет предотвратить проверку переполнения при выполнении арифметических операций и преобразований с данными целого типа.

В непроверенном контексте перегрузка не помечается, если результат выражения выходит за допустимые пределы значений конечного типа. Например, в связи с тем, что вычисление в приведенном выше примере выполняется в блоке или выражении `unchecked`, тот факт, что результат слишком велик для целого числа, игнорируется, а `int1` присваивается значение -2,147,483,639.

```
unchecked
{
    int1 = 2147483647 + 10;
}
int1 = unchecked(ConstantMax + 10);
```

При удалении среды `unchecked` возникает ошибка компиляции. Во время компиляции может быть обнаружено переполнение, поскольку все члены данного выражения являются константами.

Выражения, содержащие члены, которые не являются константами, по умолчанию не проверяются ни во время компиляции, ни во время выполнения. Дополнительные сведения о включении проверяемой среды см. в разделе checked.

Поскольку проверка на переполнение занимает определенное время, в ситуациях, где нет опасности переполнения, можно повысить производительность, выбрав непроверяемый код. Если же переполнение вероятно, следует использовать проверяемую среду.

Пример

В этом примере демонстрируется применение ключевого слова `unchecked`.

```
class UncheckedDemo
{
    static void Main(string[] args)
    {
        // int.MaxValue is 2,147,483,647.
        const int ConstantMax = int.MaxValue;
        int int1;
        int int2;
        int variableMax = 2147483647;

        // The following statements are checked by default at compile time. They do not
        // compile.
        //int1 = 2147483647 + 10;
        //int1 = ConstantMax + 10;

        // To enable the assignments to int1 to compile and run, place them inside
        // an unchecked block or expression. The following statements compile and
        // run.
        unchecked
        {
            int1 = 2147483647 + 10;
        }
        int1 = unchecked(ConstantMax + 10);

        // The sum of 2,147,483,647 and 10 is displayed as -2,147,483,639.
        Console.WriteLine(int1);

        // The following statement is unchecked by default at compile time and run
        // time because the expression contains the variable variableMax. It causes
        // overflow but the overflow is not detected. The statement compiles and runs.
        int2 = variableMax + 10;

        // Again, the sum of 2,147,483,647 and 10 is displayed as -2,147,483,639.
        Console.WriteLine(int2);

        // To catch the overflow in the assignment to int2 at run time, put the
        // declaration in a checked block or expression. The following
        // statements compile but raise an overflow exception at run time.
        checked
        {
            //int2 = variableMax + 10;
        }
        //int2 = checked(variableMax + 10);

        // Unchecked sections frequently are used to break out of a checked
        // environment in order to improve performance in a portion of code
        // that is not expected to raise overflow exceptions.
        checked
        {
            // Code that might cause overflow should be executed in a checked
            // environment.
            unchecked
            {
                // This section is appropriate for code that you are confident
                // will not result in overflow, and for which performance is
                // a priority.
            }
            // Additional checked code here.
        }
    }
}
```

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является

предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Checked и Unchecked
- checked

Оператор fixed (Справочник по C#)

18.12.2021 • 3 minutes to read

Оператор `fixed` не позволяет сборщику мусора переносить перемещаемую переменную. Оператор `fixed` допускается только в небезопасном контексте. Можно также использовать ключевое слово `fixed` для создания буферов фиксированного размера.

Оператор `fixed` задает указатель на управляемую переменную и "закрепляет" эту переменную во время выполнения оператора. Указатели на перемещаемые управляемые переменные полезны только в контексте `fixed`. Без контекста `fixed` при сборке мусора эти переменные могут переноситься непредсказуемым образом. Компилятор C# позволяет присвоить указатель только управляемой переменной в операторе `fixed`.

```
class Point
{
    public int x;
    public int y;
}

unsafe private static void ModifyFixedStorage()
{
    // Variable pt is a managed variable, subject to garbage collection.
    Point pt = new Point();

    // Using fixed allows the address of pt members to be taken,
    // and "pins" pt so that it is not relocated.

    fixed (int* p = &pt.x)
    {
        *p = 1;
    }
}
```

Вы можете инициализировать указатель, используя массив, строку, буфер фиксированного размера или адрес переменной. Следующий пример иллюстрирует использование переменных адресов, массивов и строк:

```
Point point = new Point();
double[] arr = { 0, 1.5, 2.3, 3.4, 4.0, 5.9 };
string str = "Hello World";

// The following two assignments are equivalent. Each assigns the address
// of the first element in array arr to pointer p.

// You can initialize a pointer by using an array.
fixed (double* p = arr) { /*...*/ }

// You can initialize a pointer by using the address of a variable.
fixed (double* p = &arr[0]) { /*...*/ }

// The following assignment initializes p by using a string.
fixed (char* p = str) { /*...*/ }

// The following assignment is not valid, because str[0] is a char,
// which is a value, not a variable.
//fixed (char* p = &str[0]) { /*...*/ }
```

Начиная с C# 7.3, оператор `fixed` работает с дополнительными типами, помимо массивов, строк, буферов фиксированного размера и неуправляемых переменных. Любой тип, реализующий метод `GetPinnableReference`, можно зафиксировать. `GetPinnableReference` должен вернуть переменную `ref` неуправляемого типа. Типы .NET System.Span<T> и System.ReadOnlySpan<T>, представленные в .NET Core 2.0, используют этот шаблон и могут быть зафиксированы. Это показано в следующем примере:

```
unsafe private static void FixedSpanExample()
{
    int[] PascalsTriangle = {
                   1,
                1,  1,
             1,  2,  1,
          1,  3,  3,  1,
       1,  4,  6,  4,  1,
    1,  5,  10, 10, 5,  1
    };

    Span<int> RowFive = new Span<int>(PascalsTriangle, 10, 5);

    fixed (int* ptrToRow = RowFive)
    {
        // Sum the numbers 1,4,6,4,1
        var sum = 0;
        for (int i = 0; i < RowFive.Length; i++)
        {
            sum += *(ptrToRow + i);
        }
        Console.WriteLine(sum);
    }
}
```

При создании типов, которые должны участвовать в этом шаблоне, перейдите по ссылке Span<T>.GetPinnableReference() для примера реализации шаблона.

В одном операторе можно инициализировать несколько указателей одного типа.

```
fixed (byte* ps = srcarray, pd = dstarray) {...}
```

Чтобы инициализировать указатели разных типов, просто вложите операторы `fixed`, как показано в приведенном ниже примере.

```
fixed (int* p1 = &point.x)
{
    fixed (double* p2 = &arr[5])
    {
        // Do something with p1 and p2.
    }
}
```

После выполнения кода в операторе закрепление всех переменных отменяется, и они могут пройти сборку мусора. Таким образом, не следует использовать указатели на эти переменные за пределами оператора `fixed`. Переменные, объявленные в операторе `fixed`, относятся к этому оператору, что упрощает выполнение следующего выражения.

```
fixed (byte* ps = srcarray, pd = dstarray)
{
   ...
}
// ps and pd are no longer in scope here.
```

Указатели, инициализированные в операторах `fixed`, являются переменными только для чтения. Чтобы изменить значение указателя, необходимо объявить переменную второго указателя и изменить ее. Переменную, объявленную в операторе `fixed`, изменить невозможно.

```
fixed (byte* ps = srcarray, pd = dstarray)
{
    byte* pSourceCopy = ps;
    pSourceCopy++; // point to the next element.
    ps++; // invalid: cannot modify ps, as it is declared in the fixed statement.
}
```

Вы можете выделить память в стеке, где на нее не будет распространяться сборка мусора, поэтому ее не нужно будет закреплять. Для этого используйте выражение `stackalloc`.

Спецификация языка C#

Подробные сведения см. в разделе The fixed statement (Оператор fixed) спецификации языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- unsafe
- Типы указателей
- Буферы фиксированного размера

Инструкция lock. (Справочник по C#)

18.12.2021 • 2 minutes to read

Оператор `lock` получает взаимоисключающую блокировку заданного объекта перед выполнением определенных операторов, а затем снимает блокировку. Во время блокировки поток, удерживающий блокировку, может снова поставить и снять блокировку. Любой другой поток не может получить блокировку и ожидает ее снятия.

Оператор `lock` имеет форму

```
lock (x)
{
    // Your code...
}
```

Здесь `x` — это выражение ссылочного типа. Оно является точным эквивалентом

```
object __lockObj = x;
bool __lockWasTaken = false;
try
{
    System.Threading.Monitor.Enter(__lockObj, ref __lockWasTaken);
    // Your code...
}
finally
{
    if (__lockWasTaken) System.Threading.Monitor.Exit(__lockObj);
}
```

Так как в коде используется блок try... finally, блокировка освобождается, даже если возникает исключение в теле оператора `lock`.

Нельзя использовать оператор await в теле оператора `lock`.

Рекомендации

При синхронизации доступа потоков к общему ресурсу блокируйте выделенный экземпляр объекта (например, `private readonly object balanceLock = new object();`) или другой экземпляр, который, скорее всего, не будет использоваться как объект блокировки другими частями кода. Не используйте один и тот же экземпляр объекта блокировки для разных общих ресурсов: это может привести к взаимоблокировке или состязанию при блокировке. В особенности избегайте использования следующих объектов в качестве объектов блокировки:

- `this`, так как он может использоваться вызывающими объектами как блокировка;
- экземпляров Type, так как их может получать оператор typeof или отражение;
- строковых экземпляров, включая строковые литералы, так как они могут быть интернированы.

Удерживайте блокировку в течение максимально короткого времени, чтобы сократить число конфликтов при блокировке.

Пример

В следующем примере определяется класс `Account`, который синхронизирует доступ к закрытому полю `balance` путем блокировки выделенного экземпляра `balanceLock`. Использование того же экземпляра для блокировки гарантирует, что поле `balance` не смогут одновременно обновить два потока, пытающиеся вызвать методы `Debit` или `Credit` одновременно.

```
using System;
using System.Threading.Tasks;

public class Account
{
    private readonly object balanceLock = new object();
    private decimal balance;

    public Account(decimal initialBalance) => balance = initialBalance;

    public decimal Debit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "The debit amount cannot be negative.");
        }

        decimal appliedAmount = 0;
        lock (balanceLock)
        {
            if (balance >= amount)
            {
                balance -= amount;
                appliedAmount = amount;
            }
        }
        return appliedAmount;
    }

    public void Credit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "The credit amount cannot be negative.");
        }

        lock (balanceLock)
        {
            balance += amount;
        }
    }

    public decimal GetBalance()
    {
        lock (balanceLock)
        {
            return balance;
        }
    }
}

class AccountTest
{
    static async Task Main()
    {
        var account = new Account(1000);
        var tasks = new Task[100];
        for (int i = 0; i < tasks.Length; i++)
        {
            tasks[i] = Task.Run(() => Update(account));
        }
        await Task.WhenAll(tasks);
```

```
        await Task.WhenAll(tasks);
        Console.WriteLine($"Account's balance is {account.GetBalance()}");
        // Output:
        // Account's balance is 2000
    }

    static void Update(Account account)
    {
        decimal[] amounts = { 0, 2, -3, 6, -2, -1, 8, -5, 11, -6 };
        foreach (var amount in amounts)
        {
            if (amount >= 0)
            {
                account.Credit(amount);
            }
            else
            {
                account.Debit(Math.Abs(amount));
            }
        }
    }
}
```

Спецификация языка C#

Дополнительные сведения см. в разделе об инструкции lock в документации Предварительная спецификация C# 6.0.

См. также

- справочник по C#
- Ключевые слова C#
- System.Threading.Monitor
- System.Threading.SpinLock
- System.Threading.Interlocked
- Обзор примитивов синхронизации

Параметры методов (Справочник по C#)

18.12.2021 • 2 minutes to read

Параметры, объявленные для метода без in, ref или out, передаются в вызываемый метод по значению. Это значение может изменяться в методе, однако измененное значение не будет сохраняться при возврате управления вызывающей процедуре. С помощью ключевого слова параметра метода это поведение можно изменить.

В этом разделе описываются ключевые слова, которые можно использовать при объявлении параметров метода:

- params указывает, что этот параметр может принимать переменное количество аргументов.
- in указывает, что этот параметр передается по ссылке, но лишь считывается вызванным методом.
- ref указывает, что этот параметр передается по ссылке и может быть считан или записан вызванным методом.
- out указывает, что этот параметр передается по ссылке и записывается вызванным методом.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#

params (справочник по C#)

18.12.2021 • 2 minutes to read

С помощью ключевого слова `params` можно указать параметр метода, принимающий переменное число аргументов. Тип параметра должен быть одномерным массивом.

В объявлении метода после ключевого слова `params` дополнительные параметры не допускаются, и в объявлении метода допускается только одно ключевое слово `params`.

Если объявленный тип параметра `params` не является одномерным массивом, возникает ошибка компилятора CS0225.

При вызове метода с параметром `params` можно передать следующие объекты:

- разделенный запятыми список аргументов типа элементов массива;
- массив аргументов указанного типа;
- не передавать аргументы. Если аргументы не отправляются, длина списка `params` равна нулю.

Пример

В следующем примере показаны различные способы оправки аргументов параметру `params`.

```
public class MyClass
{
    public static void UseParams(params int[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    public static void UseParams2(params object[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    static void Main()
    {
        // You can send a comma-separated list of arguments of the
        // specified type.
        UseParams(1, 2, 3, 4);
        UseParams2(1, 'a', "test");

        // A params parameter accepts zero or more arguments.
        // The following calling statement displays only a blank line.
        UseParams2();

        // An array argument can be passed, as long as the array
        // type matches the parameter type of the method being called.
        int[] myIntArray = { 5, 6, 7, 8, 9 };
        UseParams(myIntArray);

        object[] myObjArray = { 2, 'b', "test", "again" };
        UseParams2(myObjArray);

        // The following call causes a compiler error because the object
        // array cannot be converted into an integer array.
        //UseParams(myObjArray);

        // The following call does not cause an error, but the entire
        // integer array becomes the first element of the params array.
        UseParams2(myIntArray);
    }
}
/*
Output:
    1 2 3 4
    1 a test

    5 6 7 8 9
    2 b test again
    System.Int32[]
*/
```

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Параметры методов

Модификатор параметров in (справочник по C#)

18.12.2021 • 5 minutes to read

Ключевое слово `in` вызывает передачу аргументов по ссылке, но гарантирует, что аргумент не будет изменен. В результате этот формальный параметр становится псевдонимом для аргумента, который должен представлять собой переменную. Другими словами, любая операция в параметре осуществляется с аргументом. Оно аналогично ключевым словам ref и out, за исключением того, что аргументы `in` не могут быть изменены вызываемым методом. В то время как аргументы `ref` могут быть изменены, аргументы `out` должны быть изменены вызывающим объектом, и эти изменения отслеживаются в вызывающем контексте.

```
int readonlyArgument = 44;
InArgExample(readonlyArgument);
Console.WriteLine(readonlyArgument);     // value is still 44

void InArgExample(in int number)
{
    // Uncomment the following line to see error CS8331
    //number = 19;
}
```

Предыдущий пример показывает, что модификатор `in` обычно не требуется в месте вызова. Он нужен только в объявлении метода.

> **NOTE**
> Ключевое слово `in` также можно использовать с параметром универсального типа для указания на то, что тип параметра является контравариантным, в рамках оператора `foreach` или предложения `join` в запросе LINQ. Дополнительные сведения об использовании ключевого слова `in` в таких контекстах см. в разделе in, где приведены все соответствующие ссылки.

Переменные, передаваемые в качестве аргументов `in`, требуется инициализировать перед передачей в вызов метода. Но вызванный метод не может присвоить значение или изменить аргумент.

Модификатор параметра `in` доступен в C# 7.2 и более поздних версиях. Предыдущие версии создают ошибку компилятора `CS8107` ("Функция "Ссылки только для чтения" недоступна в C# 7.0. Используйте языковую версию 7.2 или выше"), чтобы настроить языковую версию компилятора, см. Выбор версии языка C#.

Несмотря на то, что модификаторы параметров `in`, `out` и `ref` считаются частью сигнатуры, элементы, объявленные в одном типе, не могут различаться в сигнатуре только по параметрам `in`, `ref` и `out`. Таким образом, методы не могут быть перегружены, если единственное различие состоит в том, что один метод принимает аргумент `ref` или `in`, а другой — `out`. Следующий код, например, компилироваться не будет.

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on in, ref and out".
    public void SampleMethod(in int i) { }
    public void SampleMethod(ref int i) { }
}
```

Перегрузка, основанная на наличии `in`, допустима:

```
class InOverloads
{
    public void SampleMethod(in int i) { }
    public void SampleMethod(int i) { }
}
```

Правила разрешения перегрузки

Разобраться в правилах разрешения перегрузки для методов по значению и аргументам `in` можно, поняв основания для применения аргументов `in`. Определение методов с помощью параметров `in` является потенциальной оптимизацией производительности. Некоторые аргументы типа `struct` могут иметь большой размер, и при вызове методов в ограниченных циклах или критических путях кода стоимость копирования этих структур имеет определяющее значение. Методы объявляют параметры `in`, чтобы указать, что аргументы можно безопасно передавать по ссылке, так как вызываемый метод не изменяет состояние этого аргумента. Передача этих аргументов по ссылке позволяет избежать (потенциально) дорогого копирования.

Указывать `in` для аргументов в месте вызова обычно необязательно. Нет семантического различия между передачей аргументов по значению и передачей их по ссылке с помощью модификатора `in`. Модификатор `in` в месте вызова является необязательным, так как не нужно указывать, что значение аргумента может изменяться. Вы явным образом добавляете модификатор `in` в место вызова, чтобы аргумент передавался по ссылке, а не по значению. Явное использование `in` приводит к двум результатам.

Во-первых, указание `in` в месте вызова вынуждает компилятор выбрать метод, определенный с помощью соответствующего параметра `in`. В противном случае, когда два метода отличаются только наличием `in`, перегрузка по значению подходит лучше.

Во-вторых, указав `in`, вы объявляете о намерении передать аргумент по ссылке. Аргумент, используемый с `in`, должен представлять расположение, на которое можно сослаться напрямую. Такие же общие правила применяются к аргументам `out` и `ref`: вы не можете использовать константы, обычные свойства или другие выражения, возвращающие значения. В противном случае пропуск `in` в месте вызова сообщает компилятору, что вы разрешите ему создать временную переменную для передачи с помощью ссылки на метод, доступной только для чтения. Компилятор создает временную переменную, чтобы преодолеть некоторые ограничения для аргументов `in`:

- Временная переменная позволяет использовать константы времени компиляции, например параметры `in`.
- Временная переменная позволяет использовать свойства или другие выражения для параметров `in`.
- Временная переменная позволяет использовать аргументы, где выполняется неявное преобразование из типа аргумента в тип параметра.

Во всех предыдущих случаях компилятор создает временную переменную, хранящую значение константы, свойства или другого выражения.

Эти правила проиллюстрированы в следующем коде:

```
static void Method(in int argument)
{
    // implementation removed
}

Method(5); // OK, temporary variable created.
Method(5L); // CS1503: no implicit conversion from long to int
short s = 0;
Method(s); // OK, temporary int created with the value 0
Method(in s); // CS1503: cannot convert from in short to in int
int i = 42;
Method(i); // passed by readonly reference
Method(in i); // passed by readonly reference, explicitly using `in`
```

Теперь предположим, что был доступен другой метод, использующий аргументы по значению. Результаты изменяются, как показано в следующем коде:

```
static void Method(int argument)
{
    // implementation removed
}

static void Method(in int argument)
{
    // implementation removed
}

Method(5); // Calls overload passed by value
Method(5L); // CS1503: no implicit conversion from long to int
short s = 0;
Method(s); // Calls overload passed by value.
Method(in s); // CS1503: cannot convert from in short to in int
int i = 42;
Method(i); // Calls overload passed by value
Method(in i); // passed by readonly reference, explicitly using `in`
```

Аргумент передается по ссылке только в последнем вызове метода.

> **NOTE**
> Для простоты предыдущий код использует `int` в качестве типа аргумента. Так как по размеру `int` не больше ссылки на большинстве современных компьютеров, не имеет смысла передавать один `int` в качестве ссылки, доступной только для чтения.

Ограничения для параметров `in`

Ключевые слова `in`, `ref` и `out` запрещено использовать для следующих типов методов.

- Асинхронные методы, которые определяются с помощью модификатора async.
- Методы итератора, которые включают оператор yield return или `yield break`.
- Первый аргумент метода расширения не может иметь модификатор `in`, если только этот аргумент не является структурой.
- Первый аргумент метода расширения, в котором этот аргумент является универсальным типом (даже если этот тип ограничен структурой).

Дополнительные сведения о модификаторе `in`, отличии от `ref` и `out` см в статье Написание

безопасного эффективного кода.

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

ref (Справочник по C#)

18.12.2021 • 8 minutes to read

Ключевое слово `ref` указывает, что значение передается по ссылке. Оно используется в четырех разных контекстах:

- В сигнатуре и вызове метода для передачи аргумента в метод по ссылке. Дополнительные сведения см. в статье о передаче аргументов по ссылке.
- В сигнатуре метода для возврата значения вызывающему объекту по ссылке. Дополнительные сведения см. в разделе Значения, возвращаемые по ссылке.
- В теле элемента для указания на то, что возвращаемые ссылочные значения хранятся локально в виде ссылки, которая может быть изменена вызывающим объектом. Или чтобы указать, что локальная переменная обращается к другому значению по ссылке. Дополнительные сведения см. в статье о ссылочных локальных переменных.
- В объявлении `struct`, чтобы объявить `ref struct` или `readonly ref struct`. Дополнительные сведения см. в описании структуры `ref` в статье Типы структур.

Передача аргументов по ссылке

При использовании в списке параметров метода ключевое слово `ref` указывает на то, что аргумент передается по ссылке, а не по значению. Ключевое слово `ref` позволяет превратить этот формальный параметр в псевдоним для аргумента, который должен представлять собой переменную. Другими словами, любая операция в параметре осуществляется с аргументом.

Для примера предположим, что вызывающая сторона передает выражение локальной переменной или выражение доступа к элементу массива. Тогда вызываемый метод может заменить объект, на который ссылается параметр ref. В этом случае или элемент массива или локальная переменная вызывающей стороны будет ссылаться на новый объект, когда метод возвращает значение.

> **NOTE**
> Не путайте понятие передачи по ссылке с понятием ссылочных типов. Эти два понятия не совпадают. Параметр метода может быть изменен с помощью `ref` независимо от того, принадлежит ли он к типу значения или ссылочному типу. При передаче по ссылке упаковка-преобразование типа значения не производится.

Для использования параметра `ref` и при определении метода, и при вызове метода следует явно использовать ключевое слово `ref`, как показано в следующем примере. (За исключением того, что вызывающий метод может опускать `ref` при вызове COM.)

```
void Method(ref int refArgument)
{
    refArgument = refArgument + 44;
}

int number = 1;
Method(ref number);
Console.WriteLine(number);
// Output: 45
```

Аргумент, передаваемый в параметр `ref` или `in`, нужно инициализировать перед передачей. В этом и заключается отличие от параметров out, аргументы которых не нужно явно инициализировать перед передачей.

Члены класса не могут иметь сигнатуры, отличающихся только `ref`, `in` или `out`. Если единственное различие между двумя членами типа состоит в том, что один из них имеет параметр `ref`, а второй — параметр `out` или `in`, возникает ошибка компилятора. Например, следующий код не будет компилироваться.

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on ref and out".
    public void SampleMethod(out int i) { }
    public void SampleMethod(ref int i) { }
}
```

Но методы можно перегружать, если один метод имеет параметр `ref`, `in` или `out`, а другой — передаваемый по значению параметр, как показано в приведенном ниже примере.

```
class RefOverloadExample
{
    public void SampleMethod(int i) { }
    public void SampleMethod(ref int i) { }
}
```

В других ситуациях, требующих соответствия сигнатур, таких как скрытие или переопределение, `in`, `ref` и `out` являются частью сигнатуры и не соответствуют друг другу.

Свойства не являются переменными. Они являются методами и не могут передаваться в параметры `ref`.

Ключевые слова `ref`, `in` и `out` запрещено использовать для следующих типов методов.

- Асинхронные методы, которые определяются с помощью модификатора async.
- Методы итератора, которые включают оператор yield return или `yield break`.

Методы расширения также имеют ряд ограничений при использовании этих ключевых слов:

- Ключевое слово `out` нельзя использовать в первом аргументе метода расширения.
- Ключевое слово `ref` нельзя использовать в первом аргументе метода расширения, если аргумент не является структурой, или если универсальный тип не ограничен структурой.
- Ключевое слово `in` нельзя использовать, если только первый аргумент не является структурой. Ключевое слово `in` нельзя использовать ни для одного универсального типа, даже если оно ограничено структурой.

Передача аргументов по ссылке. Пример

В предыдущих примерах типы значений передаются по ссылке. Также можно использовать ключевое слово `ref` для передачи ссылочных типов по ссылке. Передача ссылочного типа по ссылке позволяет вызываемому методу заменять объект, на который указывает ссылочный параметр в вызывающем объекте. Место хранения объекта передается методу в качестве значения ссылочного параметра. Если изменить место хранения параметра (с указанием на новый объект), необходимо изменить место хранения, на который ссылается вызывающий объект. В следующем примере экземпляр ссылочного типа передается как параметр `ref`.

```
class Product
{
    public Product(string name, int newID)
    {
        ItemName = name;
        ItemID = newID;
    }

    public string ItemName { get; set; }
    public int ItemID { get; set; }
}

private static void ChangeByReference(ref Product itemRef)
{
    // Change the address that is stored in the itemRef parameter.
    itemRef = new Product("Stapler", 99999);

    // You can change the value of one of the properties of
    // itemRef. The change happens to item in Main as well.
    itemRef.ItemID = 12345;
}

private static void ModifyProductsByReference()
{
    // Declare an instance of Product and display its initial values.
    Product item = new Product("Fasteners", 54321);
    System.Console.WriteLine("Original values in Main.  Name: {0}, ID: {1}\n",
        item.ItemName, item.ItemID);

    // Pass the product instance to ChangeByReference.
    ChangeByReference(ref item);
    System.Console.WriteLine("Back in Main.  Name: {0}, ID: {1}\n",
        item.ItemName, item.ItemID);
}

// This method displays the following output:
// Original values in Main.  Name: Fasteners, ID: 54321
// Back in Main.  Name: Stapler, ID: 12345
```

Дополнительные сведения о передаче ссылочных типов по значению и по ссылке см. в разделе Передача параметров ссылочного типа.

Возвращаемые ссылочные значения

Возвращаемые ссылочные значения — это значения, которые метод возвращает вызывающему объекту по ссылке. Это значит, что вызывающий объект может изменять значение, возвращаемое методом, и это изменение будет отражаться в состоянии объекта в вызывающем методе.

Возвращаемое ссылочное значение определяется с помощью ключевого слова `ref`:

- В сигнатуре метода. Например, следующая сигнатура указывает, что метод `GetCurrentPrice` возвращает значение Decimal по ссылке.

```
public ref decimal GetCurrentPrice()
```

- Между токеном `return` и переменной, возвращенной в инструкции `return` в методе. Пример:

```
return ref DecimalArray[0];
```

Чтобы вызывающий объект имел возможность изменять состояние объекта, возвращаемое ссылочное значение должно храниться в переменной, которая явно определена как ссылочная локальная переменная.

Ниже приведен более полный пример возвращаемого ссылочного значения, в котором показаны сигнатура и тело метода.

```
public static ref int Find(int[,] matrix, Func<int, bool> predicate)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
        for (int j = 0; j < matrix.GetLength(1); j++)
            if (predicate(matrix[i, j]))
                return ref matrix[i, j];
    throw new InvalidOperationException("Not found");
}
```

Вызываемый метод может также объявить возвращаемое значение `ref readonly`, чтобы вернуть значение по ссылке, и запретить вызывающему коду изменять возвращаемое значение. Вызывающий метод может обойтись без копирования возвращаемого значения, сохраняя его в локальной переменной ref readonly.

Пример см. в разделе Пример использования возвращаемых ссылочных значений и ссылочных локальных переменных.

Ссылочные локальные переменные

Ссылочная локальная переменная задает ссылку на значения, возвращаемые с помощью `return ref`. Ссылочная локальная переменная не может инициализироваться как не ссылочное возвращаемое значение. Другими словами, правая часть инициализации должна быть ссылкой. Любые изменения в значении ссылочной локальной переменной отражаются в состоянии объекта, метод которого возвращает значение по ссылке.

Чтобы определить локальное ссылочное значение, ключевое слово `ref` следует указать в двух местах:

- перед объявлением переменной;
- сразу после вызова метода, который возвращает значение по ссылке.

Например, следующая инструкция определяет значение ссылочной локальной переменной, которое возвращается методом `GetEstimatedValue`:

```
ref decimal estValue = ref Building.GetEstimatedValue();
```

Вы можете таким же образом обратиться к значению по ссылке. В некоторых случаях обращение к значению по ссылке повышает производительность, поскольку позволяет избежать потенциально затратной операции копирования. Например, следующая инструкция позволяет определить локальную ссылочную переменную, которая используется для создания ссылки на значение.

```
ref VeryLargeStruct reflocal = ref veryLargeStruct;
```

В обоих примерах ключевое слово `ref` необходимо использовать в обоих местах. В противном случае возникает ошибка компилятора CS8172, "Невозможно инициализировать значением переменную по ссылке".

Начиная с версии C# 7.3, переменная итерации в операторе `foreach` может являться локальной ссылочной переменной или локальной ссылочной переменной только для чтения. Дополнительные

сведения см. в статье, посвященной инструкции foreach.

Также, начиная с версии C# 7.3, вы можете переназначать ссылочную локальную переменную или ссылочную локальную переменную только для чтения с помощью ссылочного оператора присваивания.

Ссылочные локальные переменные только для чтения

Ссылочная локальная переменная только для чтения позволяет создать ссылку на значения, возвращаемые методом или свойством, которые имеют в сигнатуре модификатор `ref readonly` и используют `return ref`. Переменная `ref readonly` сочетает свойства локальной переменной `ref` и переменной `readonly`. Это псевдоним для хранилища, которому она присвоена, который не может изменять свое значение.

Пример использования возвращаемых ссылочных значений и ссылочных локальных переменных

В следующем примере определяется класс `Book`, содержащий два поля String: `Title` и `Author`. Также определяется класс `BookCollection`, который включает частный массив объектов `Book`. Отдельные объекты книг возвращаются по ссылке путем вызова метода `GetBookByTitle`.

```
public class Book
{
    public string Author;
    public string Title;
}

public class BookCollection
{
    private Book[] books = { new Book { Title = "Call of the Wild, The", Author = "Jack London" },
                        new Book { Title = "Tale of Two Cities, A", Author = "Charles Dickens" }
                       };
    private Book nobook = null;

    public ref Book GetBookByTitle(string title)
    {
        for (int ctr = 0; ctr < books.Length; ctr++)
        {
            if (title == books[ctr].Title)
                return ref books[ctr];
        }
        return ref nobook;
    }

    public void ListBooks()
    {
        foreach (var book in books)
        {
            Console.WriteLine($"{book.Title}, by {book.Author}");
        }
        Console.WriteLine();
    }
}
```

Если вызывающий объект сохраняет значение, возвращаемое методом `GetBookByTitle`, в качестве ссылочной локальной переменной, изменения, которые этот объект вносит в возвращаемое значение, отражаются в объекте `BookCollection`, как показано в следующем примере.

```
var bc = new BookCollection();
bc.ListBooks();

ref var book = ref bc.GetBookByTitle("Call of the Wild, The");
if (book != null)
    book = new Book { Title = "Republic, The", Author = "Plato" };
bc.ListBooks();
// The example displays the following output:
//       Call of the Wild, The, by Jack London
//       Tale of Two Cities, A, by Charles Dickens
//
//       Republic, The, by Plato
//       Tale of Two Cities, A, by Charles Dickens
```

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Написание безопасного и эффективного кода
- Возвращаемые значения ref и локальные переменные ref
- Условное выражение REF
- Передача параметров
- Параметры методов
- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#

Модификатор параметров out (справочник по C#)

18.12.2021 • 3 minutes to read

Ключевое `out` инициирует передачу аргументов по ссылке. В результате этот формальный параметр становится псевдонимом для аргумента, который должен представлять собой переменную. Другими словами, любая операция в параметре осуществляется с аргументом. Оно схоже с ключевым словом ref за исключением того, что при использовании `ref` перед передачей переменную необходимо инициализировать. Оно также похоже на ключевое слово in за исключением того, что `in` не позволяет вызываемому методу изменять значение аргумента. Для применения параметра `out` определение метода и метод вызова должны явно использовать ключевое слово `out`. Пример:

```
int initializeInMethod;
OutArgExample(out initializeInMethod);
Console.WriteLine(initializeInMethod);     // value is now 44

void OutArgExample(out int number)
{
    number = 44;
}
```

NOTE

Ключевое слово `out` также можно использовать с параметром универсального типа для указания на то, что тип параметра является ковариантным. Дополнительные сведения об использовании ключевого слова `out` в этом контексте см. в разделе out (универсальный модификатор).

Переменные, передаваемые в качестве аргументов `out`, не требуется инициализировать перед передачей в вызове метода. Но перед передачей управления из вызванного метода он должен присвоить значение.

Ключевые слова `in`, `ref` и `out` не считаются частью сигнатуры метода для разрешения перегрузки. Таким образом, методы не могут быть перегружены, если единственное различие состоит в том, что один метод принимает аргумент `ref` или `in`, а другой — `out`. Следующий код, например, компилироваться не будет.

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on ref and out".
    public void SampleMethod(out int i) { }
    public void SampleMethod(ref int i) { }
}
```

Перегрузка допустима, если один метод принимает аргумент `ref`, `in` или `out`, а другой не использует ни один из этих модификаторов, как показано ниже.

```
class OutOverloadExample
{
    public void SampleMethod(int i) { }
    public void SampleMethod(out int i) => i = 5;
}
```

Компилятор выбирает наиболее подходящую перегрузку, сравнивая модификаторы параметров в месте вызова с модификаторами параметров в вызове метода.

Свойства не являются переменными и поэтому не могут быть переданы как параметры `out`.

Ключевые слова `in`, `ref` и `out` запрещено использовать для следующих типов методов.

- Асинхронные методы, которые определяются с помощью модификатора async.
- Методы итератора, которые включают оператор yield return или `yield break`.

Кроме того, методы расширения имеют следующие ограничения:

- Ключевое слово `out` нельзя использовать в первом аргументе метода расширения.
- Ключевое слово `ref` нельзя использовать в первом аргументе метода расширения, если аргумент не является структурой, или если универсальный тип не ограничен структурой.
- Ключевое слово `in` нельзя использовать, если только первый аргумент не является структурой. Ключевое слово `in` нельзя использовать ни для одного универсального типа, даже если оно ограничено структурой.

Объявление параметров `out`

Объявление метода с аргументами `out` — стандартное решение для возвращения нескольких значений. Начиная с версии C# 7.0, вы можете использовать кортежи значений для таких сценариев. В следующем примере используется `out` для возвращения трех переменных с помощью вызова одного метода. Третьему аргументу присвоено значение NULL. Это позволяет методам возвращать значения по желанию.

```
void Method(out int answer, out string message, out string stillNull)
{
    answer = 44;
    message = "I've been returned";
    stillNull = null;
}

int argNumber;
string argMessage, argDefault;
Method(out argNumber, out argMessage, out argDefault);
Console.WriteLine(argNumber);
Console.WriteLine(argMessage);
Console.WriteLine(argDefault == null);

// The example displays the following output:
//      44
//      I've been returned
//      True
```

Вызов метода с аргументом `out`

В C# 6 и более ранних версиях необходимо было объявлять переменную в отдельном операторе, прежде

чем передавать ее как аргумент `out`. В следующем примере переменная `number` объявляется перед передачей в метод Int32.TryParse, который пытается преобразовать строку в число.

```
string numberAsString = "1640";

int number;
if (Int32.TryParse(numberAsString, out number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
// The example displays the following output:
//       Converted '1640' to 1640
```

Начиная с C# версии 7.0 переменную `out` можно объявлять в списке аргументов вызова метода, а не отдельно. Это делает код более кратким и удобным для восприятия, а также предотвращает непреднамеренное присвоение значения переменной перед вызовом метода. Следующий пример аналогичен предыдущему за тем исключением, что переменная `number` объявляется в вызове метода Int32.TryParse.

```
string numberAsString = "1640";

if (Int32.TryParse(numberAsString, out int number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
// The example displays the following output:
//       Converted '1640' to 1640
```

В предыдущем примере переменная `number` строго типизирована как `int`. Вы также можете объявить неявно типизированную локальную переменную, как в приведенном ниже примере.

```
string numberAsString = "1640";

if (Int32.TryParse(numberAsString, out var number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
// The example displays the following output:
//       Converted '1640' to 1640
```

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Параметры методов

namespace

18.12.2021 • 2 minutes to read

Ключевое слово `namespace` используется для объявления области действия, которая содержит набор связанных объектов. Пространство имен можно использовать для организации элементов кода и для создания глобально уникальных типов.

```
namespace SampleNamespace
{
    class SampleClass { }

    interface ISampleInterface { }

    struct SampleStruct { }

    enum SampleEnum { a, b }

    delegate void SampleDelegate(int i);

    namespace Nested
    {
        class SampleClass2 { }
    }
}
```

Объявления пространств имен с областью действия файла позволяют объявить, что все типы в файле находятся в одном пространстве имен. Объявления пространств имен с областью действия файла доступны в C# 10. Следующий пример похож на предыдущий, но здесь используется объявление пространства имен с областью действия файла:

```
using System;

namespace SampleFileScopedNamespace;

class SampleClass { }

interface ISampleInterface { }

struct SampleStruct { }

enum SampleEnum { a, b }

delegate void SampleDelegate(int i);
```

В предыдущем примере не было вложенного пространства имен. Пространства имен с областью действия файла не могут включать дополнительные объявления пространства имен. Вложенное пространство имен или второе пространство имен с областью действия файла объявить нельзя:

```
namespace SampleNamespace;

class AnotherSampleClass
{
    public void AnotherSampleMethod()
    {
        System.Console.WriteLine(
            "SampleMethod inside SampleNamespace");
    }
}

namespace AnotherNamespace; // Not allowed!

namespace ANestedNamespace // Not allowed!
{
   // declarations...
}
```

В пространстве имен можно объявить ноль или больше следующих типов:

- class
- interface
- struct
- enum
- delegate
- вложенные пространства имен можно объявлять везде, кроме объявлений пространств имен с областью действия файла

Компилятор добавляет пространство имен по умолчанию. Это безымянное пространство имен, иногда называемое глобальным пространством имен, существует в каждом файле. Он содержит объявления, не включенные в объявленное пространство имен. Любой идентификатор в глобальном пространстве имен доступен для использования в именованном пространстве имен.

Об этом не говорится прямо, но пространства имен имеют открытый доступ. Описание модификаторов доступа, которые можно назначить элементам в пространстве имен, см. в разделе Модификаторы доступа.

Пространство имен можно определить в двух или нескольких объявлениях. Например, в следующем примере два класса определяются в качестве части пространства имен `MyCompany`:

```
namespace MyCompany.Proj1
{
    class MyClass
    {
    }
}

namespace MyCompany.Proj1
{
    class MyClass1
    {
    }
}
```

В следующем примере показано, как можно вызвать статический метод вложенного пространства имен.

```
namespace SomeNameSpace
{
    public class MyClass
    {
        static void Main()
        {
            Nested.NestedNameSpaceClass.SayHello();
        }
    }

    // a nested namespace
    namespace Nested
    {
        public class NestedNameSpaceClass
        {
            public static void SayHello()
            {
                Console.WriteLine("Hello");
            }
        }
    }
}
// Output: Hello
```

Спецификация языка C#

Дополнительные сведения см. в статье Пространства имен в разделе Предварительная спецификация C# 6.0. Дополнительные сведения об объявлениях пространств имен с областью действия файла см. в спецификации функций.

См. также раздел

- справочник по C#
- Ключевые слова C#
- using
- using static
- Квалификатор псевдонима пространства имен `::`
- Пространства имен

using (справочник по C#)

18.12.2021 • 2 minutes to read

Ключевое слово `using` имеет три основных применения:

- Инструкция using определяет область, по завершении которой объект удаляется.
- Директива using создает псевдоним для пространства имен или импортирует типы, определенные в других пространствах имен.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Пространства имен
- extern

using - директива

18.12.2021 • 8 minutes to read

Директива `using` позволяет использовать типы, определенные в пространстве имен, без указания полного пространства имен этого типа. В базовой форме директива `using` импортирует все типы из одного пространства имен, как показано в следующем примере:

```
using System.Text;
```

К директиве `using` можно применить два модификатора:

- Модификатор `global` действует так же, как и добавление одной директивы `using` к каждому исходному файлу в проекте. Впервые этот модификатор появился в C# 10.
- Модификатор `static` импортирует элементы `static` и вложенные типы из одного типа, а не из всех типов в пространстве имен. Впервые этот модификатор появился в C# 6.0.

Оба модификатора можно применить вместе для импорта статических членов из определенного типа во всех исходных файлах проекта.

Теперь можно также создать псевдоним для пространства имен или типа, *используя директиву псевдонимов using*.

```
using Project = PC.MyCompany.Project;
```

Модификатор `global` можно использовать в *директиве using alias*.

> **NOTE**
> Ключевое слово `using` также используется для создания *операторов using*, которые помогают обеспечить правильную обработку объектов IDisposable, таких как файлы и шрифты. Дополнительные сведения об *операторе using* см. в разделе Оператор using.

Область директивы `using` без модификатора `global` ограничена файлом, в котором она находится.

Директива `using` может отображаться:

- В начале файла исходного кода перед объявлениями пространств имен и типов.
- В любом пространстве имен, но до любых пространств имен или типов, объявленных в этом пространстве имен, если только не используется модификатор `global`, если он используется, директива должна располагаться перед всеми объявлениями пространств имен и типов.

В противном случае возникнет ошибка компилятора CS1529.

Создайте директиву `using`, чтобы использовать типы в пространстве имен без необходимости указания этого пространства имен. Директива `using` не предоставляет доступ ни к одному из пространств имен, вложенных в указанное пространство имен. Пространства имен делятся на две категории: пользовательские и системные. Пользовательские пространства имен задаются в вашем коде. Список системных пространств имен см. в разделе Браузер API .NET.

Модификатор global

Добавление модификатора `global` к директиве `using` означает, что директива using должна применяться ко всем файлам в компиляции (обычно это проект). Директива `global using` впервые появилась в C# 10. Синтаксис:

```
global using <fully-qualified-namespace>;
```

где *fully-qualified-namespace* — это полное доменное имя пространства имен, на типы которого можно ссылаться без указания пространства имен.

Директива *global using* может находиться в начале любого файла исходного кода. Все директивы `global using` в одном файле должны предшествовать:

- Всем директивам `using` без модификатора `global`.
- Всем объявлениям пространств имен и типов в файле.

Директивы `global using` можно добавлять в любой исходный файл. Как правило, их хранят в одном месте. Порядок директив `global using` не имеет значения ни в одном файле, ни в нескольких.

Модификатор `global` можно сочетать с модификатором `static`. Модификатор `global` может применяться к *директиве using alias*. В обоих случаях директива применяется ко всем файлам в текущей компиляции. Следующий пример включает использование всех методов, объявленных в System.Math во всех файлах проекта:

```
global using static System.Math;
```

Вы также можете включить пространства имен глобально, добавив элемент `<Using>` в файл проекта, например так: `<Using Include="My.Awesome.Namespace" />`. Дополнительные сведения см. в разделе, посвященном элементу `<Using>`.

> **IMPORTANT**
> В шаблонах C# для .NET 6 используются *операторы верхнего уровня*. Ваше приложение может не соответствовать коду в этой статье, если вы уже выполнили обновление до предварительных версий .NET 6. Дополнительные сведения см. в статье Новые шаблоны C# для создания инструкций верхнего уровня.
>
> Пакет SDK для .NET 6 также добавляет набор *неявных* директив `global using` для проектов, использующих следующие пакеты SDK:
>
> - Microsoft.NET.Sdk
> - Microsoft.NET.Sdk.Web
> - Microsoft.NET.Sdk.Worker
>
> Эти неявные директивы `global using` включают наиболее распространенные пространства имен для соответствующего типа проектов.

static - модификатор

Директива `using static` указывает тип, доступ к статическим членам и вложенным типам которого можно получить, не указывая имя типа. Директива `using static` была представлена в C# версии 6. Синтаксис:

```
using static <fully-qualified-type-name>;
```

`<fully-qualified-type-name>` — это имя типа, на статические члены и вложенные типы которого можно ссылаться, не указывая имя типа. Если полное доменное имя (полное имя пространства имен вместе с именем типа) не указано, C# создает ошибку компилятора CS0246: "Тип или пространство имен 'type/namespace' не найдены (отсутствует директива using или ссылка сборки)".

Директива `using static` применяется к каждому типу, у которого есть статические члены (или вложенные типы), даже если при этом у него также имеются члены экземпляров. При этом для вызова членов экземпляров можно использовать только экземпляр типа.

Вы можете обращаться к статическим членам типа без необходимости квалификации доступа с помощью имени типа:

```
using static System.Console;
using static System.Math;
class Program
{
    static void Main()
    {
        WriteLine(Sqrt(3*3 + 4*4));
    }
}
```

Обычно при вызове статического члена необходимо указать имя типа и имя нужного члена. Повторный ввод одного и того же имени типа для вызова относящихся к нему элементов может сделать код слишком длинным и сложным. Например, следующее определение класса `Circle` ссылается на многие члены класса Math.

```
using System;

public class Circle
{
   public Circle(double radius)
   {
      Radius = radius;
   }

   public double Radius { get; set; }

   public double Diameter
   {
      get { return 2 * Radius; }
   }

   public double Circumference
   {
      get { return 2 * Radius * Math.PI; }
   }

   public double Area
   {
      get { return Math.PI * Math.Pow(Radius, 2); }
   }
}
```

Поскольку явно ссылаться на класс Math при каждой ссылке на член не требуется, директива `using static` создает более понятный код:

```
using System;
using static System.Math;

public class Circle
{
   public Circle(double radius)
   {
      Radius = radius;
   }

   public double Radius { get; set; }

   public double Diameter
   {
      get { return 2 * Radius; }
   }

   public double Circumference
   {
      get { return 2 * Radius * PI; }
   }

   public double Area
   {
      get { return PI * Pow(Radius, 2); }
   }
}
```

`using static` импортирует только доступные статические члены и вложенные типы, объявленные в указанном типе. Унаследованные члены не импортируются. Можно импортировать из любого именованного типа с помощью директивы `using static`, включая модули Visual Basic. Если функции F# верхнего уровня отображаются в метаданных как статические члены именованного типа, имя которого является допустимым идентификатором C#, то эти функции F# можно импортировать.

`using static` делает методы расширения, объявленные в указанном типе, доступными для поиска метода расширения. Тем не менее имена методов расширения не импортируются в область для неквалифицированной ссылки в коде.

Методы с тем же именем, импортированные из различных типов разными директивами `using static` в том же блоке компиляции или пространстве имен, формируют группу методов. Разрешение перегрузки в этих группах методов соответствует обычным правилам языка C#.

В следующем примере директива `using static` используется для того, чтобы доступ к статическим членам классов Console, Math и String можно было получать, не указывая имя типа.

```
using System;
using static System.Console;
using static System.Math;
using static System.String;

class Program
{
   static void Main()
   {
      Write("Enter a circle's radius: ");
      var input = ReadLine();
      if (!IsNullOrEmpty(input) && double.TryParse(input, out var radius)) {
         var c = new Circle(radius);

         string s = "\nInformation about the circle:\n";
         s = s + Format("   Radius: {0:N2}\n", c.Radius);
         s = s + Format("   Diameter: {0:N2}\n", c.Diameter);
         s = s + Format("   Circumference: {0:N2}\n", c.Circumference);
         s = s + Format("   Area: {0:N2}\n", c.Area);
         WriteLine(s);
      }
      else {
         WriteLine("Invalid input...");
      }
   }
}

public class Circle
{
   public Circle(double radius)
   {
      Radius = radius;
   }

   public double Radius { get; set; }

   public double Diameter
   {
      get { return 2 * Radius; }
   }

   public double Circumference
   {
      get { return 2 * Radius * PI; }
   }

   public double Area
   {
      get { return PI * Pow(Radius, 2); }
   }
}
// The example displays the following output:
//       Enter a circle's radius: 12.45
//
//       Information about the circle:
//          Radius: 12.45
//          Diameter: 24.90
//          Circumference: 78.23
//          Area: 486.95
```

В этом примере директива `using static` может также применяться к типу Double. Добавление этой директивы позволит вызывать метод TryParse(String, Double), не указывая имя типа. При этом, когда `TryParse` используется без указания имени типа, код становится менее понятным, поскольку появляется необходимость проверять директивы `using static` и определять, какой метод числового типа `TryParse` вызывается.

`using static` также применяется к типам `enum`. Если вы добавите `using static` с перечислением, в типе уже не обязательно использовать элементы перечисления.

```
using static Color;

enum Color
{
    Red,
    Green,
    Blue
}

class Program
{
    public static void Main()
    {
        Color color = Green;
    }
}
```

Псевдоним using

Создайте директиву псевдонима `using`, чтобы упростить квалификацию идентификатора для пространства имен или типа. Во всех директивах `using` следует использовать полное пространство имен или тип независимо от того, какие директивы `using` находятся перед ней. В объявлении директивы `using` не может использоваться псевдоним `using`. Например, следующий код вызывает ошибку компиляции:

```
using s = System.Text;
using s.RegularExpressions; // Generates a compiler error.
```

В следующем примере показано, как задать и использовать псевдоним `using` для пространства имен.

```
namespace PC
{
    // Define an alias for the nested namespace.
    using Project = PC.MyCompany.Project;
    class A
    {
        void M()
        {
            // Use the alias
            var mc = new Project.MyClass();
        }
    }
    namespace MyCompany
    {
        namespace Project
        {
            public class MyClass { }
        }
    }
}
```

Псевдоним using не может иметь открытый универсальный тип с правой стороны. Например, нельзя создать псевдоним using для `List<T>`, но можно создать его для `List<int>`.

В следующем примере показано, как задать директиву `using` и псевдоним `using` для класса.

```
using System;

// Using alias directive for a class.
using AliasToMyClass = NameSpace1.MyClass;

// Using alias directive for a generic class.
using UsingAlias = NameSpace2.MyClass<int>;

namespace NameSpace1
{
    public class MyClass
    {
        public override string ToString()
        {
            return "You are in NameSpace1.MyClass.";
        }
    }
}

namespace NameSpace2
{
    class MyClass<T>
    {
        public override string ToString()
        {
            return "You are in NameSpace2.MyClass.";
        }
    }
}

namespace NameSpace3
{
    class MainClass
    {
        static void Main()
        {
            var instance1 = new AliasToMyClass();
            Console.WriteLine(instance1);

            var instance2 = new UsingAlias();
            Console.WriteLine(instance2);
        }
    }
}
// Output:
//    You are in NameSpace1.MyClass.
//    You are in NameSpace2.MyClass.
```

Использование пространства имен `My` из Visual Basic

Пространство имен Microsoft.VisualBasic.MyServices (`My` в Visual Basic) обеспечивает простой и интуитивно понятный доступ к ряду классов .NET, позволяя создавать код, взаимодействующий с компьютером, приложением, параметрами, ресурсами и т. д. Пространство имен `MyServices` изначально разработано для Visual Basic, однако может применяться и в приложениях C#.

Дополнительные сведения об использовании пространства имен `MyServices` из Visual Basic см. в разделе Разработка с использованием пространства имен My.

Необходимо добавить ссылку на сборку *Microsoft.VisualBasic.dll* в проект. Некоторые классы из пространства имен `MyServices` нельзя вызывать из приложения C#, как, например, несовместимый класс FileSystemProxy. Конкретно в этом случае вместо него можно использовать статические методы из состава FileSystem (также входит в библиотеку VisualBasic.dll). Например, ниже показано, как дублировать

каталог с помощью одного из таких методов:

```
// Duplicate a directory
Microsoft.VisualBasic.FileIO.FileSystem.CopyDirectory(
    @"C:\original_directory",
    @"C:\copy_of_original_directory");
```

Спецификация языка C#

Дополнительные сведения см. в разделе Директивы using в статье Спецификация языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

Дополнительные сведения об использовании модификатора *global using* см. в спецификации функции global using — C# 10.

См. также раздел

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Пространства имен
- Оператор using

Оператор using (справочник по C#)

18.12.2021 • 3 minutes to read

Предоставляет удобный синтаксис, обеспечивающий правильное использование объектов IDisposable. Начиная с C# 8.0 инструкция `using` гарантирует правильное использование объектов IAsyncDisposable.

Пример

В следующем примере показано использование оператора `using`.

```
string manyLines = @"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line.";

using (var reader = new StringReader(manyLines))
{
    string? item;
    do
    {
        item = reader.ReadLine();
        Console.WriteLine(item);
    } while (item != null);
}
```

Начиная с версии C# 8.0, можно использовать следующий альтернативный синтаксис для оператора `using` без использования фигурных скобок:

```
string manyLines = @"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line.";

using var reader = new StringReader(manyLines);
string? item;
do
{
    item = reader.ReadLine();
    Console.WriteLine(item);
} while (item != null);
```

Примечания

File и Font представляют собой примеры управляемых типов, которые обращаются к неуправляемым ресурсам (в данном случае это обработчики файлов и контексты устройств). Существуют и многие другие виды неуправляемых ресурсов и типов библиотек классов, которые их инкапсулируют. Все эти типы должны реализовывать интерфейс IDisposable или интерфейс IAsyncDisposable.

Когда время существования объекта `IDisposable` ограничено одним методом, необходимо объявить его и создать его экземпляр в операторе `using`. Оператор `using` правильно вызывает метод Dispose для объектов, а также (если он используется, как показано выше) становится причиной выхода объекта из области действия, как только вызывается метод Dispose. В блоке `using` объект доступен только для

чтения, и изменить или переназначить его невозможно. Если объект реализует `IAsyncDisposable` вместо `IDisposable`, то инструкция `using` вызывает DisposeAsync и `awaits`, возвращенный ValueTask. Дополнительные сведения о IAsyncDisposable см. в статье Реализация метода DisposeAsync.

Использование инструкции `using` обеспечивает вызов Dispose (или DisposeAsync), даже если в блоке `using` возникает исключение. Тот же результат можно получить, поместив объект в блок `try`, а затем вызвав Dispose (или DisposeAsync) в блоке `finally`; фактически компилятор переводит инструкцию `using` именно так. Во время компиляции представленный выше код разворачивается в следующий (обратите внимание на дополнительные фигурные скобки, создающие ограниченную область действия для объекта):

```
string manyLines = @"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line.";

{
    var reader = new StringReader(manyLines);
    try
    {
        string? item;
        do
        {
            item = reader.ReadLine();
            Console.WriteLine(item);
        } while (item != null);
    }
    finally
    {
        reader?.Dispose();
    }
}
```

Новый синтаксис инструкции `using` преобразуется в похожий код. Блок `try` открывается там, где объявлена переменная. Блок `finally` добавляется в конец охватывающего блока, как правило, в конце метода.

Дополнительные сведения об инструкции `try` - `finally` см. в описании try-finally.

В одной инструкции `using` можно объявить сразу несколько экземпляров типа, как показано в следующем примере. Обратите внимание, что невозможно использовать неявно типизированные переменные (`var`) при объявлении нескольких переменных в одной инструкции:

```
string numbers = @"One
Two
Three
Four.";
string letters = @"A
B
C
D.";

using (StringReader left = new StringReader(numbers),
    right = new StringReader(letters))
{
    string? item;
    do
    {
        item = left.ReadLine();
        Console.Write(item);
        Console.Write("    ");
        item = right.ReadLine();
        Console.WriteLine(item);
    } while (item != null);
}
```

Несколько объявлений одного типа можно объединить с помощью нового синтаксиса, представленного в C# 8, как показано в следующем примере:

```
string numbers = @"One
Two
Three
Four.";
string letters = @"A
B
C
D.";

using StringReader left = new StringReader(numbers),
    right = new StringReader(letters);
string? item;
do
{
    item = left.ReadLine();
    Console.Write(item);
    Console.Write("    ");
    item = right.ReadLine();
    Console.WriteLine(item);
} while (item != null);
```

Вы можете создать экземпляр объекта ресурсов, а затем передать переменную в инструкцию `using`, однако делать это не рекомендуется. В этом случае объект остается в области действия и после того, как блок `using` теряет контроль, хотя доступа к неуправляемым ресурсам у него, скорее всего, не будет. Иными словами, он уже не полностью инициализируется. При попытке использовать объект за пределами блока `using` может возникнуть исключение. По этой причине лучше создать экземпляр объекта в инструкции `using` и ограничить область действия блоком `using`.

```
string manyLines = @"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line.";

var reader = new StringReader(manyLines);
using (reader)
{
    string? item;
    do
    {
        item = reader.ReadLine();
        Console.WriteLine(item);
    } while (item != null);
}
// reader is in scope here, but has been disposed
```

Дополнительные сведения об утилизации объектов `IDisposable` см. в разделе Использование объектов, реализующих IDisposable.

Спецификация языка C#

Дополнительные сведения см. в разделе Оператор using в статье Спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Директива using
- Сборка мусора
- Использование объектов, реализующих IDisposable
- Интерфейс IDisposable
- Инструкция using в C# 8.0

Псевдоним extern (Справочник по C#)

18.12.2021 • 2 minutes to read

Иногда может потребоваться сослаться на две версии сборок, которые имеют одинаковые полные имена типов. Например, если необходимо использовать две или более версии сборки в одном приложении. Используя внешний псевдоним сборки, пространства имен для каждой сборки можно перенести внутрь пространств имен корневого уровня с именованием по псевдониму, что позволяет использовать их в одном файле.

> **NOTE**
> Ключевое слово extern также используется в качестве модификатора метода, объявляющего метод, написанный в неуправляемом коде.

Для ссылки на две сборки с одинаковыми полными именами типов псевдоним необходимо указать в командной строке следующим образом:

```
/r:GridV1=grid.dll
```

```
/r:GridV2=grid20.dll
```

При этом создаются внешние псевдонимы `GridV1` и `GridV2`. Чтобы использовать эти псевдонимы в программе, сошлитесь на них с помощью ключевого слова `extern`. Пример:

```
extern alias GridV1;
```

```
extern alias GridV2;
```

Каждое объявление псевдонима extern создает дополнительное пространство имен корневого уровня, которое является параллельным для глобального пространства имен (но не находится в его пределах). Таким образом, на типы из каждой сборки можно ссылаться без неоднозначности, используя их полное имя, размещенное в соответствующем пространстве имен-псевдониме.

В предыдущем примере `GridV1::Grid` является элементом управления сетки из `grid.dll`, а `GridV2::Grid` — элементом управления сетки из `grid20.dll`.

Использование Visual Studio

Если вы используете Visual Studio, псевдонимы можно указать аналогичным образом.

Добавьте ссылку на *grid.dll* и *grid20.dll* в проект в Visual Studio. Откройте вкладку свойств и измените псевдонимы с глобальных на GridV1 и GridV2 соответственно.

Используйте эти псевдонимы так же, как описано выше.

```
extern alias GridV1;

extern alias GridV2;
```

Теперь можно создать псевдоним для пространства имен или типа, *используя директиву псевдонимов using*. Дополнительные сведения см. в разделе Директива using.

```
using Class1V1 = GridV1::Namespace.Class1;

using Class1V2 = GridV2::Namespace.Class1;
```

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- :: Оператор
- **References** (параметры компилятора C#)

Ограничение new (справочник по C#)

18.12.2021 • 2 minutes to read

Ограничение `new` указывает, что аргумент типа в объявлении универсального класса должен иметь открытый конструктор без параметров. Использовать ограничение `new` можно только в том случае, если тип не является абстрактным.

Примените ограничение `new` к параметру типа, когда общий класс создает новые экземпляры этого типа, как показано в следующем примере:

```
class ItemFactory<T> where T : new()
{
    public T GetNewItem()
    {
        return new T();
    }
}
```

При использовании ограничения `new()` с другими ограничениями его необходимо указывать последним:

```
public class ItemFactory2<T>
    where T : IComparable, new()
{  }
```

Дополнительные сведения см. в разделе Ограничения параметров типа.

Ключевое слово `new` можно также использовать для создания экземпляра типа или как модификатор объявления члена.

Спецификация языка C#

Дополнительные сведения см. в разделе Ограничения параметров типа в спецификации языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Универсальные шаблоны

where (ограничение универсального типа) (справочник по C#)

18.12.2021 • 4 minutes to read

Предложение `where` в универсальном определении задает ограничения на типы, которые используются в качестве аргументов для параметров типа в универсальном типе, методе, делегате или локальной функции. Ограничения могут задавать интерфейсы, базовые классы или требовать, чтобы универсальный тип был ссылочным типом, типом значения или неуправляемым типом. Они объявляют характеристики, которыми должен обладать аргумент типа.

Например, можно объявить универсальный класс `AGenericClass` так, чтобы параметр типа `T` реализовывал интерфейс IComparable<T>:

```
public class AGenericClass<T> where T : IComparable<T> { }
```

> **NOTE**
> Дополнительные сведения о предложении where в выражении запроса см. в разделе Предложение where.

Предложение `where` также может включать ограничение базового класса. Ограничение базового класса указывает, что тип, который должен использоваться как аргумент типа для этого универсального типа, имеет заданный класс в качестве базового класса или является этим базовым классом. Если ограничение базового класса используется, оно должно быть указано перед любыми другими ограничениями данного параметра типа. Некоторые типы не могут использоваться как ограничение базового класса: Object, Array и ValueType. До C# 7.3 Enum, Delegate и MulticastDelegate также не могли использоваться в качестве ограничений базового класса. Ниже приведен пример типов, которые теперь можно указать как базовый класс:

```
public class UsingEnum<T> where T : System.Enum { }

public class UsingDelegate<T> where T : System.Delegate { }

public class Multicaster<T> where T : System.MulticastDelegate { }
```

В контексте, допускающем значения NULL, в C# 8.0 и более поздних версиях принудительно применяется допустимость значений NULL для типа базового класса. Если базовый класс не допускает значения NULL (например, `Base`), аргумент типа должен иметь значение, отличное от NULL. Если базовый класс допускает значения NULL (например, `Base?`), аргумент типа может быть ссылочным типом, допускающим или не допускающим значения NULL. Компилятор выдает предупреждение, если аргумент типа является ссылочным типом, допускающим значения NULL, когда базовый класс не допускает значения NULL.

Предложение `where` может указывать, что тип является `class` или `struct`. Ограничение `struct` избавляет от необходимости указывать ограничение базового класса `System.ValueType`. Тип `System.ValueType` не может использоваться как ограничение базового класса. Ограничения `class` и `struct` показаны в следующем примере:

```
class MyClass<T, U>
    where T : class
    where U : struct
{ }
```

В контексте, допускающем значения NULL, в C# 8.0 и более поздних версиях для ограничения `class` требуется тип, являющийся ссылочным типом, не допускающим значения NULL. Чтобы разрешить ссылочные типы, допускающие значения NULL, используйте ограничение `class?`, которое разрешает ссылочные типы, допускающие и не допускающие значения NULL.

Предложение `where` может включать ограничение `notnull`. Ограничение `notnull` ограничивает параметр типа типами, допускающими значение NULL. Тип может быть типом значения или ссылочным типом, не допускающим значение NULL. Ограничение `notnull` доступно начиная с C# 8.0 для кода, скомпилированного в `nullable enable` контексте. В отличие от других ограничений, если аргумент типа нарушает ограничение `notnull`, компилятор генерирует предупреждение вместо ошибки. Предупреждения генерируются только в контексте `nullable enable`.

Добавление ссылочных типов, допускающих значения NULL, приводит к возможной неоднозначности значения `T?` в универсальных методах. Если `T` имеет значение `struct`, то `T?` соответствует System.Nullable<T>. Но если `T` имеет ссылочный тип, для `T?` допустимо значение `null`. Так как переопределяющиеся методы не могут включать в себя ограничения, возникает неоднозначность. Новое ограничение `default` устраняет ее. Его нужно добавить, если базовый класс или интерфейс объявляет две перегрузки метода, в одной из которых задано ограничение `struct`, а в другой не указано ни `struct`, ни `class`:

```
public abstract class B
{
    public void M<T>(T? item) where T : struct { }
    public abstract void M<T>(T? item);

}
```

Используйте ограничение `default`, чтобы указать, что производный класс переопределяет метод без ограничения в производном классе, или явную реализацию интерфейса. Оно допустимо только для методов, переопределяющих базовые методы, или явных реализаций интерфейса:

```
public class D : B
{
    // Without the "default" constraint, the compiler tries to override the first method in B
    public override void M<T>(T? item) where T : default { }
}
```

> **IMPORTANT**
> Универсальные объявления, включающие ограничение `notnull`, можно использовать в обнуляемом контексте, допускающем значение NULL, но компилятор не применяет ограничение.

```
#nullable enable
    class NotNullContainer<T>
        where T : notnull
    {
    }
#nullable restore
```

Предложение `where` также может включать ограничение `unmanaged`. Ограничение `unmanaged` позволяет использовать в качестве параметра типа только типы, называемые неуправляемыми типами. Ограничение `unmanaged` упрощает написание кода взаимодействия низкого уровня на языке C#. Это ограничение включает подпрограммы с возможностью повторного использования для всех неуправляемых типов. Ограничение `unmanaged` нельзя использовать с ограничением `class` или `struct`. Ограничение `unmanaged` требует тип `struct`:

```
class UnManagedWrapper<T>
    where T : unmanaged
{ }
```

Предложение `where` также может включать ограничение конструктора, `new()`. Это ограничение позволяет создать экземпляр параметра типа с помощью оператора `new`. Ограничение new() сообщает компилятору о том, что все предоставленные аргументы типа должны иметь доступный конструктор без параметров. Пример:

```
public class MyGenericClass<T> where T : IComparable<T>, new()
{
    // The following line is not possible without new() constraint:
    T item = new T();
}
```

Ограничение `new()` отображается в предложении `where` последним. Ограничение `new()` не может использоваться с ограничениями `struct` или `unmanaged`. Все типы, удовлетворяющие этим ограничениям, должны иметь доступ к конструктору без параметров, поэтому ограничение `new()` будет избыточным.

Если параметров типа несколько, для каждого из них необходимо использовать по одному предложению `where`, например:

```
public interface IMyInterface { }

namespace CodeExample
{
    class Dictionary<TKey, TVal>
        where TKey : IComparable<TKey>
        where TVal : IMyInterface
    {
        public void Add(TKey key, TVal val) { }
    }
}
```

Кроме того, ограничения можно присоединять к параметрам типа универсальных методов следующим образом:

```
public void MyMethod<T>(T t) where T : IMyInterface { }
```

Обратите внимание на то, что ограничения параметров типа для делегатов имеют такой же синтаксис, как и методы:

```
delegate T MyDelegate<T>() where T : new();
```

Дополнительные сведения об универсальных делегатах см. в разделе Универсальные делегаты.

Дополнительные сведения о синтаксисе и применении ограничений см. в разделе Ограничения

параметров типа.

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Введение в универсальные шаблоны
- Ограничение new
- Ограничения параметров типа

base (Справочник по C#)

18.12.2021 • 2 minutes to read

Ключевое слово `base` используется для доступа к членам базового из производного класса в следующих случаях:

- Вызов метода базового класса, который был переопределен другим методом.
- Определение конструктора базового класса, который должен вызываться при создании экземпляров производного класса.

Доступ к базовому классу разрешен только в конструкторе, методе экземпляра или методе доступа к свойству экземпляра.

Использование ключевого слова `base` в статическом методе является недопустимым.

Доступ осуществляется к базовому классу, заданному в объявлении класса. Например, если указать `class ClassB : ClassA`, члены ClassA будут доступны из ClassB независимо от базового класса ClassA.

Пример 1

В приведенном ниже примере метод `Getinfo` присутствует как в базовом классе `Person`, так и в производном классе `Employee`. С помощью ключевого слова `base` можно вызвать метод `Getinfo` базового класса из производного класса.

```
public class Person
{
    protected string ssn = "444-55-6666";
    protected string name = "John L. Malgraine";

    public virtual void GetInfo()
    {
        Console.WriteLine("Name: {0}", name);
        Console.WriteLine("SSN: {0}", ssn);
    }
}
class Employee : Person
{
    public string id = "ABC567EFG";
    public override void GetInfo()
    {
        // Calling the base class GetInfo method:
        base.GetInfo();
        Console.WriteLine("Employee ID: {0}", id);
    }
}

class TestClass
{
    static void Main()
    {
        Employee E = new Employee();
        E.GetInfo();
    }
}
/*
Output
Name: John L. Malgraine
SSN: 444-55-6666
Employee ID: ABC567EFG
*/
```

Дополнительные примеры см. в разделах new, virtual и override.

Пример 2

В этом примере показано, как задать конструктор базового класса, вызываемый при создании экземпляров производного класса.

```
public class BaseClass
{
    int num;

    public BaseClass()
    {
        Console.WriteLine("in BaseClass()");
    }

    public BaseClass(int i)
    {
        num = i;
        Console.WriteLine("in BaseClass(int i)");
    }

    public int GetNum()
    {
        return num;
    }
}

public class DerivedClass : BaseClass
{
    // This constructor will call BaseClass.BaseClass()
    public DerivedClass() : base()
    {
    }

    // This constructor will call BaseClass.BaseClass(int i)
    public DerivedClass(int i) : base(i)
    {
    }

    static void Main()
    {
        DerivedClass md = new DerivedClass();
        DerivedClass md1 = new DerivedClass(1);
    }
}
/*
Output:
in BaseClass()
in BaseClass(int i)
*/
```

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- this

this (Справочник по C#)

18.12.2021 • 2 minutes to read

Ключевое слово `this` ссылается на текущий экземпляр класса, а также используется в качестве модификатора первого параметра метода расширения.

> **NOTE**
> В этой статье рассматривается использование `this` с экземплярами класса. Дополнительные сведения об использовании этого ключевого слова в методах расширения см. в разделе Методы расширения.

Ниже перечислены наиболее частые способы использования `this`.

- Для квалификации элементов, скрытых одинаковыми именами, например:

```
public class Employee
{
    private string alias;
    private string name;

    public Employee(string name, string alias)
    {
        // Use this to qualify the members of the class
        // instead of the constructor parameters.
        this.name = name;
        this.alias = alias;
    }
}
```

- Для передачи другим методам объекта в качестве параметра, например:

```
CalcTax(this);
```

- Для объявления индексаторов, например:

```
public int this[int param]
{
    get { return array[param]; }
    set { array[param] = value; }
}
```

У статических функций-членов нет указателя `this`, так как они существуют только на уровне класса и не являются частями объектов. Использование ссылки на `this` в статическом методе является недопустимым.

Пример

В этом примере `this` используется для квалификации членов класса `Employee`, `name` и `alias`, которые скрыты одинаковыми именами. Это ключевое слово также используется для передачи объекта в метод `CalcTax`, который принадлежит другому классу.

```
class Employee
{
    private string name;
    private string alias;
    private decimal salary = 3000.00m;

    // Constructor:
    public Employee(string name, string alias)
    {
        // Use this to qualify the fields, name and alias:
        this.name = name;
        this.alias = alias;
    }

    // Printing method:
    public void printEmployee()
    {
        Console.WriteLine("Name: {0}\nAlias: {1}", name, alias);
        // Passing the object to the CalcTax method by using this:
        Console.WriteLine("Taxes: {0:C}", Tax.CalcTax(this));
    }

    public decimal Salary
    {
        get { return salary; }
    }
}

class Tax
{
    public static decimal CalcTax(Employee E)
    {
        return 0.08m * E.Salary;
    }
}

class MainClass
{
    static void Main()
    {
        // Create objects:
        Employee E1 = new Employee("Mingda Pan", "mpan");

        // Display results:
        E1.printEmployee();
    }
}
/*
Output:
    Name: Mingda Pan
    Alias: mpan
    Taxes: $240.00
 */
```

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#

- Ключевые слова в C#
- base
- Методы

null (справочник по C#)

18.12.2021 • 2 minutes to read

Ключевое слово `null` является литералом, представляющим пустую ссылку, которая не ссылается на объект. `null` является значением по умолчанию для переменных ссылочного типа. Обычные типы значений не могут иметь значение NULL, за исключением типов значений, допускающих значение NULL.

В следующем примере показано возможное поведение ключевого слова `null`.

```
class Program
{
    class MyClass
    {
        public void MyMethod() { }
    }

    static void Main(string[] args)
    {
        // Set a breakpoint here to see that mc = null.
        // However, the compiler considers it "unassigned."
        // and generates a compiler error if you try to
        // use the variable.
        MyClass mc;

        // Now the variable can be used, but...
        mc = null;

        // ... a method call on a null object raises
        // a run-time NullReferenceException.
        // Uncomment the following line to see for yourself.
        // mc.MyMethod();

        // Now mc has a value.
        mc = new MyClass();

        // You can call its method.
        mc.MyMethod();

        // Set mc to null again. The object it referenced
        // is no longer accessible and can now be garbage-collected.
        mc = null;

        // A null string is not the same as an empty string.
        string s = null;
        string t = String.Empty; // Logically the same as ""

        // Equals applied to any null object returns false.
        bool b = (t.Equals(s));
        Console.WriteLine(b);

        // Equality operator also returns false when one
        // operand is null.
        Console.WriteLine("Empty string {0} null string", s == t ? "equals": "does not equal");

        // Returns true.
        Console.WriteLine("null == null is {0}", null == null);

        // A value type cannot be null
        // int i = null; // Compiler error!

        // Use a nullable value type instead:
        int? i = null;

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
```

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- справочник по C#
- Ключевые слова C#
- Значения по умолчанию типов C#
- Nothing (Visual Basic)

bool (справочник по C#)

18.12.2021 • 2 minutes to read

Ключевое слово типа `bool` — это псевдоним для типа структуры System.Boolean .NET, представляющий логическое значение: `true` или `false`.

Для выполнения логических операций со значениями типа `bool` используйте логические операторы. Тип `bool` является типом результата операторов сравнения и равенства. Выражение `bool` может быть управляющим условным выражением в операторах if, do, while и for и условном операторе `?:`.

Значение по умолчанию для типа `bool` — `false`.

Литералы

Вы можете использовать литералы `true` и `false` для инициализации переменной `bool` или передачи значения `bool`:

```
bool check = true;
Console.WriteLine(check ? "Checked" : "Not checked");  // output: Checked

Console.WriteLine(false ? "Checked" : "Not checked");  // output: Not checked
```

Трехзначная булева логика

Используйте тип `bool?`, допускающий значение NULL, если нужно использовать трехзначную логику, например, при работе с базами данных, которые поддерживают трехзначный логический тип. Для операндов `bool?` предопределенные операторы `&` и `|` поддерживают троичную логику. См. подробнее о логических операторах, поддерживающих значение NULL в описании логических операторов.

См. дополнительные сведения о типах значений, допускающих значение NULL.

Преобразования

В C# доступно только два преобразования, использующих тип `bool`. Это неявное преобразование в соответствующий тип `bool?`, допускающий значение NULL, и явное преобразование из типа `bool?`. Однако .NET предоставляет дополнительные методы, позволяющие выполнять преобразование в тип `bool` или из него. Дополнительные сведения см. в разделе о преобразовании в логические значения и из них справочника по API System.Boolean.

Спецификация языка C#

Дополнительные сведения см. в разделе Тип boolспецификации языка C#.

См. также раздел

- справочник по C#
- Типы значений
- Операторы true и false

default (справочник по C#)

18.12.2021 • 2 minutes to read

Ключевое слово `default` можно использовать в следующих контекстах:

- для указания регистра по умолчанию в инструкции `switch`;
- в качестве оператора или литерала по умолчанию для создания значения по умолчанию для типа.
- в качестве `default` ограничения типа для переопределения универсального метода или явной реализации интерфейса.

См. также раздел

- справочник по C#
- Ключевые слова C#

Контекстные ключевые слова (справочник по C#)

18.12.2021 • 2 minutes to read

Контекстное ключевое слово используется для предоставления конкретного значения в коде, но оно не является зарезервированным словом в C#. В этом разделе представлены следующие контекстные ключевые слова:

КЛЮЧЕВОЕ СЛОВО	ОПИСАНИЕ
add	Определяет метод доступа настраиваемого события, который вызывается, когда клиентский код подписывается на событие.
and	Создает шаблон, для которого выполняется условие соответствия, если оба вложенных шаблона совпадают.
async	Указывает на то, что измененный метод, лямбда-выражение или анонимный метод является асинхронным.
await	Приостанавливает асинхронный метод до тех пор, пока не будет завершена ожидаемая задача.
dynamic	Определяет ссылочный тип, который включает операции, в которых он применяется для обхода проверки типов во время компиляции.
get	Определяет метод доступа для свойства или индексатора.
global	Псевдоним глобального пространства имен, которое в противном случае не имеет имени.
init	Определяет метод доступа для свойства или индексатора.
nint	Определяет такой тип данных, как целое число собственного размера.
not (не);	Создает шаблон, для которого выполняется условие соответствия, если шаблон с отрицанием не соответствует выражению.
nuint	Определяет такой тип данных, как целое число собственного размера без знака.
или диспетчер конфигурации служб	Создает шаблон, для которого выполняется условие соответствия, если любой из вложенных шаблонов соответствует выражению.
partial	Определяет разделяемые классы, структуры и интерфейсы в одном блоке компиляции.

КЛЮЧЕВОЕ СЛОВО	ОПИСАНИЕ
record	Используется для определения типа записи.
remove	Определяет метод доступа настраиваемого события, который вызывается, когда клиентский код отменяет подписку на событие.
set	Определяет метод доступа для свойства или индексатора.
value	Используется для задания методов доступа, а также для добавления или удаления обработчиков событий.
var	Позволяет компилятору определить тип переменной, объявленной в области метода.
when	Задает условие фильтрации для блока `catch` или метки `case` оператора `switch`.
where	Добавляет ограничения в универсальное объявление. (См. также where.)
yield	Используется в блоке итератора для возвращения значения в объект перечислителя или для обозначения конца итерации.

Все ключевые слова запросов, представленные в C# 3.0, также являются контекстными. Дополнительные сведения см. в разделе Ключевые слова запроса (LINQ).

См. также

- справочник по C#
- Ключевые слова C#
- Операторы и выражения C#

add (Справочник по C#)

18.12.2021 • 2 minutes to read

Контекстное ключевое слово `add` определяет метод доступа настраиваемого события, который вызывается, когда клиентский код подписывается на событие. Если указан настраиваемый метод доступа `add`, также необходимо указать метод доступа remove.

Пример

В следующем примере показано событие с настраиваемыми методами доступа `add` и remove. Полный пример см. в статье Как реализовать события интерфейса (руководство по программированию на C#).

```
class Events : IDrawingObject
{
    event EventHandler PreDrawEvent;

    event EventHandler IDrawingObject.OnDraw
    {
        add => PreDrawEvent += value;
        remove => PreDrawEvent -= value;
    }
}
```

Как правило, настраиваемые методы доступа к событиям не используются. В большинстве сценариев достаточно методов доступа, которые автоматически создаются компилятором при объявлении события.

См. также

- События

get (Справочник по C#)

18.12.2021 • 2 minutes to read

Ключевое слово `get` определяет *метод доступа* в свойстве или индексаторе, который возвращает значение свойства или элемент индексатора. Дополнительные сведения см. в разделах Свойства, Автоматически реализуемые свойства и Индексаторы.

В приведенном ниже примере определен как метод доступа `get`, так и метод доступа `set` для свойства с именем `Seconds`. Для возвращения значения свойства в нем используется закрытое поле с именем `_seconds`.

```
class TimePeriod
{
     private double _seconds;

     public double Seconds
     {
         get { return _seconds; }
         set { _seconds = value; }
     }
}
```

Метод доступа `get` часто состоит из одного оператора, который возвращает значение, как в предыдущем примере. Начиная с C# версии 7.0 метод доступа `get` можно реализовывать как член, воплощающий выражение. В приведенном ниже примере методы доступа `get` и `set` реализуются как члены, воплощающие выражение.

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        set => _seconds = value;
    }
}
```

В простых случаях, когда методы доступа `get` и `set` свойства не выполняют никаких иных операций, кроме задания или извлечения значения в закрытом поле, можно использовать поддержку автоматически реализуемых свойств в компиляторе C#. В приведенном ниже примере `Hours` реализуется как автоматически реализуемое свойство.

```
class TimePeriod2
{
    public double Hours { get; set; }
}
```

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Свойства

init (справочник по C#)

18.12.2021 • 2 minutes to read

В C# 9 и более поздних версий ключевое слово `init` определяет *метод доступа* в свойстве или индексаторе. Метод задания только для инициализации присваивает значение свойству или элементу индексатора только во время создания объекта. Дополнительные сведения и примеры см. в разделах Свойства, Автоматически реализуемые свойства и Индексаторы.

В приведенном ниже примере определен как метод доступа `get`, так и метод доступа `init` для свойства с именем `Seconds`. Для возвращения значения свойства в нем используется закрытое поле с именем `_seconds`.

```
class InitExample
{
     private double _seconds;

     public double Seconds
     {
         get { return _seconds; }
         init { _seconds = value; }
     }
}
```

Метод доступа `init` часто состоит из одного оператора, который присваивает значение, как в предыдущем примере. Метод доступа `init` можно реализовывать как элемент, воплощающий выражение. В приведенном ниже примере методы доступа `get` и `init` реализуются как члены, воплощающие выражение.

```
class InitExampleExpressionBodied
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        init => _seconds = value;
    }
}
```

В простых случаях, когда методы доступа `get` и `init` свойства не выполняют никаких иных операций, кроме задания или извлечения значения в закрытом поле, можно использовать поддержку автоматически реализуемых свойств в компиляторе C#. В приведенном ниже примере `Hours` реализуется как автоматически реализуемое свойство.

```
class InitExampleAutoProperty
{
    public double Hours { get; init; }
}
```

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является

предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Свойства

разделяемый тип (справочник по C#)

18.12.2021 • 2 minutes to read

Определения разделяемых типов позволяют разделять определения классов, структур, интерфейсов или записей на несколько файлов.

В файле *File1.cs*:

```
namespace PC
{
    partial class A
    {
        int num = 0;
        void MethodA() { }
        partial void MethodC();
    }
}
```

Объявление в файле *File2.cs*:

```
namespace PC
{
    partial class A
    {
        void MethodB() { }
        partial void MethodC() { }
    }
}
```

Remarks

Разделение типа класса, структуры или интерфейса на несколько файлов может пригодиться при работе с крупными проектами или с автоматически созданным кодом, например предоставляемым конструктором Windows Forms. Разделяемый тип может содержать разделяемый метод. Дополнительные сведения см. в разделе Разделяемые классы и методы.

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Модификаторы
- Введение в универсальные шаблоны

Метод partial (Справочник по C#)

18.12.2021 • 2 minutes to read

Сигнатура разделяемого метода определяется в одной части разделяемого типа, а его реализация — в другой части этого типа. С помощью разделяемых методов разработчики классов могут при необходимости реализовывать ловушки методов, которые схожи с обработчиками событий. Если реализация не предоставлена, компилятор удаляет сигнатуру во время компиляции. В отношении разделяемых методов применяются следующие условия:

- Объявления должны начинаться с контекстного ключевого слова partial.
- Сигнатуры в обеих частях разделяемого типа должны совпадать.

Разделяемый метод может не иметь реализацию в следующих случаях.

- У него нет модификаторов доступа (включая private по умолчанию).
- Он возвращает значение void.
- У него нет параметров out.
- У него нет ни одного из следующих модификаторов: virtual, override, sealed, new или extern.

Любой метод, не соответствующий всем этим ограничениям (например, метод `public virtual partial void`), должен предоставлять реализацию.

В следующем примере показан разделяемый метод, определенный в двух частях разделяемого класса:

```
namespace PM
{
    partial class A
    {
        partial void OnSomethingHappened(string s);
    }

    // This part can be in a separate file.
    partial class A
    {
        // Comment out this method and the program
        // will still compile.
        partial void OnSomethingHappened(String s)
        {
            Console.WriteLine("Something happened: {0}", s);
        }
    }
}
```

Разделяемые методы также могут быть удобны в сочетании с генераторами источника. Например, можно определить регулярное выражение, используя следующий шаблон.

```
[RegexGenerated("(dog|cat|fish)")]
partial bool IsPetMatch(string input);
```

Дополнительные сведения см. в разделе Разделяемые классы и методы.

См. также

- Справочник по C#
- Тип partial

remove (Справочник по C#)

18.12.2021 • 2 minutes to read

Контекстное ключевое слово `remove` определяет метод доступа настраиваемого события, который вызывается, когда клиентский код отменяет подписку на событие. Если указан настраиваемый метод доступа `remove`, также необходимо указать метод доступа add.

Пример

В следующем примере показано событие с настраиваемыми методами доступа add и `remove`. Полный пример см. в статье Как реализовать события интерфейса (руководство по программированию на C#).

```
class Events : IDrawingObject
{
    event EventHandler PreDrawEvent;

    event EventHandler IDrawingObject.OnDraw
    {
        add => PreDrawEvent += value;
        remove => PreDrawEvent -= value;
    }
}
```

Как правило, настраиваемые методы доступа к событиям не используются. В большинстве сценариев достаточно методов доступа, которые автоматически создаются компилятором при объявлении события.

См. также

- События

set (Справочник по C#)

18.12.2021 • 2 minutes to read

Ключевое слово `set` определяет *метод доступа* в свойстве или индексаторе, который присваивает значение свойству элемента индексатора. Дополнительные сведения и примеры см. в разделах Свойства, Автоматически реализуемые свойства и Индексаторы.

В приведенном ниже примере определен как метод доступа `get`, так и метод доступа `set` для свойства с именем `Seconds`. Для возвращения значения свойства в нем используется закрытое поле с именем `_seconds`.

```
class TimePeriod
{
     private double _seconds;

     public double Seconds
     {
         get { return _seconds; }
         set { _seconds = value; }
     }
}
```

Метод доступа `set` часто состоит из одного оператора, который присваивает значение, как в предыдущем примере. Начиная с C# версии 7.0 метод доступа `set` можно реализовывать как член, воплощающий выражение. В приведенном ниже примере методы доступа `get` и `set` реализуются как члены, воплощающие выражение.

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        set => _seconds = value;
    }
}
```

В простых случаях, когда методы доступа `get` и `set` свойства не выполняют никаких иных операций, кроме задания или извлечения значения в закрытом поле, можно использовать поддержку автоматически реализуемых свойств в компиляторе C#. В приведенном ниже примере `Hours` реализуется как автоматически реализуемое свойство.

```
class TimePeriod2
{
    public double Hours { get; set; }
}
```

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#
- Свойства

when (справочник по C#)

18.12.2021 • 2 minutes to read

Чтобы указать условие фильтра в следующих контекстах, можно использовать контекстное ключевое слово `when`:

- В операторе `catch` блока try/catch или try/catch/finally.
- В качестве условия случая в операторе `switch`.
- В качестве условия случая в выражении `switch`.

`when` в операторе `catch`

Начиная с C# версии 6, `when` можно использовать в операторе `catch`, чтобы задать условие, которое должно быть истинным для выполнения обработчика определенного исключения. Он имеет следующий синтаксис:

```
catch (ExceptionType [e]) when (expr)
```

здесь *expr* представляет собой выражение, которое оценивает значение типа Boolean. Если он возвращает `true`, обработчик исключений выполняется, а если `false`, то нет.

В следующем примере используется ключевое слово `when`, которое позволяет условно выполнять обработчики для HttpRequestException в зависимости от того, какой текст содержит сообщение об исключении.

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

class Program
{
    static void Main()
    {
        Console.WriteLine(MakeRequest().Result);
    }

    public static async Task<string> MakeRequest()
    {
        var client = new HttpClient();
        var streamTask = client.GetStringAsync("https://localHost:10000");
        try
        {
            var responseText = await streamTask;
            return responseText;
        }
        catch (HttpRequestException e) when (e.Message.Contains("301"))
        {
            return "Site Moved";
        }
        catch (HttpRequestException e) when (e.Message.Contains("404"))
        {
            return "Page Not Found";
        }
        catch (HttpRequestException e)
        {
            return e.Message;
        }
    }
}
```

См. также

- Оператор try-catch
- Оператор try/catch/finally

value (Справочник по C#)

18.12.2021 • 2 minutes to read

Контекстное ключевое слово `value` используется в методе доступа `set` в объявлениях свойства и индексатора. Оно аналогично входному параметру метода. Ключевое слово `value` ссылается на значение, которое клиентский код пытается присвоить свойству или индексатору. В приведенном ниже примере класс `MyDerivedClass` имеет свойство с именем `Name`, в котором используется параметр `value` для присвоения новой строки резервному полю `name`. С точки зрения клиентского кода эта операция выглядит как простое присвоение.

```
class MyBaseClass
{
    // virtual auto-implemented property. Overrides can only
    // provide specialized behavior if they implement get and set accessors.
    public virtual string Name { get; set; }

    // ordinary virtual property with backing field
    private int _num;
    public virtual int Number
    {
        get { return _num; }
        set { _num = value; }
    }
}

class MyDerivedClass : MyBaseClass
{
    private string _name;

    // Override auto-implemented property with ordinary property
    // to provide specialized accessor behavior.
    public override string Name
    {
        get
        {
            return _name;
        }
        set
        {
            if (!string.IsNullOrEmpty(value))
            {
                _name = value;
            }
            else
            {
                _name = "Unknown";
            }
        }
    }
}
```

Дополнительные сведения см. в статьях Свойства и Индексаторы.

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- Ключевые слова в C#

yield (справочник по C#)

18.12.2021 • 4 minutes to read

Использование в операторе контекстного ключевого слова `yield` означает, что метод, оператор или метод доступа `get`, в котором присутствует это ключевое слово, является итератором. Использование `yield` для определения итератора исключает необходимость применения явного дополнительного класса (в котором содержится состояние перечисления; в качестве примера см. IEnumerator<T>) при реализации шаблонов IEnumerable и IEnumerator для пользовательского типа коллекции.

В следующем примере показаны две формы оператора `yield`.

```
yield return <expression>;
yield break;
```

Примечания

Оператор `yield return` используется для возврата каждого элемента по одному.

Последовательность, которая возвращается после выполнения метода итератора, можно использовать с помощью оператора foreach или запроса LINQ. Каждая итерация цикла `foreach` вызывает метод итератора. При достижении в методе итератора оператора `yield return` возвращается `expression` и сохраняется текущее расположение в коде. При следующем вызове функции итератора выполнение возобновляется с этого места.

Когда итератор возвращает System.Collections.Generic.IAsyncEnumerable<T>, эту последовательность можно использовать асинхронно с помощью оператора await foreach. Итерация цикла аналогична выполнению инструкции `foreach`. Разница заключается в том, что каждая итерация может быть приостановлена для асинхронной операции перед возвратом выражения для следующего элемента.

Для завершения итерации можно использовать оператор `yield break`.

Дополнительные сведения об итераторах см. в разделе Итераторы.

Методы итератора и методы доступа get

Объявление итератора должно соответствовать следующим требованиям.

- Возвращаемое значение должно иметь один из следующих типов:
  - IAsyncEnumerable<T>
  - IEnumerable<T>
  - IEnumerable
  - IEnumerator<T>
  - IEnumerator
- Объявление не должно содержать параметры in, ref или out.

Тип `yield` итератора, который возвращает IEnumerable или IEnumerator, — `object`. Если итератор возвращает IEnumerable<T> или IEnumerator<T>, необходимо выполнить неявное преобразование из типа выражения в операторе `yield return` в параметр универсального типа.

Нельзя включать инструкцию `yield return` или `yield break`:

- Лямбда-выражения и анонимные методы.
- Методы, содержащие небезопасные блоки. Дополнительные сведения см. в разделе unsafe.

Обработка исключений

Оператор `yield return` нельзя размещать в блоке try-catch. Оператор `yield return` можно размещать в блоке try оператора try-finally.

Оператор `yield break` можно размещать в блоке try или catch, но не в блоке finally.

Если тело `foreach` или `finally` (за пределами метода итератора) вызывает исключение, выполняется блок `await foreach` в методе итератора.

Техническая реализация

В следующем коде возвращается объект `IEnumerable<string>` из метода итератора и затем выполняется перебор его элементов.

```
IEnumerable<string> elements = MyIteratorMethod();
foreach (string element in elements)
{
   ...
}
```

При вызове `MyIteratorMethod` тело метода не выполняется. Вместо этого вызов возвращает `IEnumerable<string>` в переменную `elements`.

В итерации цикла `foreach` метод MoveNext вызывается для `elements`. Этот вызов выполняет тело `MyIteratorMethod` до достижения следующего оператора `yield return`. Выражение, возвращаемое оператором `yield return`, определяет не только значение переменной `element` для использования телом цикла, но и свойство Current объекта `elements`, представляющее собой `IEnumerable<string>`.

В каждой последующей итерации цикла `foreach` выполнение тела итератора продолжается с места остановки и при достижении оператора `yield return` оно снова останавливается. Цикл `foreach` завершается при достижении конца метода итератора или оператора `yield break`.

В следующем коде возвращается объект `IAsyncEnumerable<string>` из метода итератора и затем выполняется перебор его элементов.

```
IAsyncEnumerable<string> elements = MyAsyncIteratorMethod();
await foreach (string element in elements)
{
   // ...
}
```

В итерации цикла `await foreach` метод IAsyncEnumerator<T>.MoveNextAsync вызывается для `elements`. Возврат System.Threading.Tasks.ValueTask<TResult>, выполняемый `MoveNext`, завершается при достижении следующего оператора `yield return`.

В каждой последующей итерации цикла `await foreach` выполнение тела итератора продолжается с места остановки и при достижении оператора `yield return` оно снова останавливается. Цикл `await foreach` завершается при достижении конца метода итератора или оператора `yield break`.

Примеры

В следующем примере имеется оператор `yield return`, расположенный в цикле `for`. Каждая итерация тела оператора `foreach` в методе `Main` создает вызов функции итератора `Power`. При каждом вызове функции итератора происходит переход к следующему выполнению оператора `yield return`, которое осуществляется во время следующей итерации цикла `for`.

Возвращаемый тип метода итератора — IEnumerable (тип интерфейса итератора). При вызове метода итератора возвращается перечисляемый объект, содержащий степени числа.

```
public class PowersOf2
{
    static void Main()
    {
        // Display powers of 2 up to the exponent of 8:
        foreach (int i in Power(2, 8))
        {
            Console.Write("{0} ", i);
        }
    }

    public static System.Collections.Generic.IEnumerable<int> Power(int number, int exponent)
    {
        int result = 1;

        for (int i = 0; i < exponent; i++)
        {
            result = result * number;
            yield return result;
        }
    }

    // Output: 2 4 8 16 32 64 128 256
}
```

В следующем примере демонстрируется метод доступа `get`, представляющий собой итератор. В этом примере каждый оператор `yield return` возвращает экземпляр пользовательского класса.

```
public static class GalaxyClass
{
    public static void ShowGalaxies()
    {
        var theGalaxies = new Galaxies();
        foreach (Galaxy theGalaxy in theGalaxies.NextGalaxy)
        {
            Debug.WriteLine(theGalaxy.Name + " " + theGalaxy.MegaLightYears.ToString());
        }
    }

    public class Galaxies
    {

        public System.Collections.Generic.IEnumerable<Galaxy> NextGalaxy
        {
            get
            {
                yield return new Galaxy { Name = "Tadpole", MegaLightYears = 400 };
                yield return new Galaxy { Name = "Pinwheel", MegaLightYears = 25 };
                yield return new Galaxy { Name = "Milky Way", MegaLightYears = 0 };
                yield return new Galaxy { Name = "Andromeda", MegaLightYears = 3 };
            }
        }
    }

    public class Galaxy
    {
        public String Name { get; set; }
        public int MegaLightYears { get; set; }
    }
}
```

Спецификация языка C#

Дополнительные сведения см. в спецификации языка C#. Спецификация языка является предписывающим источником информации о синтаксисе и использовании языка C#.

См. также

- Справочник по C#
- Руководство по программированию на C#
- foreach, in
- Итераторы

Ключевые слова запроса (Справочник по C#)

18.12.2021 • 2 minutes to read

В этом разделе приводятся контекстные ключевые слова, используемые в выражениях запросов.

Содержание раздела

ПРЕДЛОЖЕНИЕ	ОПИСАНИЕ
from	Задает источник данных и переменную диапазона (аналогично переменной итерации).
where	Фильтрует элементы источника на основе одного или нескольких логических выражений, разделенных операторами логического И и ИЛИ (`&&` или `||`).
select	Задает тип и форму, которые будут иметь элементы в возвращаемой последовательности при выполнении запроса.
group	Группирует результаты запроса по заданному значению ключа.
into	Предоставляет идентификатор, который может использоваться в качестве ссылки на результаты предложения join, group или select.
orderby	Сортирует результаты запроса по возрастанию или убыванию на основе функции сравнения по умолчанию для соответствующего типа элемента.
join	Соединяет два источника данных посредством сравнения на равенство между двумя заданными критериями сопоставления.
let	Предоставляет переменную диапазона для хранения результатов выражения, вложенного в выражение запроса.
in	Контекстное ключевое слово в предложении join.
on	Контекстное ключевое слово в предложении join.
equals	Контекстное ключевое слово в предложении join.
by	Контекстное ключевое слово в предложении group.
ascending	Контекстное ключевое слово в предложении orderby.
descending	Контекстное ключевое слово в предложении orderby.

См. также

- Ключевые слова в C#
- Встроенный язык запросов LINQ
- LINQ в C#

Предложение from (справочник по C#)

18.12.2021 • 5 minutes to read

Выражение запроса должно начинаться с предложения `from`. Кроме того, выражение запроса может содержать вложенные запросы, которые также начинаются с предложения `from`. Предложение `from` определяет следующее:

- Источник данных, в отношении которого будет выполнен запрос или вложенный запрос.
- Локальная *переменная диапазона*, которая представляет каждый элемент в последовательности источника.

Переменная диапазона и источник данных строго типизированы. Источник данных, на который ссылается предложение `from`, должен иметь тип IEnumerable, IEnumerable<T> или производный от них тип, например IQueryable<T>.

В следующем примере `numbers` — это источник данных, а `num` — переменная диапазона. Обратите внимание, что обе переменные строго типизированы, даже несмотря на использование ключевого слова var.

```
class LowNums
{
    static void Main()
    {
        // A simple data source.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query.
        // lowNums is an IEnumerable<int>
        var lowNums = from num in numbers
            where num < 5
            select num;

        // Execute the query.
        foreach (int i in lowNums)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 4 1 3 2 0
```

Переменная диапазона

Компилятор выводит тип переменной диапазона, если источник данных реализует IEnumerable<T>. Например, если источник данных имеет тип `IEnumerable<Customer>`, выводится переменная диапазона `Customer`. Явно задавать тип необходимо только в том случае, если источник имеет не являющийся универсальным тип `IEnumerable`, например ArrayList. Дополнительные сведения см. в статье Практическое руководство. Выполнение запроса к ArrayList с помощью LINQ (C#).

В предыдущем примере `num` выводится к типу `int`. Поскольку переменная диапазона строго типизирована, вы можете использовать ее в вызовах методов и других операциях. Например, вместо кода `select num` можно использовать запись `select num.ToString()`, в результате чего выражение запроса будет возвращать последовательность строк вместо целых чисел. Также можно использовать запись

`select num + 10`, в результате чего выражение будет возвращать последовательность 14, 11, 13, 12, 10. Дополнительные сведения см. в разделе Предложение select.

Переменные диапазона схожи с переменными итерации в инструкции foreach, за исключением одного важного отличия: в переменной диапазона никогда не хранятся фактические данные из источника. Это лишь способ оптимизировать код, позволяющий запросу описывать действия, которые произойдут при выполнении запроса. Дополнительные сведения см. в разделе Введение в запросы LINQ (C#).

Составные предложения from

В некоторых случаях каждый элемент в последовательности источника может сам представлять собой последовательность или содержать ее. Например, в качестве источника данных может выступать `IEnumerable<Student>`, в котором каждый объект учащегося представляет собой последовательность, содержащую список полученных оценок. Для доступа к внутреннему списку каждого элемента `Student` можно использовать составные предложения `from`. Этот способ аналогичен использованию вложенных предложений foreach. Для фильтрации результатов можно добавить предложения where или orderby в любое из предложений `from`. В следующем примере показана последовательность объектов `Student`, каждый из которых содержит внутренний список `List` с целыми числами, соответствующими оценкам. Для доступа к внутреннему списку следует использовать предложение `from`. При необходимости вы можете вставлять предложения между двумя предложениями `from`.

```
class CompoundFrom
{
    // The element type of the data source.
    public class Student
    {
        public string LastName { get; set; }
        public List<int> Scores {get; set;}
    }

    static void Main()
    {

        // Use a collection initializer to create the data source. Note that
        // each element in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
           new Student {LastName="Omelchenko", Scores= new List<int> {97, 72, 81, 60}},
           new Student {LastName="O'Donnell", Scores= new List<int> {75, 84, 91, 39}},
           new Student {LastName="Mortensen", Scores= new List<int> {88, 94, 65, 85}},
           new Student {LastName="Garcia", Scores= new List<int> {97, 89, 85, 82}},
           new Student {LastName="Beebe", Scores= new List<int> {35, 72, 91, 70}}
        };

        // Use a compound from to access the inner sequence within each element.
        // Note the similarity to a nested foreach statement.
        var scoreQuery = from student in students
                         from score in student.Scores
                            where score > 90
                            select new { Last = student.LastName, score };

        // Execute the queries.
        Console.WriteLine("scoreQuery:");
        // Rest the mouse pointer on scoreQuery in the following line to
        // see its type. The type is IEnumerable<'a>, where 'a is an
        // anonymous type defined as new {string Last, int score}. That is,
        // each instance of this anonymous type has two members, a string
        // (Last) and an int (score).
        foreach (var student in scoreQuery)
        {
            Console.WriteLine("{0} Score: {1}", student.Last, student.score);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
scoreQuery:
Omelchenko Score: 97
O'Donnell Score: 91
Mortensen Score: 94
Garcia Score: 97
Beebe Score: 91
*/
```

Использование нескольких предложений from для выполнения соединений

Составное предложение `from` используется для доступа к внутренним коллекциям в одном источнике данных. Тем не менее запрос может содержать несколько предложений `from`, которые создают дополнительные запросы из независимых источников данных. Такой способ позволяет выполнять определенные типы операций соединения, которые нельзя выполнить с помощью предложения join.

В следующем примере показано, как можно выполнить полное перекрестное соединение двух источников данных с помощью двух предложений `from`.

```
class CompoundFrom2
{
    static void Main()
    {
        char[] upperCase = { 'A', 'B', 'C' };
        char[] lowerCase = { 'x', 'y', 'z' };

        // The type of joinQuery1 is IEnumerable<'a>, where 'a
        // indicates an anonymous type. This anonymous type has two
        // members, upper and lower, both of type char.
        var joinQuery1 =
            from upper in upperCase
            from lower in lowerCase
            select new { upper, lower };

        // The type of joinQuery2 is IEnumerable<'a>, where 'a
        // indicates an anonymous type. This anonymous type has two
        // members, upper and lower, both of type char.
        var joinQuery2 =
            from lower in lowerCase
            where lower != 'x'
            from upper in upperCase
            select new { lower, upper };

        // Execute the queries.
        Console.WriteLine("Cross join:");
        // Rest the mouse pointer on joinQuery1 to verify its type.
        foreach (var pair in joinQuery1)
        {
            Console.WriteLine("{0} is matched to {1}", pair.upper, pair.lower);
        }

        Console.WriteLine("Filtered non-equijoin:");
        // Rest the mouse pointer over joinQuery2 to verify its type.
        foreach (var pair in joinQuery2)
        {
            Console.WriteLine("{0} is matched to {1}", pair.lower, pair.upper);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
        Cross join:
        A is matched to x
        A is matched to y
        A is matched to z
        B is matched to x
        B is matched to y
        B is matched to z
        C is matched to x
        C is matched to y
        C is matched to z
        Filtered non-equijoin:
        y is matched to A
        y is matched to B
        y is matched to C
        z is matched to A
        z is matched to B
        z is matched to C
        */
```

Дополнительные сведения об операциях соединения, использующих несколько предложений `from`, см. в разделе Выполнение левых внешних соединений.

См. также раздел

- Ключевые слова запроса (LINQ)
- LINQ

Предложение where (Справочник по C#)

18.12.2021 • 2 minutes to read

Предложение `where` используется в выражении запроса для того, чтобы указать, какие элементы из источника данных будут возвращаться в выражении запроса. Оно применяет логическое условие (*предикат*) к каждому исходному элементу (на который ссылается переменная диапазона) и возвращает те из них, для которых указанное условие имеет значение true. Одно выражение запроса может содержать сразу несколько предложений `where`, а одно предложение — несколько частей выражения предиката.

Пример 1

В следующем примере предложение `where` отфильтровывает все номера, кроме тех, которые меньше пяти. Если удалить предложение `where`, возвращаются все номера из источника данных. Выражение `num < 5` является предикатом, который применяется к каждому элементу.

```
class WhereSample
{
    static void Main()
    {
        // Simple data source. Arrays support IEnumerable<T>.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Simple query with one predicate in where clause.
        var queryLowNums =
            from num in numbers
            where num < 5
            select num;

        // Execute the query.
        foreach (var s in queryLowNums)
        {
            Console.Write(s.ToString() + " ");
        }
    }
}
//Output: 4 1 3 2 0
```

Пример 2

В одном предложении `where` можно указать необходимое число предикатов, используя операторы && и ||. В следующем примере запрос определяет два предиката, позволяющие отобрать только четные номера меньше пяти.

```
class WhereSample2
{
    static void Main()
    {
        // Data source.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query with two predicates in where clause.
        var queryLowNums2 =
            from num in numbers
            where num < 5 && num % 2 == 0
            select num;

        // Execute the query
        foreach (var s in queryLowNums2)
        {
            Console.Write(s.ToString() + " ");
        }
        Console.WriteLine();

        // Create the query with two where clause.
        var queryLowNums3 =
            from num in numbers
            where num < 5
            where num % 2 == 0
            select num;

        // Execute the query
        foreach (var s in queryLowNums3)
        {
            Console.Write(s.ToString() + " ");
        }
    }
}
// Output:
// 4 2 0
// 4 2 0
```

Пример 3

Предложение `where` может содержать один или несколько методов, возвращающих логические значения. В следующем примере предложение `where` использует метод для того, чтобы определить, является ли текущее значение диапазона четным или нечетным.

```
class WhereSample3
{
    static void Main()
    {
        // Data source
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query with a method call in the where clause.
        // Note: This won't work in LINQ to SQL unless you have a
        // stored procedure that is mapped to a method by this name.
        var queryEvenNums =
            from num in numbers
            where IsEven(num)
            select num;

         // Execute the query.
        foreach (var s in queryEvenNums)
        {
            Console.Write(s.ToString() + " ");
        }
    }

    // Method may be instance method or static method.
    static bool IsEven(int i)
    {
        return i % 2 == 0;
    }
}
//Output: 4 8 6 2 0
```

Remarks

Предложение `where` представляет собой механизм фильтрации. Он может располагаться практически в любом месте выражения запроса, но не может быть первым или последним предложением. Предложение `where` может отображаться до или после предложения group в зависимости от того, необходимо ли отфильтровать исходные элементы до или после их объединения в группы.

Если указанный предикат недопустим для элементов в источнике данных, это вызовет ошибку компиляции. Это одно из преимуществ надежной проверки типов, предоставляемой LINQ.

Во время компиляции ключевое слово `where` преобразуется в вызов метода стандартного оператора запроса Where.

См. также раздел

- Ключевые слова запроса (LINQ)
- предложение from
- предложение select
- Фильтрация данных
- LINQ в C#
- LINQ

Предложение "select" (справочник по C#)

18.12.2021 • 5 minutes to read

В выражении запроса предложение `select` задает тип значений, которые будут получены при выполнении запроса. Получаемый результат зависит от вычисления всех предыдущих предложений и любых выражений в самом предложении `select`. Выражение запроса должно оканчиваться предложением `select` или group.

В следующем примере показано простое предложение `select` в выражении запроса.

```
class SelectSample1
{
    static void Main()
    {
        //Create the data source
        List<int> Scores = new List<int>() { 97, 92, 81, 60 };

        // Create the query.
        IEnumerable<int> queryHighScores =
            from score in Scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in queryHighScores)
        {
            Console.Write(i + " ");
        }
    }
}
//Output: 97 92 81
```

Тип последовательности, создаваемой предложением `select`, определяет тип переменной запроса `queryHighScores`. В самом простом случае предложение `select` просто задает переменную диапазона. В этом случае возвращаемая последовательность содержит элементы того же типа, что и источник данных. Дополнительные сведения см. в разделе Связи типов в операциях запроса LINQ. Тем не менее предложение `select` также реализует эффективный механизм для преобразования (или *проецирования*) данных источника в новые типы. Дополнительные сведения см. в разделе Преобразования данных с помощью LINQ (C#).

Пример

В следующем примере показаны различные формы, которые может принимать предложение `select`. В каждом запросе обратите внимание на связь между предложением `select` и типом *переменной запроса* (`studentQuery1`, `studentQuery2` и т. д.).

```
    class SelectSample2
    {
        // Define some classes
        public class Student
        {
            public string First { get; set; }
            public string Last { get; set; }
            public int ID { get; set; }
            public List<int> Scores;
```

```
            public ContactInfo GetContactInfo(SelectSample2 app, int id)
            {
                ContactInfo cInfo =
                    (from ci in app.contactList
                    where ci.ID == id
                    select ci)
                    .FirstOrDefault();

                return cInfo;
            }

            public override string ToString()
            {
                return First + " " + Last + ":" + ID;
            }
        }

        public class ContactInfo
        {
            public int ID { get; set; }
            public string Email { get; set; }
            public string Phone { get; set; }
            public override string ToString() { return Email + "," + Phone; }
        }

        public class ScoreInfo
        {
            public double Average { get; set; }
            public int ID { get; set; }
        }

        // The primary data source
        List<Student> students = new List<Student>()
        {
             new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int>() {97, 92, 81,
60}},
             new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int>() {75, 84, 91,
39}},
             new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int>() {88, 94, 65, 91}},
             new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int>() {97, 89, 85, 82}},
        };

        // Separate data source for contact info.
        List<ContactInfo> contactList = new List<ContactInfo>()
        {
            new ContactInfo {ID=111, Email="SvetlanO@Contoso.com", Phone="206-555-0108"},
            new ContactInfo {ID=112, Email="ClaireO@Contoso.com", Phone="206-555-0298"},
            new ContactInfo {ID=113, Email="SvenMort@Contoso.com", Phone="206-555-1130"},
            new ContactInfo {ID=114, Email="CesarGar@Contoso.com", Phone="206-555-0521"}
        };

        static void Main(string[] args)
        {
            SelectSample2 app = new SelectSample2();

            // Produce a filtered sequence of unmodified Students.
            IEnumerable<Student> studentQuery1 =
                from student in app.students
                where student.ID > 111
                select student;

            Console.WriteLine("Query1: select range_variable");
            foreach (Student s in studentQuery1)
            {
                Console.WriteLine(s.ToString());
            }

            // Produce a filtered sequence of elements that contain
            // only one property of each Student.
```

```
            IEnumerable<String> studentQuery2 =
                from student in app.students
                where student.ID > 111
                select student.Last;

            Console.WriteLine("\r\n studentQuery2: select range_variable.Property");
            foreach (string s in studentQuery2)
            {
                Console.WriteLine(s);
            }

            // Produce a filtered sequence of objects created by
            // a method call on each Student.
            IEnumerable<ContactInfo> studentQuery3 =
                from student in app.students
                where student.ID > 111
                select student.GetContactInfo(app, student.ID);

            Console.WriteLine("\r\n studentQuery3: select range_variable.Method");
            foreach (ContactInfo ci in studentQuery3)
            {
                Console.WriteLine(ci.ToString());
            }

            // Produce a filtered sequence of ints from
            // the internal array inside each Student.
            IEnumerable<int> studentQuery4 =
                from student in app.students
                where student.ID > 111
                select student.Scores[0];

            Console.WriteLine("\r\n studentQuery4: select range_variable[index]");
            foreach (int i in studentQuery4)
            {
                Console.WriteLine("First score = {0}", i);
            }

            // Produce a filtered sequence of doubles
            // that are the result of an expression.
            IEnumerable<double> studentQuery5 =
                from student in app.students
                where student.ID > 111
                select student.Scores[0] * 1.1;

            Console.WriteLine("\r\n studentQuery5: select expression");
            foreach (double d in studentQuery5)
            {
                Console.WriteLine("Adjusted first score = {0}", d);
            }

            // Produce a filtered sequence of doubles that are
            // the result of a method call.
            IEnumerable<double> studentQuery6 =
                from student in app.students
                where student.ID > 111
                select student.Scores.Average();

            Console.WriteLine("\r\n studentQuery6: select expression2");
            foreach (double d in studentQuery6)
            {
                Console.WriteLine("Average = {0}", d);
            }

            // Produce a filtered sequence of anonymous types
            // that contain only two properties from each Student.
            var studentQuery7 =
                from student in app.students
                where student.ID > 111
                select new { student.First, student.Last };
```

```
            Console.WriteLine("\r\n studentQuery7: select new anonymous type");
            foreach (var item in studentQuery7)
            {
                Console.WriteLine("{0}, {1}", item.Last, item.First);
            }

            // Produce a filtered sequence of named objects that contain
            // a method return value and a property from each Student.
            // Use named types if you need to pass the query variable
            // across a method boundary.
            IEnumerable<ScoreInfo> studentQuery8 =
                from student in app.students
                where student.ID > 111
                select new ScoreInfo
                {
                    Average = student.Scores.Average(),
                    ID = student.ID
                };

            Console.WriteLine("\r\n studentQuery8: select new named type");
            foreach (ScoreInfo si in studentQuery8)
            {
                Console.WriteLine("ID = {0}, Average = {1}", si.ID, si.Average);
            }

            // Produce a filtered sequence of students who appear on a contact list
            // and whose average is greater than 85.
            IEnumerable<ContactInfo> studentQuery9 =
                from student in app.students
                where student.Scores.Average() > 85
                join ci in app.contactList on student.ID equals ci.ID
                select ci;

            Console.WriteLine("\r\n studentQuery9: select result of join clause");
            foreach (ContactInfo ci in studentQuery9)
            {
                Console.WriteLine("ID = {0}, Email = {1}", ci.ID, ci.Email);
            }

            // Keep the console window open in debug mode
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
            }
    }
/* Output
    Query1: select range_variable
    Claire O'Donnell:112
    Sven Mortensen:113
    Cesar Garcia:114

    studentQuery2: select range_variable.Property
    O'Donnell
    Mortensen
    Garcia

    studentQuery3: select range_variable.Method
    ClaireO@Contoso.com,206-555-0298
    SvenMort@Contoso.com,206-555-1130
    CesarGar@Contoso.com,206-555-0521

    studentQuery4: select range_variable[index]
    First score = 75
    First score = 88
    First score = 97

    studentQuery5: select expression
    Adjusted first score = 82.5
    Adjusted first score = 96.8
```

```
        Adjusted first score = 106.7

        studentQuery6: select expression2
        Average = 72.25
        Average = 84.5
        Average = 88.25

        studentQuery7: select new anonymous type
        O'Donnell, Claire
        Mortensen, Sven
        Garcia, Cesar

        studentQuery8: select new named type
        ID = 112, Average = 72.25
        ID = 113, Average = 84.5
        ID = 114, Average = 88.25

        studentQuery9: select result of join clause
        ID = 114, Email = CesarGar@Contoso.com
*/
```

Как показано в `studentQuery8` в предыдущем примере, в некоторых случаях требуется, чтобы элементы возвращаемой последовательности содержали только подмножество свойств элементов источника. Максимально уменьшая размер возвращаемой последовательности, вы можете снизить требования к памяти и, соответственно, повысить скорость выполнения запроса. Это можно сделать, создав анонимный тип в предложении `select` и используя инициализатор объекта для его инициализации с соответствующими свойствами из элементов источника. Пример того, как это сделать, см. в разделе Инициализаторы объектов и коллекций.

Remarks

Во время компиляции предложение `select` преобразуется в вызов метода Select стандартного оператора запроса.

См. также

- Справочник по C#
- Ключевые слова запроса (LINQ)
- предложение from
- partial (метод) (справочник по C#)
- Анонимные типы
- LINQ в C#
- LINQ

Предложение group (Справочник по C#)

18.12.2021 • 7 minutes to read

Предложение `group` возвращает последовательность объектов IGrouping<TKey,TElement>, содержащих нуль или более элементов, соответствующих значению ключа группы. Например, можно сгруппировать последовательность строк в соответствии с первой буквой в каждой строке. В этом случае первая буква является ключом и имеет тип char. Она хранится в свойстве `Key` каждого объекта IGrouping<TKey,TElement>. Тип ключа определяется компилятором.

Завершить выражение запроса с предложением `group` можно следующим образом:

```
// Query variable is an IEnumerable<IGrouping<char, Student>>
var studentQuery1 =
    from student in students
    group student by student.Last[0];
```

Чтобы выполнить дополнительные запросы для каждой из групп, можно указать временный идентификатор, воспользовавшись для этого контекстным ключевым словом into. При использовании ключевого слова `into` необходимо продолжить запрос и завершить его инструкцией `select` или другим предложением `group`, как показано в следующем фрагменте:

```
// Group students by the first letter of their last name
// Query variable is an IEnumerable<IGrouping<char, Student>>
var studentQuery2 =
    from student in students
    group student by student.Last[0] into g
    orderby g.Key
    select g;
```

Более подробные примеры использования предложения `group` с ключевым словом `into` и без него см. в разделе "Пример" в этой статье.

Перечисление результатов запроса group

Так как возвращаемые запросом `group` объекты IGrouping<TKey,TElement> представляют собой список списков, для доступа к каждому из элементов этих групп необходимо использовать вложенный цикл foreach. Во внешнем цикле итерация будет выполняться по ключам групп, а во внутреннем цикле — по элементам самих групп. У группы может быть ключ, но не быть элементов. Ниже приведен пример цикла `foreach`, который выполняет запрос, показанный в предыдущих примерах кода:

```
// Iterate group items with a nested foreach. This IGrouping encapsulates
// a sequence of Student objects, and a Key of type char.
// For convenience, var can also be used in the foreach statement.
foreach (IGrouping<char, Student> studentGroup in studentQuery2)
{
     Console.WriteLine(studentGroup.Key);
     // Explicit type for student could also be used here.
     foreach (var student in studentGroup)
     {
         Console.WriteLine("   {0}, {1}", student.Last, student.First);
     }
 }
```

Типы ключей

Ключи групп могут быть любого типа, например строкового, встроенного числового, пользовательского именованного или анонимного типа.

Группировка по строке

В предыдущих примерах кода использовался ключ типа `char`. Вместо него можно было указать строку, например фамилию:

```
// Same as previous example except we use the entire last name as a key.
// Query variable is an IEnumerable<IGrouping<string, Student>>
var studentQuery3 =
    from student in students
    group student by student.Last;
```

Группировка по логическому значению

В следующем примере показано разбиение результатов на две группы в зависимости от значения логического ключа. Обратите внимание, что значение формируется с помощью входящего в состав предложения `group` выражения.

```
class GroupSample1
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
        public List<int> Scores;
    }

    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that each element
        //  in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
           new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int> {97, 72, 81,
60}},
           new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int> {75, 84, 91, 39}},
           new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int> {99, 89, 91, 95}},
           new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int> {72, 81, 65, 84}},
           new Student {First="Debra", Last="Garcia", ID=115, Scores= new List<int> {97, 89, 85, 82}}
        };

        return students;
    }

    static void Main()
    {
        // Obtain the data source.
        List<Student> students = GetStudents();

        // Group by true or false.
        // Query variable is an IEnumerable<IGrouping<bool, Student>>
        var booleanGroupQuery =
            from student in students
            group student by student.Scores.Average() >= 80; //pass or fail!

        // Execute the query and access items in each group
        foreach (var studentGroup in booleanGroupQuery)
        {
            Console.WriteLine(studentGroup.Key == true ? "High averages" : "Low averages");
            foreach (var student in studentGroup)
            {
                Console.WriteLine("   {0}, {1}:{2}", student.Last, student.First, student.Scores.Average());
            }
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
  Low averages
   Omelchenko, Svetlana:77.5
   O'Donnell, Claire:72.25
   Garcia, Cesar:75.5
  High averages
   Mortensen, Sven:93.5
   Garcia, Debra:88.25
*/
```

Группировка по числовому диапазону

В следующем примере с помощью выражения создаются числовые ключи групп, обозначающие

диапазоны значений в выборке. Обратите внимание на удобное использование let для хранения результатов вызова метода, чтобы в предложении `group` не приходилось вызывать метод дважды. См. сведения о безопасном использовании методов в выражениях запросов в руководстве по обработке исключений в выражениях запросов.

```
class GroupSample2
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
        public List<int> Scores;
    }

    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that each element
        //  in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
           new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int> {97, 72, 81,
60}},
           new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int> {75, 84, 91, 39}},
           new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int> {99, 89, 91, 95}},
           new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int> {72, 81, 65, 84}},
           new Student {First="Debra", Last="Garcia", ID=115, Scores= new List<int> {97, 89, 85, 82}}
        };

        return students;
    }

    // This method groups students into percentile ranges based on their
    // grade average. The Average method returns a double, so to produce a whole
    // number it is necessary to cast to int before dividing by 10.
    static void Main()
    {
        // Obtain the data source.
        List<Student> students = GetStudents();

        // Write the query.
        var studentQuery =
            from student in students
            let avg = (int)student.Scores.Average()
            group student by (avg / 10) into g
            orderby g.Key
            select g;

        // Execute the query.
        foreach (var studentGroup in studentQuery)
        {
            int temp = studentGroup.Key * 10;
            Console.WriteLine("Students with an average between {0} and {1}", temp, temp + 10);
            foreach (var student in studentGroup)
            {
                Console.WriteLine("   {0}, {1}:{2}", student.Last, student.First, student.Scores.Average());
            }
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
     Students with an average between 70 and 80
```

```
      Omelchenko, Svetlana:77.5
      O'Donnell, Claire:72.25
      Garcia, Cesar:75.5
    Students with an average between 80 and 90
      Garcia, Debra:88.25
    Students with an average between 90 and 100
      Mortensen, Sven:93.5
*/
```

Группировка по составному ключу

Составные ключи используются в тех случаях, когда необходимо сгруппировать элементы по более чем одному ключу. Составной ключ создается с помощью анонимного типа или именованного типа для хранения элементов ключа. В следующем примере предполагается, что для класса `Person` были объявлены члены `surname` и `city`. Предложение `group` позволяет создать отдельную группу для каждого набора людей, имеющих одинаковые фамилии и города.

```
group person by new {name = person.surname, city = person.city};
```

Если требуется передать переменную запроса другому методу, следует использовать именованные типы. Создайте особый класс, используя автоматически реализуемые свойства для ключей, а затем переопределите методы Equals и GetHashCode. Можно также воспользоваться структурой, чтобы избежать необходимости переопределять эти методы. См. сведения в руководствах по реализации облегченного класса с автоматически реализуемыми свойствами и получению повторяющихся файлов в дереве каталогов. В последней статье имеется пример кода, демонстрирующий использование составных ключей с именованным типом.

Пример 1

В следующем примере показан стандартный шаблон для разбиения данных на группы без применения к группам дополнительной логики запроса. Такой прием называется группировкой без продолжения. Элементы массива строк группируются в соответствии со своими первыми буквами. Результатом запроса является тип IGrouping<TKey,TElement>, который содержит открытое свойство `Key` типа `char` и коллекцию IEnumerable<T>, содержащую каждый элемент в группе.

Результатом выполнения предложения `group` является последовательность из последовательностей. Таким образом, для доступа к отдельным элементам каждой из полученных групп следует использовать вложенный цикл `foreach`, в котором итерация выполняется по ключам групп, как показано в следующем примере.

```
class GroupExample1
{
    static void Main()
    {
        // Create a data source.
        string[] words = { "blueberry", "chimpanzee", "abacus", "banana", "apple", "cheese" };

        // Create the query.
        var wordGroups =
            from w in words
            group w by w[0];

        // Execute the query.
        foreach (var wordGroup in wordGroups)
        {
            Console.WriteLine("Words that start with the letter '{0}':", wordGroup.Key);
            foreach (var word in wordGroup)
            {
                Console.WriteLine(word);
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
      Words that start with the letter 'b':
        blueberry
        banana
      Words that start with the letter 'c':
        chimpanzee
        cheese
      Words that start with the letter 'a':
        abacus
        apple
     */
```

Пример 2

В этом примере показано выполнение дополнительных операций над группами после их создания. Для этого используется *continuation* с `into`. Дополнительные сведения см. в разделе into. В следующем примере создается запрос, выбирающий только те группы, элементы которых начинаются на гласную.

```
class GroupClauseExample2
{
    static void Main()
    {
        // Create the data source.
        string[] words2 = { "blueberry", "chimpanzee", "abacus", "banana", "apple", "cheese", "elephant",
"umbrella", "anteater" };

        // Create the query.
        var wordGroups2 =
            from w in words2
            group w by w[0] into grps
            where (grps.Key == 'a' || grps.Key == 'e' || grps.Key == 'i'
                   || grps.Key == 'o' || grps.Key == 'u')
            select grps;

        // Execute the query.
        foreach (var wordGroup in wordGroups2)
        {
            Console.WriteLine("Groups that start with a vowel: {0}", wordGroup.Key);
            foreach (var word in wordGroup)
            {
                Console.WriteLine("   {0}", word);
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Groups that start with a vowel: a
        abacus
        apple
        anteater
    Groups that start with a vowel: e
        elephant
    Groups that start with a vowel: u
        umbrella
*/
```

Комментарии

Во время компиляции предложения `group` преобразуются в вызовы метода GroupBy.

См. также

- IGrouping<TKey,TElement>
- GroupBy
- ThenBy
- ThenByDescending
- Ключевые слова запроса
- LINQ
- Создание вложенной группы
- Группировка результатов запросов
- Вложенный запрос в операции группирования

into (Справочник по C#)

18.12.2021 • 2 minutes to read

Контекстное слово `into` можно использовать для создания временного идентификатора для сохранения результатов предложений group, join или select в новый идентификатор. Этот идентификатор может сам по себе стать источником дополнительных команд запроса. При использовании в предложениях `group` или `select` применение нового идентификатора может называться *продолжением*.

Пример

В следующем примере показано использование ключевого слова `into` для создания временного идентификатора `fruitGroup`, который имеет выведенный тип `IGrouping`. С помощью идентификатора можно вызвать метод Count для каждой группы и выбрать только те группы, которые содержат несколько слов.

```
class IntoSample1
{
    static void Main()
    {

        // Create a data source.
        string[] words = { "apples", "blueberries", "oranges", "bananas", "apricots"};

        // Create the query.
        var wordGroups1 =
            from w in words
            group w by w[0] into fruitGroup
            where fruitGroup.Count() >= 2
            select new { FirstLetter = fruitGroup.Key, Words = fruitGroup.Count() };

        // Execute the query. Note that we only iterate over the groups,
        // not the items in each group
        foreach (var item in wordGroups1)
        {
            Console.WriteLine(" {0} has {1} elements.", item.FirstLetter, item.Words);
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
   a has 2 elements.
   b has 2 elements.
*/
```

Использование ключевого слова `into` в предложении `group` необходимо только в тех случаях, когда требуется выполнить дополнительные операции запроса для каждой группы. Дополнительные сведения см. в разделе Предложение group.

Пример использования ключевого слова `into` в предложении `join` см. в разделе Предложение join.

См. также раздел

- Ключевые слова запроса (LINQ)

- LINQ в C#
- предложение group

Предложение orderby (Справочник по C#)

18.12.2021 • 2 minutes to read

В выражении запроса предложение `orderby` задает сортировку возвращаемой последовательности или вложенной последовательности (группы) в порядке возрастания или убывания. Для выполнения одной или нескольких операций последующей сортировки можно указать несколько ключей. Сортировка выполняется с помощью функции сравнения по умолчанию для соответствующего типа элемента. По умолчанию используется порядок сортировки по возрастанию. Также можно указать настраиваемую функцию сравнения. Тем не менее сделать это можно только с использованием синтаксиса на основе метода. Дополнительные сведения см. в разделе Сортировка данных.

Пример 1

В следующем примере первый запрос сортирует слова в алфавитном порядке, начиная с A. Второй запрос сортирует эти же слова в порядке убывания. (Ключевое слово `ascending` определяет порядок сортировки по умолчанию и может быть опущено.)

```
class OrderbySample1
{
    static void Main()
    {
        // Create a delicious data source.
        string[] fruits = { "cherry", "apple", "blueberry" };

        // Query for ascending sort.
        IEnumerable<string> sortAscendingQuery =
            from fruit in fruits
            orderby fruit //"ascending" is default
            select fruit;

        // Query for descending sort.
        IEnumerable<string> sortDescendingQuery =
            from w in fruits
            orderby w descending
            select w;

        // Execute the query.
        Console.WriteLine("Ascending:");
        foreach (string s in sortAscendingQuery)
        {
            Console.WriteLine(s);
        }

        // Execute the query.
        Console.WriteLine(Environment.NewLine + "Descending:");
        foreach (string s in sortDescendingQuery)
        {
            Console.WriteLine(s);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
Ascending:
apple
blueberry
cherry

Descending:
cherry
blueberry
apple
*/
```

Пример 2

В следующем примере выполняется первичная сортировка списка студентов по фамилиям, а затем дополнительная сортировка по именам.

```
class OrderbySample2
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
    }
```

```
    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that each element
        //  in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
           new Student {First="Svetlana", Last="Omelchenko", ID=111},
           new Student {First="Claire", Last="O'Donnell", ID=112},
           new Student {First="Sven", Last="Mortensen", ID=113},
           new Student {First="Cesar", Last="Garcia", ID=114},
           new Student {First="Debra", Last="Garcia", ID=115}
        };

        return students;
    }
    static void Main(string[] args)
    {
        // Create the data source.
        List<Student> students = GetStudents();

        // Create the query.
        IEnumerable<Student> sortedStudents =
            from student in students
            orderby student.Last ascending, student.First ascending
            select student;

        // Execute the query.
        Console.WriteLine("sortedStudents:");
        foreach (Student student in sortedStudents)
            Console.WriteLine(student.Last + " " + student.First);

        // Now create groups and sort the groups. The query first sorts the names
        // of all students so that they will be in alphabetical order after they are
        // grouped. The second orderby sorts the group keys in alpha order.
        var sortedGroups =
            from student in students
            orderby student.Last, student.First
            group student by student.Last[0] into newGroup
            orderby newGroup.Key
            select newGroup;

        // Execute the query.
        Console.WriteLine(Environment.NewLine + "sortedGroups:");
        foreach (var studentGroup in sortedGroups)
        {
            Console.WriteLine(studentGroup.Key);
            foreach (var student in studentGroup)
            {
                Console.WriteLine("   {0}, {1}", student.Last, student.First);
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
sortedStudents:
Garcia Cesar
Garcia Debra
Mortensen Sven
O'Donnell Claire
Omelchenko Svetlana

sortedGroups:
G
   Garcia, Cesar
   Garcia, Debra
```

```
   Garcia, Debra
M
   Mortensen, Sven
O
   O'Donnell, Claire
   Omelchenko, Svetlana
*/
```

Комментарии

Во время компиляции предложение `orderby` преобразуется в вызов метода OrderBy. Если в предложении `orderby` используется несколько ключей, они преобразуются в вызовы метода ThenBy.

См. также

- Справочник по C#
- Ключевые слова запроса (LINQ)
- LINQ в C#
- предложение group
- LINQ

Предложение join (Справочник по C#)

18.12.2021 • 9 minutes to read

Предложение `join` удобно для связывания элементов из разных исходных последовательностей, не имеющих прямых связей в объектной модели. Единственное требование заключается в том, что у элементов в каждом источнике должно быть общим некоторое значение, которое может быть проверено на равенство. Например, у дистрибьютора может быть список поставщиков определенного продукта и список покупателей. Предложение `join` может использоваться, например, для создания списка поставщиков и покупателей этого продукта, которые находятся в одном заданном регионе.

Предложение `join` принимает две исходные последовательности в качестве входных данных. Элементы в каждой последовательности должны являться свойством (или содержать свойство), которое можно сравнить с соответствующим свойством в другой последовательности. Предложение `join` сравнивает указанные ключи на равенство с помощью специального ключевого слова `equals`. Все соединения, выполняемые предложением `join`, являются эквисоединениями. Форма выходных данных предложения `join` зависит от конкретного типа выполняемого соединения. Ниже приведены три наиболее распространенных типа соединения:

- внутреннее соединение,
- групповое соединение,
- левое внешнее соединение.

внутреннее соединение,

В следующем примере показано простое внутреннее эквисоединение. По этому запросу создается прямая последовательность пар "название продукта/категория". Одна и та же строка, обозначающая категорию, будет присутствовать в нескольких элементах. Если для элемента из `categories` нет соответствующего элемента в `products`, эта категория не будет отображаться в результатах.

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name }; //produces flat sequence
```

Дополнительные сведения см. в разделе Выполнение внутренних соединений.

групповое соединение,

Предложение `join` с выражением `into` называется групповым соединением.

```
var innerGroupJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into prodGroup
    select new { CategoryName = category.Name, Products = prodGroup };
```

Групповое соединение создает иерархическую последовательность результатов, которая связывает элементы в левой исходной последовательности с одним или несколькими соответствующими элементами в правой исходной последовательности. Групповое соединение не имеет эквивалента в терминах реляционной базы данных. По сути это последовательность массивов объектов.

Если для элементов в левом источнике не удается найти элементы из правой исходной последовательности, предложение `join` создаст пустой массив для этого элемента. Таким образом, групповое соединение — это такое же внутреннее эквисоединение, за исключением того что последовательность результатов организуется в группы.

Если просто выбрать результаты группового соединения, можно получить доступ к элементам, но не определить ключ, по которому они совпадают. Таким образом, удобнее будет выбирать результаты группового соединения в новый тип, который также имеет имя ключа, как показано в предыдущем примере.

Кроме того, результаты группового соединения можно использовать как генераторы других вложенных запросов:

```
var innerGroupJoinQuery2 =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into prodGroup
    from prod2 in prodGroup
    where prod2.UnitPrice > 2.50M
    select prod2;
```

Дополнительные сведения см. в разделе Выполнение групповых соединений.

левое внешнее соединение.

В левом внешнем соединении возвращаются все элементы в левой исходной последовательности, даже если в правой последовательности нет соответствующих элементов. Для выполнения левого внешнего соединения в LINQ используйте метод `DefaultIfEmpty` в сочетании с групповым соединением, чтобы указать правый элемент по умолчанию, создаваемый, если левый элемент не имеет совпадений. `null` можно использовать как значение по умолчанию для любого ссылочного типа, кроме того, можно указать определенный пользователем тип по умолчанию. В следующем примере показан определяемый пользователем тип по умолчанию:

```
var leftOuterJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into prodGroup
    from item in prodGroup.DefaultIfEmpty(new Product { Name = String.Empty, CategoryID = 0 })
    select new { CatName = category.Name, ProdName = item.Name };
```

Дополнительные сведения см. в разделе Выполнение левых внешних соединений.

Оператор равенства

Предложение `join` выполняет эквисоединение. Другими словами совпадения могут основываться только на равенстве двух ключей. Другие типы сравнений, такие как "больше чем" или "не равно" не поддерживаются. Чтобы гарантировать, что все соединения являются эквисоединениями, предложение `join` использует ключевое слово `equals` оператора `==`. Ключевое слово `equals` может использоваться только в предложении `join`, и оно имеет важные отличия от оператора `==`. При сравнении строк `equals` имеет перегрузку для сравнения по значению, а оператор `==` использует равенство ссылок. Если обе стороны сравнения имеют одинаковые строковые переменные, `equals` и `==` получат одинаковый результат — true. Это связано с тем, что когда программа объявляет две или более эквивалентные строковые переменные, компилятор хранит их в одном расположении. Дополнительные сведения, см. в разделе Равенство ссылок и интернирование строк. Другим важным отличием является сравнение со значением NULL: `null equals null` вычисляется как false с оператором `equals`, а оператор `==` вычисляет его как true. Наконец, отличается поведения включения в область: с `equals` левый ключ использует

внешнюю исходную последовательность, а правый ключ — внутренний источник. Внешний источник попадает в область только левой части `equals`, а внутренняя исходная последовательность — только в область правой части.

Соединения, не являющиеся уравнивающими

Соединения, не являющиеся эквисоединениями, перекрестные соединения и другие настраиваемые операции соединения можно выполнять с помощью нескольких предложений `from` для независимого ввода новых последовательностей в запрос. Дополнительные сведения см. в разделе Выполнение пользовательских операций соединения.

Соединения коллекций объектов и реляционных таблиц

В выражении запроса LINQ операции соединения выполняются для коллекций объектов. Коллекции объектов нельзя "соединять" точно так же, как две реляционные таблицы. В LINQ явные предложения `join` требуются, только если две исходные последовательности не связаны каким-либо отношением. При работе с LINQ to SQL таблицы внешних ключей представлены в объектной модели как свойства основной таблицы. Например, в базе данных Northwind таблица Customer имеет связь типа "внешний ключ" с таблицей Orders. При сопоставлении таблиц с объектной моделью класс Customer имеет свойство Orders, содержащий коллекцию заказов, связанных с клиентом. Фактически соединение уже выполнено автоматически.

Дополнительные сведения о создании запросов по связанным таблицам в контексте LINQ to SQL см. в разделе Практическое руководство. Сопоставление связей баз данных.

Составные ключи

Равенство нескольких значений можно проверить с помощью составного ключа. Дополнительные сведения см. в разделе Соединение с помощью составных ключей. Составные ключи можно также использовать в предложении `group`.

Пример

В следующем примере сравниваются результаты внутреннего соединения, группового соединения и левого внешнего соединения для одних и тех же исходных данных с использованием одинаковых совпадающих ключей. В эти примеры для уточнения результатов в окне консоли добавлен дополнительный код.

```
class JoinDemonstration
{
    #region Data

    class Product
    {
        public string Name { get; set; }
        public int CategoryID { get; set; }
    }

    class Category
    {
        public string Name { get; set; }
        public int ID { get; set; }
    }

    // Specify the first data source.
    List<Category> categories = new List<Category>()
    {
        new Category {Name="Beverages", ID=001},
```

```
        new Category {Name="Condiments", ID=002},
        new Category {Name="Vegetables", ID=003},
        new Category {Name="Grains", ID=004},
        new Category {Name="Fruit", ID=005}
    };

    // Specify the second data source.
    List<Product> products = new List<Product>()
   {
      new Product {Name="Cola",  CategoryID=001},
      new Product {Name="Tea",  CategoryID=001},
      new Product {Name="Mustard", CategoryID=002},
      new Product {Name="Pickles", CategoryID=002},
      new Product {Name="Carrots", CategoryID=003},
      new Product {Name="Bok Choy", CategoryID=003},
      new Product {Name="Peaches", CategoryID=005},
      new Product {Name="Melons", CategoryID=005},
    };
    #endregion

    static void Main(string[] args)
    {
        JoinDemonstration app = new JoinDemonstration();

        app.InnerJoin();
        app.GroupJoin();
        app.GroupInnerJoin();
        app.GroupJoin3();
        app.LeftOuterJoin();
        app.LeftOuterJoin2();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    void InnerJoin()
    {
        // Create the query that selects
        // a property from each element.
        var innerJoinQuery =
           from category in categories
           join prod in products on category.ID equals prod.CategoryID
           select new { Category = category.ID, Product = prod.Name };

        Console.WriteLine("InnerJoin:");
        // Execute the query. Access results
        // with a simple foreach statement.
        foreach (var item in innerJoinQuery)
        {
            Console.WriteLine("{0,-10}{1}", item.Product, item.Category);
        }
        Console.WriteLine("InnerJoin: {0} items in 1 group.", innerJoinQuery.Count());
        Console.WriteLine(System.Environment.NewLine);
    }

    void GroupJoin()
    {
        // This is a demonstration query to show the output
        // of a "raw" group join. A more typical group join
        // is shown in the GroupInnerJoin method.
        var groupJoinQuery =
           from category in categories
           join prod in products on category.ID equals prod.CategoryID into prodGroup
           select prodGroup;

        // Store the count of total items (for demonstration only).
        int totalItems = 0;
```

```
        Console.WriteLine("Simple GroupJoin:");

        // A nested foreach statement is required to access group items.
        foreach (var prodGrouping in groupJoinQuery)
        {
            Console.WriteLine("Group:");
            foreach (var item in prodGrouping)
            {
                totalItems++;
                Console.WriteLine("   {0,-10}{1}", item.Name, item.CategoryID);
            }
        }
        Console.WriteLine("Unshaped GroupJoin: {0} items in {1} unnamed groups", totalItems,
groupJoinQuery.Count());
        Console.WriteLine(System.Environment.NewLine);
    }

    void GroupInnerJoin()
    {
        var groupJoinQuery2 =
            from category in categories
            orderby category.ID
            join prod in products on category.ID equals prod.CategoryID into prodGroup
            select new
            {
                Category = category.Name,
                Products = from prod2 in prodGroup
                           orderby prod2.Name
                           select prod2
            };

        //Console.WriteLine("GroupInnerJoin:");
        int totalItems = 0;

        Console.WriteLine("GroupInnerJoin:");
        foreach (var productGroup in groupJoinQuery2)
        {
            Console.WriteLine(productGroup.Category);
            foreach (var prodItem in productGroup.Products)
            {
                totalItems++;
                Console.WriteLine("  {0,-10} {1}", prodItem.Name, prodItem.CategoryID);
            }
        }
        Console.WriteLine("GroupInnerJoin: {0} items in {1} named groups", totalItems,
groupJoinQuery2.Count());
        Console.WriteLine(System.Environment.NewLine);
    }

    void GroupJoin3()
    {

        var groupJoinQuery3 =
            from category in categories
            join product in products on category.ID equals product.CategoryID into prodGroup
            from prod in prodGroup
            orderby prod.CategoryID
            select new { Category = prod.CategoryID, ProductName = prod.Name };

        //Console.WriteLine("GroupInnerJoin:");
        int totalItems = 0;

        Console.WriteLine("GroupJoin3:");
        foreach (var item in groupJoinQuery3)
        {
            totalItems++;
            Console.WriteLine("   {0}:{1}", item.ProductName, item.Category);
        }
```

```
        Console.WriteLine("GroupJoin3: {0} items in 1 group", totalItems);
        Console.WriteLine(System.Environment.NewLine);
    }

    void LeftOuterJoin()
    {
        // Create the query.
        var leftOuterQuery =
           from category in categories
           join prod in products on category.ID equals prod.CategoryID into prodGroup
           select prodGroup.DefaultIfEmpty(new Product() { Name = "Nothing!", CategoryID = category.ID });

        // Store the count of total items (for demonstration only).
        int totalItems = 0;

        Console.WriteLine("Left Outer Join:");

        // A nested foreach statement  is required to access group items
        foreach (var prodGrouping in leftOuterQuery)
        {
            Console.WriteLine("Group:");
            foreach (var item in prodGrouping)
            {
                totalItems++;
                Console.WriteLine("  {0,-10}{1}", item.Name, item.CategoryID);
            }
        }
        Console.WriteLine("LeftOuterJoin: {0} items in {1} groups", totalItems, leftOuterQuery.Count());
        Console.WriteLine(System.Environment.NewLine);
    }

    void LeftOuterJoin2()
    {
        // Create the query.
        var leftOuterQuery2 =
           from category in categories
           join prod in products on category.ID equals prod.CategoryID into prodGroup
           from item in prodGroup.DefaultIfEmpty()
           select new { Name = item == null ? "Nothing!" : item.Name, CategoryID = category.ID };

        Console.WriteLine("LeftOuterJoin2: {0} items in 1 group", leftOuterQuery2.Count());
        // Store the count of total items
        int totalItems = 0;

        Console.WriteLine("Left Outer Join 2:");

        // Groups have been flattened.
        foreach (var item in leftOuterQuery2)
        {
            totalItems++;
            Console.WriteLine("{0,-10}{1}", item.Name, item.CategoryID);
        }
        Console.WriteLine("LeftOuterJoin2: {0} items in 1 group", totalItems);
    }
}
/*Output:

InnerJoin:
Cola      1
Tea       1
Mustard   2
Pickles   2
Carrots   3
Bok Choy  3
Peaches   5
Melons    5
InnerJoin: 8 items in 1 group.
```

```
Unshaped GroupJoin:
Group:
    Cola      1
    Tea       1
Group:
    Mustard   2
    Pickles   2
Group:
    Carrots   3
    Bok Choy  3
Group:
Group:
    Peaches   5
    Melons    5
Unshaped GroupJoin: 8 items in 5 unnamed groups


GroupInnerJoin:
Beverages
    Cola       1
    Tea        1
Condiments
    Mustard    2
    Pickles    2
Vegetables
    Bok Choy   3
    Carrots    3
Grains
Fruit
    Melons     5
    Peaches    5
GroupInnerJoin: 8 items in 5 named groups


GroupJoin3:
    Cola:1
    Tea:1
    Mustard:2
    Pickles:2
    Carrots:3
    Bok Choy:3
    Peaches:5
    Melons:5
GroupJoin3: 8 items in 1 group


Left Outer Join:
Group:
    Cola      1
    Tea       1
Group:
    Mustard   2
    Pickles   2
Group:
    Carrots   3
    Bok Choy  3
Group:
    Nothing!  4
Group:
    Peaches   5
    Melons    5
LeftOuterJoin: 9 items in 5 groups


LeftOuterJoin2: 9 items in 1 group
Left Outer Join 2:
Cola      1
Tea       1
```

```
Mustard   2
Pickles   2
Carrots   3
Bok Choy  3
Nothing!  4
Peaches   5
Melons    5
LeftOuterJoin2: 9 items in 1 group
Press any key to exit.
*/
```

Комментарии

Предложение `join`, за которым не следует `into`, преобразуется в вызов метода Join. Предложение `join`, за которым следует `into`, преобразуется в вызов метода GroupJoin.

См. также раздел

- Ключевые слова запроса (LINQ)
- LINQ
- Операции соединения
- предложение group
- Выполнение левых внешних соединений
- Выполнение внутренних соединений
- Выполнение групповых соединений
- Упорядочение результатов предложения соединения
- Соединение с помощью составных ключей
- Совместимые системы баз данных для Visual Studio

Предложение let (справочник по C#)

18.12.2021 • 2 minutes to read

В выражении запроса иногда требуется сохранить результат вложенного выражения, который будет использоваться в последующих предложениях. Это можно сделать с помощью ключевого слова `let`, которое создает новую переменную диапазона и инициализирует ее, используя результат предоставленного выражения. После инициализации с использованием этого значения такую переменную диапазона нельзя использовать для хранения других значений. Тем не менее если переменная диапазона хранит запрашиваемый тип, к ней можно выполнять запросы.

Пример

В следующем примере показываются два способа использования `let`:

1. Создание перечисляемого типа, к которому можно выполнять запросы.
2. Реализация запроса с однократным вызовом `ToLower` для переменной диапазона `word`. Если ключевое слово `let` не используется, потребуется выполнять вызов `ToLower` в каждом предикате предложения `where`.

```
class LetSample1
{
    static void Main()
    {
        string[] strings =
        {
            "A penny saved is a penny earned.",
            "The early bird catches the worm.",
            "The pen is mightier than the sword."
        };

        // Split the sentence into an array of words
        // and select those whose first letter is a vowel.
        var earlyBirdQuery =
            from sentence in strings
            let words = sentence.Split(' ')
            from word in words
            let w = word.ToLower()
            where w[0] == 'a' || w[0] == 'e'
                || w[0] == 'i' || w[0] == 'o'
                || w[0] == 'u'
            select word;

        // Execute the query.
        foreach (var v in earlyBirdQuery)
        {
            Console.WriteLine("\"{0}\" starts with a vowel", v);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    "A" starts with a vowel
    "is" starts with a vowel
    "a" starts with a vowel
    "earned." starts with a vowel
    "early" starts with a vowel
    "is" starts with a vowel
*/
```

См. также

- Справочник по C#
- Ключевые слова запроса (LINQ)
- LINQ в C#
- LINQ
- Обработка исключений в выражениях запросов

ascending (Справочник по C#)

18.12.2021 • 2 minutes to read

Контекстное ключевое слово `ascending` используется в предложении orderby в выражении запроса, задавая порядок сортировки от наименьшего значения к наибольшему. Поскольку порядок сортировки `ascending` используется по умолчанию, отдельно задавать его не нужно.

Пример

В следующем примере показано использование ключевого слова `ascending` в предложении orderby.

```
IEnumerable<string> sortAscendingQuery =
    from vegetable in vegetables
    orderby vegetable ascending
    select vegetable;
```

См. также

- Справочник по C#
- LINQ в C#
- descending

descending (Справочник по C#)

18.12.2021 • 2 minutes to read

Контекстное ключевое слово `descending` используется в предложении orderby в выражении запроса, задавая порядок сортировки от наибольшего значения к наименьшему.

Пример

В следующем примере показано использование ключевого слова `descending` в предложении orderby.

```
IEnumerable<string> sortDescendingQuery =
    from vegetable in vegetables
    orderby vegetable descending
    select vegetable;
```

См. также

- Справочник по C#
- LINQ в C#
- ascending

on (Справочник по C#)

18.12.2021 • 2 minutes to read

Контекстное ключевое слово `on` используется в предложении join выражения запроса, задавая условие объединения.

Пример

В следующем примере показано использование ключевого слова `on` в предложении `join`.

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name };
```

См. также

- Справочник по C#
- LINQ

equals (Справочник по C#)

18.12.2021 • 2 minutes to read

Контекстное ключевое слово `equals` используется в предложении `join` в выражении запроса для сравнения элементов двух последовательностей. Дополнительные сведения см. в разделе Предложение join.

Пример

В следующем примере показано использование ключевого слова `equals` в предложении `join`.

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name };
```

См. также

- LINQ

by (Справочник по C#)

18.12.2021 • 2 minutes to read

Контекстное ключевое слово `by` используется в предложении `group` в выражении запроса для определения способа группировки возвращаемых элементов. Дополнительные сведения см. в разделе Предложение group.

Пример

В следующем примере показано использование контекстного слова `by` в предложении `group` для группировки студентов по первой букве фамилии.

```
var query = from student in students
            group student by student.LastName[0];
```

См. также

- LINQ в C#

in (Справочник по C#)

18.12.2021 • 2 minutes to read

Ключевое слово `in` используется в следующих контекстах:

- Параметры универсального типа в универсальных интерфейсах и делегатах.
- В качестве модификатора параметра, который позволяет передавать аргумент в метод по ссылке, а не по значению.
- Операторы foreach.
- Предложения from в выражениях запроса LINQ.
- Предложения join в выражениях запроса LINQ.

См. также

- Ключевые слова в C#
- Справочник по C#

Операторы и выражения C# (справочник по C#)

18.12.2021 • 4 minutes to read

C# предоставляет ряд операторов. Многие из них поддерживаются встроенными типами и позволяют выполнять базовые операции со значениями этих типов. В число этих операторов входят следующие группы:

- Арифметические операторы, выполняющие арифметические операции с числовыми операндами.
- Операторы сравнения, сравнивающие числовые операнды.
- Логические операторы, выполняющие логические операции с операндами `bool`.
- Битовые операторы и операторы сдвига выполняют битовые операции или операции сдвига с операндами целочисленных типов.
- Операторы равенства проверяют равенство или неравенство своих операндов.

Как правило, можно выполнить перегрузку этих операторов, то есть указать поведение оператора для операндов определяемого пользователем типа.

Простейшими выражениями C# являются литералы (например, целые и реальные числа) и имена переменных. Их можно объединить в сложные выражения с помощью операторов. Приоритет и ассоциативность операторов определяют порядок выполнения операций в выражении. Порядок вычисления, определяемый приоритетом и ассоциативностью операторов, можно изменить с помощью скобок.

В следующем коде примеры выражений находятся в правой части назначений:

```
int a, b, c;
a = 7;
b = a;
c = b++;
b = a + b * c;
c = a >= 100 ? b : c / 10;
a = (int)Math.Sqrt(b * b + c * c);

string s = "String literal";
char l = s[s.Length - 1];

var numbers = new List<int>(new[] { 1, 2, 3 });
b = numbers.FindLast(n => n > 1);
```

Как правило, выражение выдает результат и может быть заключено в другое выражение. Вызов метода `void` является примером выражения, которое дает результат. Его можно использовать только в качестве оператора, как показано в следующем примере:

```
Console.WriteLine("Hello, world!");
```

Ниже приведены некоторые другие виды выражений, доступные в C#:

- Выражения интерполированных строк, которые предоставляют удобный синтаксис для создания форматированных строк:

```
var r = 2.3;
var message = $"The area of a circle with radius {r} is {Math.PI * r * r:F3}.";
Console.WriteLine(message);
// Output:
// The area of a circle with radius 2.3 is 16.619.
```

- Лямбда-выражения, позволяющие создавать анонимные функции:

```
int[] numbers = { 2, 3, 4, 5 };
var maximumSquare = numbers.Max(x => x * x);
Console.WriteLine(maximumSquare);
// Output:
// 25
```

- Выражения запроса, позволяющие использовать возможности запросов непосредственно в C#:

```
var scores = new[] { 90, 97, 78, 68, 85 };
IEnumerable<int> highScoresQuery =
    from score in scores
    where score > 80
    orderby score descending
    select score;
Console.WriteLine(string.Join(" ", highScoresQuery));
// Output:
// 97 90 85
```

Определение тела выражения можно использовать, чтобы предоставить краткое определение для метода, конструктора, свойства, индексатора или метода завершения.

Приоритет операторов

В выражении с несколькими операторами операторы с более высоким приоритетом оцениваются до операторов с более низким приоритетом. В следующем примере умножение выполняется сначала, так как оно имеет более высокий приоритет, чем сложение:

```
var a = 2 + 2 * 2;
Console.WriteLine(a); //  output: 6
```

Используйте скобки, чтобы изменить порядок вычисления, накладываемый приоритетом операторов:

```
var a = (2 + 2) * 2;
Console.WriteLine(a); //  output: 8
```

В следующей таблице перечислены операторы C# в порядке убывания приоритета. Операторы в каждой строке имеют одинаковый приоритет.

ОПЕРАТОРЫ	КАТЕГОРИЯ ИЛИ ИМЯ
x.y, f(x), a[i], `x?.y`, `x?[y]`, x++, x--, x!, new, typeof, checked, unchecked, default, nameof, delegate, sizeof, stackalloc, x->y	Первичный
+x, -x, !x, ~x, ++x, --x, ^x, (T)x, await, &x, *x, true и false	Унарный

ОПЕРАТОРЫ	КАТЕГОРИЯ ИЛИ ИМЯ
x..y	Диапазон
switch, with	Выражения `switch` и `with`
x * y, x / y, x % y	Мультипликативный
x + y, x – y	Аддитивный
x << y, x >> y	Сдвиг
x < y, x > y, x <= y, x >= y, is, as	Тестирование типов и относительный
x == y, x != y	Равенство
`x & y`	Логическое И или побитовое логическое И
`x ^ y`	Логическое исключающее ИЛИ или побитовое логическое исключающее ИЛИ
`x | y`	Логическое ИЛИ или побитовое логическое ИЛИ
x && y	Условное И
x || y	Условное ИЛИ
x ?? y	Оператор объединения с NULL
c ? t : f	Условный оператор
x = y, x += y, x -= y, x *= y, x /= y, x %= y, x &= y, x |= y, x ^= y, x <<= y, x >>= y, x ??= y, =>	Назначение и объявление лямбда-выражений

Ассоциативность операторов

Если операторы имеют одинаковый приоритет, порядок их выполнения определяется ассоциативностью операторов:

- Операторы с *левой ассоциативностью* вычисляются слева направо. За исключением операторов присваивания и оператора объединения со значением NULL, все бинарные операторы имеют левую ассоциативность. Например, выражение `a + b - c` вычисляется как `(a + b) - c`.
- Операторы с *правой ассоциативностью* вычисляются справа налево. Операторы присваивания, оператор объединения со значением NULL и условный оператор `?:` имеют правую ассоциативность. Например, выражение `x = y = z` вычисляется как `x = (y = z)`.

Используйте скобки, чтобы изменить порядок вычисления, накладываемый ассоциативностью операторов:

```
int a = 13 / 5 / 2;
int b = 13 / (5 / 2);
Console.WriteLine($"a = {a}, b = {b}");  // output: a = 1, b = 6
```

Вычисление операнда

Не связанные с приоритетом и ассоциативностью операторов операнды в выражении вычисляются слева направо. В следующих примерах иллюстрируется порядок вычисления операторов и операндов:

ВЫРАЖЕНИЕ	ПОРЯДОК ВЫЧИСЛЕНИЙ
`a + b`	a, b, +
`a + b * c`	a, b, c, *, +
`a / b + c * d`	a, b, /, c, d, *, +
`a / (b + c) * d`	a, b, c, +, /, d, *

Как правило, оцениваются все операнды операторов. Однако некоторые операторы оценивают операнды условно. То есть значение крайнего левого операнда такого оператора определяет, следует ли оценивать другие операнды. Эти операторы являются условными логическими операторами И (`&&`) и ИЛИ (`||`) , операторами объединения со значением NULL `??` и `??=` , условными операторами со значением NULL `?.` и `?[]` и условным оператором `?:` . Дополнительные сведения см. в описании каждого оператора.

Спецификация языка C#

Дополнительные сведения см. в следующих разделах статьи Спецификация языка C#:

- Выражения
- Инструкции

См. также

- справочник по C#
- Перегрузка операторов
- Деревья выражений

Арифметические операторы (справочник по C#)

18.12.2021 • 8 minutes to read

Следующие операторы выполняют арифметические операции с операндами числовых типов:

- унарные — `++` (приращение), `--` (уменьшение), `+` (плюс) и `-` (минус);
- бинарные — `*` (умножение), `/` (деление), `%` (остаток от деления), `+` (сложение) и `-` (вычитание).

Эти операторы поддерживаются всеми целочисленными типами и типами с плавающей запятой.

В случае целочисленных типов эти операторы (за исключением операторов `++` и `--`) определяются для типов `int`, `uint`, `long` и `ulong`. Если операнды принадлежат к другим целочисленным типам (`sbyte`, `byte`, `short`, `ushort` или `char`), их значения преобразуются в тип `int`, который также является типом результата операции. Если операнды принадлежат к разным целочисленным типам или типам с плавающей запятой, их значения преобразуются в ближайший содержащий тип, если такой тип существует. Дополнительные сведения см. в разделе Числовые повышения уровня в статье Спецификации языка C#. Операторы `++` и `--` определяются для всех целочисленных числовых типов и числовых типов с плавающей запятой, а также типа char.

Оператор инкремента ++

Оператор инкремента `++` увеличивает операнд на 1. Операндом должна быть переменная, свойство или индексатор.

Оператор инкремента поддерживается в двух формах: постфиксный оператор инкремента (`x++`) и префиксный оператор инкремента (`++x`).

Постфиксный оператор приращения

Результатом `x++` является значение `x` *перед* выполнением операции, как показано в следующем примере:

```
int i = 3;
Console.WriteLine(i);   // output: 3
Console.WriteLine(i++); // output: 3
Console.WriteLine(i);   // output: 4
```

Префиксный оператор инкремента

Результатом `++x` является значение `x` *после* выполнения операции, как показано в следующем примере:

```
double a = 1.5;
Console.WriteLine(a);   // output: 1.5
Console.WriteLine(++a); // output: 2.5
Console.WriteLine(a);   // output: 2.5
```

Оператор декремента --

Унарный оператор декремента `--` уменьшает операнд на 1. Операндом должна быть переменная, свойство или индексатор.

Оператор декремента поддерживается в двух формах: постфиксный оператор декремента (`x--`) и

префиксный оператор декремента (`--x`).

Постфиксный оператор уменьшения

Результатом `x--` является значение `x` *перед* выполнением операции, как показано в следующем примере:

```
int i = 3;
Console.WriteLine(i);   // output: 3
Console.WriteLine(i--); // output: 3
Console.WriteLine(i);   // output: 2
```

Префиксный оператор декремента

Результатом `--x` является значение `x` *после* выполнения операции, как показано в следующем примере:

```
double a = 1.5;
Console.WriteLine(a);   // output: 1.5
Console.WriteLine(--a); // output: 0.5
Console.WriteLine(a);   // output: 0.5
```

Операторы унарного плюса и минуса

Унарный оператор `+` возвращает значение полученного операнда. Унарный оператор `-` изменяет знак операнда на противоположный.

```
Console.WriteLine(+4);     // output: 4

Console.WriteLine(-4);     // output: -4
Console.WriteLine(-(-4));  // output: 4

uint a = 5;
var b = -a;
Console.WriteLine(b);            // output: -5
Console.WriteLine(b.GetType());  // output: System.Int64

Console.WriteLine(-double.NaN);  // output: NaN
```

Тип ulong не поддерживает унарный оператор `-`.

Оператор умножения *

Оператор умножения `*` вычисляет произведение операндов:

```
Console.WriteLine(5 * 2);         // output: 10
Console.WriteLine(0.5 * 2.5);     // output: 1.25
Console.WriteLine(0.1m * 23.4m);  // output: 2.34
```

Унарный оператор `*` представляет собой оператор косвенного обращения к указателю.

Оператор деления /

Оператор деления `/` делит левый операнд на правый.

Деление целых чисел

Для операндов цельночисленных типов результат оператора `/` является целочисленным типом, который равен частному двух операндов, округленному в сторону нуля:

```
Console.WriteLine(13 / 5);    // output: 2
Console.WriteLine(-13 / 5);   // output: -2
Console.WriteLine(13 / -5);   // output: -2
Console.WriteLine(-13 / -5);  // output: 2
```

Чтобы получить частное двух операндов в виде числа с плавающей запятой, используйте тип `float`, `double` или `decimal`:

```
Console.WriteLine(13 / 5.0);       // output: 2.6

int a = 13;
int b = 5;
Console.WriteLine((double)a / b);  // output: 2.6
```

Деление чисел с плавающей запятой

Для типов `float`, `double` и `decimal` результатом оператора `/` является частное двух операндов:

```
Console.WriteLine(16.8f / 4.1f);   // output: 4.097561
Console.WriteLine(16.8d / 4.1d);   // output: 4.09756097560976
Console.WriteLine(16.8m / 4.1m);   // output: 4.0975609756097560975609756098
```

Если один из операндов — это `decimal`, второй операнд не может быть ни `float`, ни `double`, так как ни `float`, ни `double` не преобразуется неявно в тип `decimal`. Необходимо явным образом преобразовать операнд `float` или `double` в тип `decimal`. Дополнительные сведения о числовых преобразованиях см. в разделе Встроенные числовые преобразования.

Оператор остатка %

Оператор остатка `%` вычисляет остаток от деления левого операнда на правый.

Целочисленный остаток

Для целочисленных операндов результатом `a % b` является значение, произведенное `a - (a / b) * b`. Знак ненулевого остатка такой же, как и у левого операнда, как показано в следующем примере:

```
Console.WriteLine(5 % 4);   // output: 1
Console.WriteLine(5 % -4);  // output: 1
Console.WriteLine(-5 % 4);  // output: -1
Console.WriteLine(-5 % -4); // output: -1
```

Используйте метод Math.DivRem для вычисления результатов как целочисленного деления, так и определения остатка.

Остаток с плавающей запятой

Для операндов типа `float` и `double` результатом `x % y` для конечных `x` и `y` будет значение `z`, так что:

- знак `z`, если отлично от нуля, совпадает со знаком `x`;
- абсолютное значение `z` является значением, произведенным `|x| - n * |y|`, где `n` — это наибольшее возможное целое число, которое меньше или равно `|x| / |y|`, а `|x|` и `|y|` являются абсолютными значениями `x` и `y`, соответственно.

> **NOTE**
> Этот метод вычисления остатка аналогичен тому, который использовался для целочисленных операндов, но отличается от спецификации IEEE 754. Если вам нужна операция вычисления остатка, которая соответствует спецификации IEEE 754, используйте метод Math.IEEERemainder.

Сведения о поведение оператора `%` в случае неконечных операндов см. в разделе Оператор остаткаспецификации языка C#.

Для операндов `decimal` оператор остатка `%` эквивалентен оператору остатка типа System.Decimal.

В следующем примере показано поведение оператора остатка для операндов с плавающей запятой:

```
Console.WriteLine(-5.2f % 2.0f); // output: -1.2
Console.WriteLine(5.9 % 3.1);    // output: 2.8
Console.WriteLine(5.9m % 3.1m);  // output: 2.8
```

Оператор сложения +

Оператор сложения `+` вычисляет сумму своих операндов:

```
Console.WriteLine(5 + 4);       // output: 9
Console.WriteLine(5 + 4.3);     // output: 9.3
Console.WriteLine(5.1m + 4.2m); // output: 9.3
```

Кроме того, оператор `+` можно использовать для объединения строк и делегатов. Дополнительные сведения см. в статье Операторы `+` и `+=`.

Оператор вычитания -

Оператор вычитания `-` вычитает правый операнд из левого:

```
Console.WriteLine(47 - 3);      // output: 44
Console.WriteLine(5 - 4.3);     // output: 0.7
Console.WriteLine(7.5m - 2.3m); // output: 5.2
```

Кроме того, оператор `-` можно использовать для удаления делегатов. Дополнительные сведения см. в статье Операторы `-` и `-=`.

Составное присваивание

Для бинарного оператора `op` выражение составного присваивания в форме

```
x op= y
```

эквивалентно

```
x = x op y
```

за исключением того, что `x` вычисляется только один раз.

Следующий пример иллюстрирует использование составного присваивания с арифметическими

операторами:

```
int a = 5;
a += 9;
Console.WriteLine(a);  // output: 14

a -= 4;
Console.WriteLine(a);  // output: 10

a *= 2;
Console.WriteLine(a);  // output: 20

a /= 4;
Console.WriteLine(a);  // output: 5

a %= 3;
Console.WriteLine(a);  // output: 2
```

Из-за восходящих приведений результат операции `op` может быть невозможно неявно преобразовать в тип `T` из `x`. В этом случае, если `op` является предопределенным оператором, и результат операции является явно преобразуемым в тип `T``x`, выражение составного присваивания формы `x op= y` эквивалентно `x = (T)(x op y)`, за исключением того, что `x` вычисляется только один раз. В следующем примере продемонстрировано такое поведение.

```
byte a = 200;
byte b = 100;

var c = a + b;
Console.WriteLine(c.GetType());  // output: System.Int32
Console.WriteLine(c);  // output: 300

a += b;
Console.WriteLine(a);  // output: 44
```

Вы также можете использовать операторы `+=` и `-=` для подписки и отмены подписки на события соответственно. Дополнительные сведения см. в разделе Практическое руководство. Подписка и отмена подписки на события.

Приоритет и ассоциативность операторов

В следующем списке перечислены арифметические операторы в порядке убывания приоритета:

- Постфиксный инкремент `x++` и декремент `x--`
- Префиксный инкремент `++x` и декремент `--x`, унарные операторы `+` и `-`
- Мультипликативные операторы `*`, `/`, и `%`
- Аддитивные операторы `+` и `-`

Бинарные арифметические операторы имеют левую ассоциативность. То есть операторы с одинаковым приоритетом вычисляются в направлении слева направо.

Порядок вычисления, определяемый приоритетом и ассоциативностью операторов, можно изменить с помощью скобок (`()`).

```
Console.WriteLine(2 + 2 * 2);   // output: 6
Console.WriteLine((2 + 2) * 2); // output: 8

Console.WriteLine(9 / 5 / 2);   // output: 0
Console.WriteLine(9 / (5 / 2)); // output: 4
```

Полный список операторов C#, упорядоченный по уровню приоритета, можно найти в разделе Приоритет операторов статьи Операторы C#.

Арифметическое переполнение и деление на нуль

Если результат арифметической операции выходит за пределы диапазона возможных конечных значений соответствующего числового типа, поведение арифметического оператора зависит от типа его операндов.

Целочисленное арифметическое переполнение

Деление целого числа на ноль всегда вызывает исключение DivideByZeroException.

В случае целочисленного арифметического переполнения итоговое поведение определяется контекстом проверки переполнения, который может быть проверяемым или непроверяемым:

- Если в проверяемом контексте переполнение возникает в константном выражении, происходит ошибка времени компиляции. В противном случае, если операция производится во время выполнения, возникает исключение OverflowException.
- В непроверяемом контексте результат усекается путем удаления старших разрядов, которые не помещаются в целевой тип данных.

Вместе с проверяемыми и непроверяемыми операторами вы можете использовать операторы `checked` и `unchecked`, чтобы управлять контекстом проверки переполнения, в котором вычисляется выражение:

```
int a = int.MaxValue;
int b = 3;

Console.WriteLine(unchecked(a + b));  // output: -2147483646
try
{
    int d = checked(a + b);
}
catch(OverflowException)
{
    Console.WriteLine($"Overflow occurred when adding {a} to {b}.");
}
```

По умолчанию арифметические операции выполняются в *непроверяемом* контексте.

Арифметическое переполнение с плавающей запятой

Арифметические операции с типами `float` и `double` никогда не вызывают исключение. Результатом арифметических операций с этими типами может быть одно из специальных значений, представляющих бесконечность и объект, не являющийся числовым:

```
double a = 1.0 / 0.0;
Console.WriteLine(a);                    // output: Infinity
Console.WriteLine(double.IsInfinity(a)); // output: True

Console.WriteLine(double.MaxValue + double.MaxValue); // output: Infinity

double b = 0.0 / 0.0;
Console.WriteLine(b);                // output: NaN
Console.WriteLine(double.IsNaN(b));  // output: True
```

Для операндов типа `decimal` арифметическое переполнение всегда вызывает исключение OverflowException, а деление на нуль всегда вызывает исключение DivideByZeroException.

Ошибки округления

Из-за общих ограничений, касающихся представления вещественных чисел в форме с плавающей запятой и арифметических операций с плавающей запятой, при вычислениях с использованием типов с плавающей запятой могут возникать ошибки округления. То есть полученный результат выражения может отличаться от ожидаемого математического результата. В следующем примере показано несколько таких случаев:

```
Console.WriteLine(.41f % .2f); // output: 0.00999999

double a = 0.1;
double b = 3 * a;
Console.WriteLine(b == 0.3);   // output: False
Console.WriteLine(b - 0.3);    // output: 5.55111512312578E-17

decimal c = 1 / 3.0m;
decimal d = 3 * c;
Console.WriteLine(d == 1.0m);  // output: False
Console.WriteLine(d);          // output: 0.99999999999999999999999999
```

См. заметки в справочной документации по System.Double, System.Single и System.Decimal.

Возможность перегрузки оператора

Определяемый пользователем тип может перегружать унарные (`++` , `--` , `+` и `-`) и бинарные (`*` , `/` , `%` , `+` и `-`) арифметические операторы. При перегрузке бинарного оператора соответствующий оператор составного присваивания также неявно перегружается. Определяемый пользователем тип не может перегружать оператор составного присваивания явным образом.

Спецификация языка C#

Дополнительные сведения см. в следующих разделах статьи Спецификация языка C#:

- Постфиксные операторы инкремента и декремента
- Префиксные операторы инкремента и декремента
- Оператор унарного плюса
- Оператор унарного минуса
- Оператор умножения
- Оператор деления
- Оператор остатка
- Оператор сложения
- Оператор вычитания

- Составное присваивание
- Операторы checked и unchecked
- Восходящие приведения числовых типов

См. также

- справочник по C#
- Операторы и выражения C#
- System.Math
- System.MathF
- Числовые значения в .NET

Логические операторы (справочник по C#)

18.12.2021 • 7 minutes to read

Следующие операторы выполняют логические операции с использованием логических операндов:

- Унарный `!` (логическое отрицание) оператор.
- Бинарные `&` (логическое И), `|` (логическое ИЛИ), а также `^` (логическое исключающее ИЛИ) операторы. Эти операторы всегда обрабатывают оба операнда.
- Бинарные `&&` (условное логическое И) и `||` (условное логическое ИЛИ) операторы. Эти операторы вычисляют правый операнд, только если это необходимо.

Для операндов целочисленных типов операторы `&`, `|` и `^` выполняют побитовые логические операции. Дополнительные сведения см. в разделе Побитовые операторы и операторы сдвига.

Оператор логического отрицания !

Унарный префиксный оператор `!` выполняет логическое отрицание операнда, возвращая `true`, если операнд имеет значение `false`, и `false`, если операнд имеет значение `true`.

```
bool passed = false;
Console.WriteLine(!passed);  // output: True
Console.WriteLine(!true);    // output: False
```

Начиная с версии C# 8.0, унарный постфиксный оператор `!` допускает значение NULL.

Оператор логического И &

Оператор `&` вычисляет логическое И для всех своих операндов. Результат операции `x & y` принимает значение `true`, если оба оператора `x` и `y` имеют значение `true`. В противном случае результат будет `false`.

Оператор `&` вычисляет оба операнда, даже если левый операнд имеет значение `false`. При этом операция должна вернуть значение `false`, независимо от значения правого операнда.

В следующем примере правый операнд оператора `&` является вызовом метода, который выполняется независимо от значения левого операнда:

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = false & SecondOperand();
Console.WriteLine(a);
// Output:
// Second operand is evaluated.
// False

bool b = true & SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

Условный оператор логического И `&&` также вычисляет логическое И для своих операндов, но не вычисляет правый операнд, если левый операнд имеет значение `false`.

Для операндов целочисленных типов оператор `&` вычисляет побитовое логическое И своих операндов. Унарный оператор `&` является оператором AddressOf.

Оператор логического исключения ИЛИ ^

Оператор `^` вычисляет логическое исключение ИЛИ для всех своих операндов, возвращая `true` для `x ^ y`, если `x` имеет значение `true` и `y` имеет значение `false` или `x` имеет значение `false` и `y` имеет значение `true`. В противном случае результат будет `false`. То есть для операндов `bool` оператор `^` возвращает тот же результат, что и оператор неравенства `!=`.

```
Console.WriteLine(true ^ true);    // output: False
Console.WriteLine(true ^ false);   // output: True
Console.WriteLine(false ^ true);   // output: True
Console.WriteLine(false ^ false);  // output: False
```

Для операндов целочисленных типов оператор `^` вычисляет побитовое исключающее ИЛИ своих операндов.

Оператор логического ИЛИ |

Оператор `|` вычисляет логическое ИЛИ для всех своих операндов. Результат операции `x | y` принимает значение `true`, если хотя бы один из операторов `x` или `y` имеет значение `true`. В противном случае результат будет `false`.

Оператор `|` вычисляет оба операнда, даже если левый операнд имеет значение `true`. При этом операция должна вернуть значение `true`, независимо от значения правого операнда.

В следующем примере правый операнд оператора `|` является вызовом метода, который выполняется независимо от значения левого операнда:

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = true | SecondOperand();
Console.WriteLine(a);
// Output:
// Second operand is evaluated.
// True

bool b = false | SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

Условный оператор логического ИЛИ `||` также вычисляет логическое ИЛИ для своих операндов, но не вычисляет правый операнд, если левый операнд имеет значение `true`.

Для операндов целочисленных типов оператор `|` вычисляет побитовое логическое ИЛИ своих операндов.

Условный оператор логического И &&

Условный оператор логического И `&&` (оператор короткого замыкания) вычисляет логическое И для своих операндов. Результат операции `x && y` принимает значение `true`, если оба оператора `x` и `y` имеют значение `true`. В противном случае результат будет `false`. Если `x` имеет значение `false`, `y` не вычисляется.

В следующем примере правый операнд оператора `&&` является вызовом метода, который не выполняется, если левый операнд имеет значение `false`:

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = false && SecondOperand();
Console.WriteLine(a);
// Output:
// False

bool b = true && SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

Оператор логического И `&` также вычисляет логическое И для своих операндов, но он всегда вычисляет оба операнда.

Условный оператор логического ИЛИ ||

Условный оператор логического ИЛИ `||` (оператор короткого замыкания) вычисляет логическое ИЛИ для своих операндов. Результат операции `x || y` принимает значение `true`, если хотя бы один из операторов `x` или `y` имеет значение `true`. В противном случае результат будет `false`. Если `x` имеет

значение `true`, `y` не вычисляется.

В следующем примере правый операнд оператора `||` является вызовом метода, который не выполняется, если левый операнд имеет значение `true`:

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = true || SecondOperand();
Console.WriteLine(a);
// Output:
// True

bool b = false || SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

Оператор логического ИЛИ `|` также вычисляет логическое ИЛИ для своих операндов, но всегда вычисляет оба операнда.

Операторы, допускающие логическое значение NULL

Для операндов `bool?` операторы `&` (логическое И) и `|` (логическое ИЛИ) поддерживают следующую логику с тремя значениями:

- Оператор `&` возвращает `true` только в том случае, если оба операнда имеют значение `true`. Если `x` или `y` имеет значение `false`, оператор `x & y` возвращает `false` (даже если другой операнд имеет значение `null`). В противном случае выражение `x & y` будет иметь значение `null`.
- Оператор `|` возвращает `false` только в том случае, если оба операнда имеют значение `false`. Если `x` или `y` имеет значение `true`, оператор `x | y` возвращает `true` (даже если другой операнд имеет значение `null`). В противном случае выражение `x | y` будет иметь значение `null`.

Эта семантика описывается в следующей таблице:

x	y	x&y	x|y
true	true	true	true
true	false	false	true
true	null	null	true
false	true	false	true
false	false	false	false
false	null	false	null
null	true	null	true

x	y	x&y	x|y
null	false	false	null
null	null	null	null

Поведение этих операторов отличается от типичного поведения операторов, допускающих значение NULL. Как правило, оператор, который определяется для операндов типа значения, можно также использовать с соответствующими операндами типа, допускающего значение NULL. Такой оператор возвращает `null`, если какой-либо из операндов имеет значение `null`. При этом операторы `&` и `|` могут возвращать отличное от NULL значение, даже если один из операндов имеет значение `null`. См. подробнее о поведении операторов, допускающих значение NULL, в разделе Операторы с нулификацией в статье Типы, допускающие значение NULL.

Вы также можете также использовать операторы `!` и `^` с операндами `bool?`, как показано в следующем примере:

```
bool? test = null;
Display(!test);         // output: null
Display(test ^ false);  // output: null
Display(test ^ null);   // output: null
Display(true ^ null);   // output: null

void Display(bool? b) => Console.WriteLine(b is null ? "null" : b.Value.ToString());
```

Условные логические операторы `&&` и `||` не поддерживают операнды типа `bool?`.

Составное присваивание

Для бинарного оператора `op` выражение составного присваивания в форме

```
x op= y
```

эквивалентно

```
x = x op y
```

за исключением того, что `x` вычисляется только один раз.

Операторы `&`, `|` и `^` поддерживают составное присваивание, как показано в следующем примере:

```
bool test = true;
test &= false;
Console.WriteLine(test);  // output: False

test |= true;
Console.WriteLine(test);  // output: True

test ^= false;
Console.WriteLine(test);  // output: True
```

> **NOTE**
> Условные логические операторы `&&` и `||` не поддерживают составное присваивание.

Приоритет операторов

В следующем списке перечислены логические операторы в порядке убывания приоритета:

- Оператор логического отрицания `!`
- Оператор логического И `&`
- Оператор логического исключающего ИЛИ `^`
- Оператор логического ИЛИ `|`
- Условный оператор логического И `&&`
- Условный оператор логического ИЛИ `||`

Порядок вычисления, определяемый приоритетом операторов, можно изменить с помощью скобок (`()`).

```
Console.WriteLine(true | true & false);   // output: True
Console.WriteLine((true | true) & false); // output: False

bool Operand(string name, bool value)
{
    Console.WriteLine($"Operand {name} is evaluated.");
    return value;
}

var byDefaultPrecedence = Operand("A", true) || Operand("B", true) && Operand("C", false);
Console.WriteLine(byDefaultPrecedence);
// Output:
// Operand A is evaluated.
// True

var changedOrder = (Operand("A", true) || Operand("B", true)) && Operand("C", false);
Console.WriteLine(changedOrder);
// Output:
// Operand A is evaluated.
// Operand C is evaluated.
// False
```

Полный список операторов C#, упорядоченный по уровню приоритета, можно найти в разделе Приоритет операторов статьи Операторы C#.

Возможность перегрузки оператора

Определяемый пользователем тип может перегружать операторы `!`, `&`, `|` и `^`. При перегрузке бинарного оператора соответствующий оператор составного присваивания также неявно перегружается. Определяемый пользователем тип не может перегружать оператор составного присваивания явным образом.

Определяемый пользователем тип не может перегружать условные логические операторы `&&` и `||`. При этом, если определяемый пользователем тип каким-либо образом перегружает операторы true и false и операторы `&` и `|`, операция `&&` или `||` может быть применена для операндов этого типа. Дополнительные сведения см. в разделе Пользовательские условные логические операторы в Спецификации языка C#.

Спецификация языка C#

Дополнительные сведения см. в следующих разделах статьи Спецификация языка C#:

- Оператор логического отрицания
- Логические операторы
- Условные логические операторы
- Составное присваивание

См. также

- справочник по C#
- Операторы и выражения C#
- Побитовые операторы и операторы сдвига

Побитовые операторы и операторы сдвига (справочник по C#)

18.12.2021 • 6 minutes to read

Следующие операторы выполняют побитовые операции или операции сдвига с операндами целочисленных или символьных типов:

- Унарный оператор `~` (побитовое дополнение)
- Бинарные операторы сдвига `<<` (сдвиг влево) и `>>` (сдвиг вправо)
- Бинарные операторы `&` (логическое и), `|` (логическое или) и `^` (логическое исключающее или)

Эти операторы определены для типов `int`, `uint`, `long` и `ulong`. Если оба операнда имеют другие целочисленные типы (`sbyte`, `byte`, `short`, `ushort` или `char`), их значения преобразуются в тип `int`, который также является типом результата операции. Если операнды имеют разные целочисленные типы, их значения преобразуются в ближайший содержащий целочисленный тип. Дополнительные сведения см. в разделе Числовые повышения уровня в статье Спецификации языка C#.

Операторы `&`, `|` и `^` также определены для операндов типа `bool`. Дополнительные сведения см. в разделе Логические операторы.

Побитовые операции и операции сдвига никогда не вызывают переполнение и дают одинаковые результаты в проверенных и непроверенных контекстах.

Оператор побитового дополнения ~

Оператор `~` создает побитовое дополнение своего операнда путем инвертирования каждого бита:

```
uint a = 0b_0000_1111_0000_1111_0000_1111_0000_1100;
uint b = ~a;
Console.WriteLine(Convert.ToString(b, toBase: 2));
// Output:
// 11110000111100001111000011110011
```

Можно также использовать символ `~` для объявления методов завершения. Дополнительные сведения см. в разделе Методы завершения.

Оператор сдвига влево <<

Оператор `<<` сдвигает левый операнд влево на количество битов, определенное правым операндом.

Операция сдвига влево отбрасывает старшие биты, которые находятся за пределами диапазона типа результата, и задает позиции пустых битов низкого порядка, равные нулю, как показано в следующем примере:

```
uint x = 0b_1100_1001_0000_0000_0000_0000_0001_0001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2)}");

uint y = x << 4;
Console.WriteLine($"After:  {Convert.ToString(y, toBase: 2)}");
// Output:
// Before: 11001001000000000000000000010001
// After:  10010000000000000000000100010000
```

Поскольку операторы сдвига определены только для типов `int`, `uint`, `long` и `ulong`, результат операции всегда содержит по крайней мере 32 бита. Если левый операнд имеет другой целочисленный тип (`sbyte`, `byte`, `short`, `ushort` или `char`), его значение преобразуется в тип `int`, как показано в следующем примере:

```
byte a = 0b_1111_0001;

var b = a << 8;
Console.WriteLine(b.GetType());
Console.WriteLine($"Shifted byte: {Convert.ToString(b, toBase: 2)}");
// Output:
// System.Int32
// Shifted byte: 1111000100000000
```

Сведения о том, как правый операнд оператора `<<` определяет величину сдвига, см. в разделе Величина смещения операторов сдвига.

Оператор сдвига вправо >>

Оператор `>>` сдвигает левый операнд вправо на количество битов, определенное правым операндом.

Операция сдвига вправо удаляет младшие разряды, как показано в следующем примере:

```
uint x = 0b_1001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2), 4}");

uint y = x >> 2;
Console.WriteLine($"After:  {Convert.ToString(y, toBase: 2), 4}");
// Output:
// Before: 1001
// After:    10
```

Позиции пустых битов высокого порядка задаются с учетом типа левого операнда следующим образом:

- Если левый операнд имеет тип `int` или `long`, оператор сдвига вправо выполняет *арифметический* сдвиг, то есть значение наиболее значимого бита (знаковый бит) левого операнда переносится в пустые битовые позиции высокого разряда. То есть для пустых битовых позиций высокого порядка задается ноль, если левый операнд неотрицательный, и единица, если он отрицательный.

  ```
  int a = int.MinValue;
  Console.WriteLine($"Before: {Convert.ToString(a, toBase: 2)}");

  int b = a >> 3;
  Console.WriteLine($"After:  {Convert.ToString(b, toBase: 2)}");
  // Output:
  // Before: 10000000000000000000000000000000
  // After:  11110000000000000000000000000000
  ```

- Если левый операнд имеет тип `uint` или `ulong`, оператор сдвига вправо выполняет *логический* сдвиг, то есть пустым битовым позициям высокого порядка всегда присваивается нулевое значение.

```
uint c = 0b_1000_0000_0000_0000_0000_0000_0000_0000;
Console.WriteLine($"Before: {Convert.ToString(c, toBase: 2), 32}");

uint d = c >> 3;
Console.WriteLine($"After:  {Convert.ToString(d, toBase: 2), 32}");
// Output:
// Before: 10000000000000000000000000000000
// After:     10000000000000000000000000000
```

Сведения о том, как правый операнд оператора `>>` определяет величину сдвига, см. в разделе Величина смещения операторов сдвига.

Оператор логического И &

Оператор `&` вычисляет побитовое логическое И целочисленных операндов:

```
uint a = 0b_1111_1000;
uint b = 0b_1001_1101;
uint c = a & b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 10011000
```

Для операндов типа `bool` оператор `&` вычисляет логическое И по своим операндам. Унарный оператор `&` является оператором AddressOf.

Оператор логического исключения ИЛИ ^

Оператор `^` вычисляет побитовое логическое исключающее ИЛИ, также известное как побитовое логическое XOR, своих целочисленных операндов:

```
uint a = 0b_1111_1000;
uint b = 0b_0001_1100;
uint c = a ^ b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 11100100
```

Для операндов типа `bool` оператор `^` вычисляет логическое исключающее ИЛИ по своим операндам.

Оператор логического ИЛИ |

Оператор `|` вычисляет побитовое логическое ИЛИ своих целочисленных операндов:

```
uint a = 0b_1010_0000;
uint b = 0b_1001_0001;
uint c = a | b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 10110001
```

Для операндов типа `bool` оператор `|` вычисляет логическое ИЛИ по своим операндам.

Составное присваивание

Для бинарного оператора `op` выражение составного присваивания в форме

```
x op= y
```

эквивалентно

```
x = x op y
```

за исключением того, что `x` вычисляется только один раз.

В следующем примере показано использование составного присваивания с побитовыми операторами и операторами сдвига:

```
uint INITIAL_VALUE = 0b_1111_1000;

uint a = INITIAL_VALUE;
a &= 0b_1001_1101;
Display(a);  // output: 10011000

a = INITIAL_VALUE;
a |= 0b_0011_0001;
Display(a);  // output: 11111001

a = INITIAL_VALUE;
a ^= 0b_1000_0000;
Display(a);  // output: 1111000

a = INITIAL_VALUE;
a <<= 2;
Display(a);  // output: 1111100000

a = INITIAL_VALUE;
a >>= 4;
Display(a);  // output: 1111

void Display(uint x) => Console.WriteLine($"{Convert.ToString(x, toBase: 2), 8}");
```

Из-за восходящих приведений результат операции `op` может быть невозможно неявно преобразовать в тип `T` из `x`. В этом случае, если `op` является предопределенным оператором, и результат операции является явно преобразуемым в тип `T``x`, выражение составного присваивания формы `x op= y` эквивалентно `x = (T)(x op y)`, за исключением того, что `x` вычисляется только один раз. В следующем примере продемонстрировано такое поведение.

```
byte x = 0b_1111_0001;

int b = x << 8;
Console.WriteLine($"{Convert.ToString(b, toBase: 2)}");  // output: 1111000100000000

x <<= 8;
Console.WriteLine(x);  // output: 0
```

Приоритет операторов

Следующий список упорядочивает побитовые операторы и операторы сдвига по приоритету, от высокого до низкого:

- Оператор побитового дополнения `~`
- Операторы сдвига `<<` и `>>`
- Оператор логического И `&`
- Оператор логического исключающего ИЛИ `^`
- Оператор логического ИЛИ `|`

Порядок вычисления, определяемый приоритетом операторов, можно изменить с помощью скобок (`()`).

```
uint a = 0b_1101;
uint b = 0b_1001;
uint c = 0b_1010;

uint d1 = a | b & c;
Display(d1);  // output: 1101

uint d2 = (a | b) & c;
Display(d2);  // output: 1000

void Display(uint x) => Console.WriteLine($"{Convert.ToString(x, toBase: 2), 4}");
```

Полный список операторов C#, упорядоченный по уровню приоритета, можно найти в разделе Приоритет операторов статьи Операторы C#.

Величина смещения операторов сдвига

Для операторов сдвига `<<` и `>>` тип правого операнда должен быть `int` или типом, для которого существует предопределенное неявное числовое преобразование в `int` .

Для выражений `x << count` и `x >> count` фактическая величина сдвига зависит от типа `x` следующим образом:

- Если `x` имеет тип `int` или `uint` , величина сдвига определяется младшими *пятью* битами правого операнда. То есть величина сдвига вычисляется на основе `count & 0x1F` (или `count & 0b_1_1111`).
- Если `x` имеет тип `long` или `ulong` , величина сдвига определяется младшими *шестью* битами правого операнда. То есть величина сдвига вычисляется на основе `count & 0x3F` (или `count & 0b_11_1111`).

В следующем примере продемонстрировано такое поведение.

```
int count1 = 0b_0000_0001;
int count2 = 0b_1110_0001;

int a = 0b_0001;
Console.WriteLine($"{a} << {count1} is {a << count1}; {a} << {count2} is {a << count2}");
// Output:
// 1 << 1 is 2; 1 << 225 is 2

int b = 0b_0100;
Console.WriteLine($"{b} >> {count1} is {b >> count1}; {b} >> {count2} is {b >> count2}");
// Output:
// 4 >> 1 is 2; 4 >> 225 is 2
```

> **NOTE**
> Как показано в предыдущем примере, результат операции сдвига может быть ненулевым, даже если значение правого операнда больше числа битов в левом операнде.

Логические операторы перечисления

Операторы `~`, `&`, `|` и `^` также поддерживаются для любого типа перечисления. Для операндов одного типа перечисления логическая операция выполняется по соответствующим значениям базового целочисленного типа. Например, для любого `x` и `y` типа перечисления `T` с базовым типом `U` выражение `x & y` дает тот же результат, что и выражение `(T)((U)x & (U)y)`.

Побитовые логические операторы обычно используются с типом перечисления, который определяется с помощью атрибута Flags. Дополнительные сведения см. в разделе Типы перечислений как битовые флаги в статье Типы перечислений.

Возможность перегрузки оператора

Определяемый пользователем тип может перегружать операторы `~`, `<<`, `>>`, `&`, `|` и `^`. При перегрузке бинарного оператора соответствующий оператор составного присваивания также неявно перегружается. Определяемый пользователем тип не может перегружать оператор составного присваивания явным образом.

Если определяемый пользователем тип `T` перегружает оператор `<<` или `>>`, тип левого операнда должен быть `T`, а тип правого — `int`.

Спецификация языка C#

Дополнительные сведения см. в следующих разделах статьи Спецификация языка C#:

- Оператор побитового дополнения
- Операторы сдвига
- Логические операторы
- Составное присваивание
- Восходящие приведения числовых типов

См. также

- справочник по C#
- Операторы и выражения C#
- Логические операторы

Операторы равенства (справочник по C#)

18.12.2021 • 4 minutes to read

Операторы `==` (равенство) и `!=` (неравенство) проверяют равенство или неравенство своих операндов.

Оператор равенства ==

Оператор равенства `==` возвращает значение `true`, если его операнды равны. В противном случае возвращается значение `false`.

Равенство типов значений

Операнды встроенных типов значений равны, если равны их значения.

```
int a = 1 + 2 + 3;
int b = 6;
Console.WriteLine(a == b);  // output: True

char c1 = 'a';
char c2 = 'A';
Console.WriteLine(c1 == c2);  // output: False
Console.WriteLine(c1 == char.ToLower(c2));  // output: True
```

NOTE

У операторов `==`, `<`, `>`, `<=` и `>=`, если какой-то из операндов не является числом (Double.NaN или Single.NaN), результатом операции является `false`. Это означает, что значение `NaN` не больше, не меньше и не равно любому другому значению `double` (или `float`), включая `NaN`. Дополнительные сведения и примеры см. в справочных статьях по Double.NaN или Single.NaN.

Два операнда одного типа enum равны, если равны соответствующие значения базового целочисленного типа.

По умолчанию пользовательские типы struct не поддерживают оператор `==`. Чтобы поддерживать оператор `==`, пользовательская структура должна перегружать его.

Начиная с версии C# 7.3 операторы `==` и `!=` поддерживаются кортежами C#. Дополнительные сведения см. в разделе Равенство кортежей статьи Типы кортежей.

Равенство ссылочных типов

По умолчанию два операнда ссылочного типа, отличные от записи, являются равными, если они ссылаются на один и тот же объект.

```
public class ReferenceTypesEquality
{
    public class MyClass
    {
        private int id;

        public MyClass(int id) => this.id = id;
    }

    public static void Main()
    {
        var a = new MyClass(1);
        var b = new MyClass(1);
        var c = a;
        Console.WriteLine(a == b);  // output: False
        Console.WriteLine(a == c);  // output: True
    }
}
```

Как показано в примере, определяемые пользователем ссылочные типы поддерживают оператор `==` по умолчанию. Однако ссылочный тип может перегружать оператор `==`. Если ссылочный тип перегружает оператор `==`, воспользуйтесь методом Object.ReferenceEquals, чтобы проверить, что две ссылки этого типа указывают на один и тот же объект.

Равенство типов записей

Типы записей, доступные в C# 9.0 и более поздних версий, поддерживают операторы `==` и `!=`, которые по умолчанию обеспечивают семантику равенства значений. То есть два операнда записи равны, когда оба они равны `null` или равны соответствующие значения всех полей и автоматически реализуемых свойств.

```
public class RecordTypesEquality
{
    public record Point(int X, int Y, string Name);
    public record TaggedNumber(int Number, List<string> Tags);

    public static void Main()
    {
        var p1 = new Point(2, 3, "A");
        var p2 = new Point(1, 3, "B");
        var p3 = new Point(2, 3, "A");

        Console.WriteLine(p1 == p2);  // output: False
        Console.WriteLine(p1 == p3);  // output: True

        var n1 = new TaggedNumber(2, new List<string>() { "A" });
        var n2 = new TaggedNumber(2, new List<string>() { "A" });
        Console.WriteLine(n1 == n2);  // output: False
    }
}
```

Как показано в предыдущем примере, в случае с элементами ссылочного типа, отличными от записей, сравниваются их ссылочные значения, а не экземпляры, на которые они ссылаются.

Равенство строк

Два операнда string равны, если они оба имеют значение `null` или оба экземпляра строки имеют одинаковую длину и идентичные символы в каждой позиции символа.

```
string s1 = "hello!";
string s2 = "HeLLo!";
Console.WriteLine(s1 == s2.ToLower());  // output: True

string s3 = "Hello!";
Console.WriteLine(s1 == s3);  // output: False
```

Это порядковое сравнение, учитывающее регистр. Дополнительные сведения о том, как сравнивать строки, см. в статье Сравнение строк в C#.

Равенство делегатов

Два операнда делегатов одного типа среды выполнения равны, если оба из них имеют значение `null` или их списки вызовов имеют одинаковую длину и содержат одинаковые записи в каждой позиции:

```
Action a = () => Console.WriteLine("a");

Action b = a + a;
Action c = a + a;
Console.WriteLine(object.ReferenceEquals(b, c));  // output: False
Console.WriteLine(b == c);  // output: True
```

Подробные сведения см. в разделе Delegate equality operators (Операторы равенства делегатов) в спецификации языка C#.

Делегаты, созданные в результате оценки семантически идентичных лямбда-выражений не будут равны, как показано в примере ниже:

```
Action a = () => Console.WriteLine("a");
Action b = () => Console.WriteLine("a");

Console.WriteLine(a == b);  // output: False
Console.WriteLine(a + b == a + b);  // output: True
Console.WriteLine(b + a == a + b);  // output: False
```

Оператор неравенства !=

Оператор неравенства `!=` возвращает значение `true`, если его операнды не равны. В противном случае возвращается значение `false`. Для операндов встроенных типов выражение `x != y` дает тот же результат, что и выражение `!(x == y)`. Дополнительные сведения о равенстве типов см. в разделе Оператор равенства.

В следующем примере иллюстрируется использование оператора `!=`.

```
int a = 1 + 1 + 2 + 3;
int b = 6;
Console.WriteLine(a != b);  // output: True

string s1 = "Hello";
string s2 = "Hello";
Console.WriteLine(s1 != s2);  // output: False

object o1 = 1;
object o2 = 1;
Console.WriteLine(o1 != o2);  // output: True
```

Возможность перегрузки оператора

Определяемый пользователем тип может перегружать операторы `==` и `!=`. Если тип перегружает один из двух операторов, он должен также перегружать и другой.

Тип записи не может перегружать операторы `==` и `!=` явным образом. Если необходимо изменить поведение операторов `==` и `!=` для типа записи `T`, реализуйте метод IEquatable<T>.Equals со следующей сигнатурой.

```
public virtual bool Equals(T? other);
```

Спецификация языка C#

Дополнительные сведения см. в разделе Операторы отношения и проверки типа в статье по спецификации языка C#.

Дополнительные сведения о равенстве типов записей см. в разделе Элементы равенства предложения функции записей.

См. также

- справочник по C#
- Операторы и выражения C#
- System.IEquatable<T>
- Object.Equals
- Object.ReferenceEquals
- Сравнения на равенство
- Операторы сравнения

Операторы сравнения (справочник по C#)

18.12.2021 • 2 minutes to read

Операторы сравнения `<` (меньше чем), `>` (больше чем), `<=` (меньше или равно) и `>=` (больше или равно) (или реляционные операторы) сравнивают операнды. Эти операторы поддерживаются всеми целочисленными типами и типами с плавающей запятой.

> **NOTE**
> Если какой-то из операндов операторов `==`, `<`, `>`, `<=` и `>=` не является числом (Double.NaN или Single.NaN), результатом операции будет `false`. Это означает, что значение `NaN` не больше, не меньше и не равно любому другому значению `double` (или `float`), включая `NaN`. Дополнительные сведения и примеры см. в справочных статьях по Double.NaN или Single.NaN.

Тип char также поддерживает операторы сравнения. В случае операндов `char` сравниваются соответствующие коды символов.

Типы перечисления также поддерживают операторы сравнения. Если операнды имеют одинаковый тип enum, сравниваются соответствующие значения базового целочисленного типа.

Операторы `==` и `!=` проверяют равенство или неравенство своих операндов.

Оператор "меньше чем" <

Оператор `<` возвращает `true`, если его левый операнд меньше правого. В противном случае возвращается `false`:

```
Console.WriteLine(7.0 < 5.1);   // output: False
Console.WriteLine(5.1 < 5.1);   // output: False
Console.WriteLine(0.0 < 5.1);   // output: True

Console.WriteLine(double.NaN < 5.1);   // output: False
Console.WriteLine(double.NaN >= 5.1);  // output: False
```

Оператор "больше чем" >

Оператор `>` возвращает `true`, если его левый операнд больше правого. В противном случае возвращается `false`:

```
Console.WriteLine(7.0 > 5.1);   // output: True
Console.WriteLine(5.1 > 5.1);   // output: False
Console.WriteLine(0.0 > 5.1);   // output: False

Console.WriteLine(double.NaN > 5.1);   // output: False
Console.WriteLine(double.NaN <= 5.1);  // output: False
```

Оператор "меньше или равно" <=

Оператор `<=` возвращает `true`, если его левый операнд меньше правого или равен ему. В противном случае возвращается `false`:

```
Console.WriteLine(7.0 <= 5.1);   // output: False
Console.WriteLine(5.1 <= 5.1);   // output: True
Console.WriteLine(0.0 <= 5.1);   // output: True

Console.WriteLine(double.NaN > 5.1);   // output: False
Console.WriteLine(double.NaN <= 5.1);  // output: False
```

Оператор "больше или равно" >=

Оператор `>=` возвращает `true`, если его левый операнд больше правого или равен ему. В противном случае возвращается `false`:

```
Console.WriteLine(7.0 >= 5.1);   // output: True
Console.WriteLine(5.1 >= 5.1);   // output: True
Console.WriteLine(0.0 >= 5.1);   // output: False

Console.WriteLine(double.NaN < 5.1);   // output: False
Console.WriteLine(double.NaN >= 5.1);  // output: False
```

Возможность перегрузки оператора

Определяемый пользователем тип может перегружать операторы `<`, `>`, `<=` и `>=`.

Если тип перегружает один из операторов `<` и `>`, он должен также перегружать операторы `<` и `>`.
Если тип перегружает один из операторов `<=` и `>=`, он должен также перегружать операторы `<=` и `>=`
.

Спецификация языка C#

Дополнительные сведения см. в разделе Операторы отношения и проверки типа в статье по спецификации языка C#.

См. также

- справочник по C#
- Операторы и выражения C#
- System.IComparable<T>
- Операторы равенства

Операторы и выражения для доступа к элементам (справочник по C#)

18.12.2021 • 8 minutes to read

При получении доступа к элементу типа можно использовать следующие операторы и выражения:

- `.` (доступ к члену) : для доступа к члену пространства имен или типа;
- `[]` (элемент массива или индексатор доступа) : для доступа к элементу массива или индексатору типа;
- `?.` и `?[]` (null-условные операторы): для выполнения операции доступа члена или элемента только в том случае, если операнд не равен null;
- `()` (вызов) : для вызова метода или делегата.
- `^` (индекс от конца) : для определения того, что расположение элемента находится в конце последовательности.
- `..` (диапазон): для определения диапазона индексов, которые можно использовать для получения диапазона элементов последовательности.

Выражение доступа к члену

Маркер `.` используется для обращения к члену пространства имен или типа, как в следующих примерах.

- Используйте `.` для обращения к пространству имен, вложенному в другое пространство имен, как показано в следующем примере директивы `using`.

```
using System.Collections.Generic;
```

- Используйте `.` для создания *полного имени* для обращения к типу в пределах пространства имен, как показано в следующем коде:

```
System.Collections.Generic.IEnumerable<int> numbers = new int[] { 1, 2, 3 };
```

Используйте директиву `using`, чтобы сделать использование полных имен необязательным.

- Используйте `.` для обращения к членам типов (статическим и нестатическим), как показано в следующем коде:

```
var constants = new List<double>();
constants.Add(Math.PI);
constants.Add(Math.E);
Console.WriteLine($"{constants.Count} values to show:");
Console.WriteLine(string.Join(", ", constants));
// Output:
// 2 values to show:
// 3.14159265358979, 2.71828182845905
```

Можно также использовать `.` для вызова метода расширения.

Оператор индексатора []

Квадратные скобки, `[]`, обычно используются для доступа к элементам массива, индексатора или указателя.

Доступ к массиву

В приведенном ниже примере показано, как получить доступ к элементам массива.

```
int[] fib = new int[10];
fib[0] = fib[1] = 1;
for (int i = 2; i < fib.Length; i++)
{
    fib[i] = fib[i - 1] + fib[i - 2];
}
Console.WriteLine(fib[fib.Length - 1]);  // output: 55

double[,] matrix = new double[2,2];
matrix[0,0] = 1.0;
matrix[0,1] = 2.0;
matrix[1,0] = matrix[1,1] = 3.0;
var determinant = matrix[0,0] * matrix[1,1] - matrix[1,0] * matrix[0,1];
Console.WriteLine(determinant);  // output: -3
```

Если индекс массива выходит за границы соответствующего измерения массива, возникает исключение IndexOutOfRangeException.

Как показано в предыдущем примере, квадратные скобки также используются в объявлении типа массива и для создания экземпляров массива.

Дополнительные сведения см. в руководстве по работе с массивами.

Доступ к индексатору

В приведенном ниже примере используется тип .NET Dictionary<TKey,TValue> для получения доступа к индексатору:

```
var dict = new Dictionary<string, double>();
dict["one"] = 1;
dict["pi"] = Math.PI;
Console.WriteLine(dict["one"] + dict["pi"]);  // output: 4.14159265358979
```

Индексаторы позволяют индексировать экземпляры определяемого пользователем типа аналогично индексации массива. В отличие от индексов массива, которые должны быть целым числом, параметры индексатора могут быть объявлены любым типом.

Дополнительные сведения см. в руководстве по работе с индексаторами.

Другие данные об использовании []

Сведения о доступе к элементу указателя см. в разделе, посвященном оператору доступа к элементу указателя [], статьи Операторы, связанные с указателем.

Кроме того, с помощью квадратных скобок можно указывать атрибуты.

```
[System.Diagnostics.Conditional("DEBUG")]
void TraceMethod() {}
```

NULL-условные операторы: ?. и ?[]

В C# 6 и более поздних версий доступен оператор с условием null, который применяется для доступа к членам, `?.`, или доступа к элементам, `?[]`, к операнду только в том случае, если он имеет значение,

отличное от NULL, в противном случае он возвращает `null`. Это означает следующее:

- Если `a` вычисляется как `null`, то результатом `a?.x` или `a?[x]` является `null`.
- Если `a` принимает значение, отличное от NULL, результат `a?.x` или `a?[x]` совпадает с результатом `a.x` или `a[x]` соответственно.

> **NOTE**
> Если `a.x` или `a[x]` вызывает исключение, `a?.x` или `a?[x]` вызовут то же исключение для отличного от NULL `a`. Например, если `a` является экземпляром массива, не равным null, и `x` находится вне границ `a`, `a?[x]` вызовет IndexOutOfRangeException.

Операторы с условием NULL предусматривают сокращенную обработку. То есть, если в цепочке операций условного доступа к элементу или члену одна из операций возвращает значение `null`, остальная цепочка не выполняется. В следующем примере `B` не вычисляется, если `A` принимает значение `null`, и `C` не вычисляется, если `A` или `B` принимает значение `null`.

```
A?.B?.Do(C);
A?.B?[C];
```

Если параметр `A` может быть пустым, а `B` и `C` нет, если A не является пустым, необходимо только применить условный оператор NULL к параметру `A`:

```
A?.B.C();
```

В предыдущем примере `B` не вычисляется, а `C()` не вызывается, если `A` имеет значение NULL. Однако при прерывании доступа к связанному члену, например, в круглых скобках, как в `(A?.B).C()`, не происходит сокращенное вычисление.

В следующем примере иллюстрируется использование операторов `?.` и `?[]`:

```
double SumNumbers(List<double[]> setsOfNumbers, int indexOfSetToSum)
{
    return setsOfNumbers?[indexOfSetToSum]?.Sum() ?? double.NaN;
}

var sum1 = SumNumbers(null, 0);
Console.WriteLine(sum1);  // output: NaN

var numberSets = new List<double[]>
{
    new[] { 1.0, 2.0, 3.0 },
    null
};

var sum2 = SumNumbers(numberSets, 0);
Console.WriteLine(sum2);  // output: 6

var sum3 = SumNumbers(numberSets, 1);
Console.WriteLine(sum3);  // output: NaN
```

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace MemberAccessOperators2
{
    public static class NullConditionalShortCircuiting
    {
        public static void Main()
        {
            Person person = null;
            person?.Name.Write(); // no output: Write() is not called due to short-circuit.
            try
            {
                (person?.Name).Write();
            }
            catch (NullReferenceException)
            {
                Console.WriteLine("NullReferenceException");
            }; // output: NullReferenceException
        }
    }

    public class Person
    {
        public FullName Name { get; set; }
    }

    public class FullName
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public void Write()
        {
            Console.WriteLine($"{FirstName} {LastName}");
        }
    }
}
```

В первом из двух приведенных выше примеров также используется оператор объединения со значением NULL `??`, что позволяет указать альтернативное выражение для вычисления в случае, если результат выполнения условной операции NULL — это `null`.

Если `a.x` или `a[x]` является типом `T`, не допускающим значение NULL, `a?.x` или `a?[x]` является соответствующим типом `T?`, допускающим значение NULL. Если требуется выражение типа `T`, примените оператор объединения со значением NULL `??` к условному выражению NULL, как показано в следующем примере:

```
int GetSumOfFirstTwoOrDefault(int[] numbers)
{
    if ((numbers?.Length ?? 0) < 2)
    {
        return 0;
    }
    return numbers[0] + numbers[1];
}

Console.WriteLine(GetSumOfFirstTwoOrDefault(null));  // output: 0
Console.WriteLine(GetSumOfFirstTwoOrDefault(new int[0]));  // output: 0
Console.WriteLine(GetSumOfFirstTwoOrDefault(new[] { 3, 4, 5 }));  // output: 7
```

Если в предыдущем примере оператор `??` не используется, `numbers?.Length < 2` вычисляется как `false`,

если `numbers` имеет значение `null`.

Null-условный оператор доступа к элементу `?.` также называется элвис-оператором.

Потокобезопасный вызов делегата

Используйте оператор `?.` для проверки того, что делегат не равен null, и его вызова потокобезопасным способом (например, в том случае, когда вы собираетесь породить событие), как показано в следующем коде:

```
PropertyChanged?.Invoke(…)
```

Этот код эквивалентен следующему коду, который будет использоваться в C# 5 или более ранней версии:

```
var handler = this.PropertyChanged;
if (handler != null)
{
    handler(…);
}
```

Этот потокобезопасный способ гарантирует, что вызывается только `handler`, не имеющий значения NULL. Так как экземпляры делегата являются неизменяемыми, ни один из потоков не может изменить объект, на который ссылается локальная переменная `handler`. В частности, если код, выполняемый другим потоком, отменяет подписку на событие `PropertyChanged` и событие `PropertyChanged` принимает значение `null` до вызова `handler`, объект, на который ссылается `handler`, остается неизменным. Оператор `?.` вычисляет левый операнд не более одного раза, гарантируя, что его нельзя изменить на `null` после того, как он пройдет проверку как не имеющий значение NULL.

Выражения вызова ()

Используйте скобки, `()`, чтобы вызвать метод или делегат.

Приведенный ниже пример демонстрирует вызов делегата и метода с аргументами или без них.

```
Action<int> display = s => Console.WriteLine(s);

var numbers = new List<int>();
numbers.Add(10);
numbers.Add(17);
display(numbers.Count);   // output: 2

numbers.Clear();
display(numbers.Count);   // output: 0
```

Круглые скобки также можно использовать при вызове конструктора с оператором `new`.

Другие данные об использовании ()

Кроме того, с помощью круглых скобок можно настраивать порядок выполнения операций в выражении. Дополнительные сведения см. в разделе Операторы C#.

В выражениях приведения, которые выполняют явные преобразования типов, также используйте круглые скобки.

Индекс от конца: оператор ^

Оператор `^`, доступный в C# 8.0 и последующих версиях, определяет расположение элемента от конца последовательности. Для последовательности длины `length``^n` указывает на элемент с `length - n` смещения от начала последовательности. Например, `^1` указывает на последний элемент последовательности, а `^length` — на первый элемент последовательности.

```
int[] xs = new[] { 0, 10, 20, 30, 40 };
int last = xs[^1];
Console.WriteLine(last);  // output: 40

var lines = new List<string> { "one", "two", "three", "four" };
string prelast = lines[^2];
Console.WriteLine(prelast);  // output: three

string word = "Twenty";
Index toFirst = ^word.Length;
char first = word[toFirst];
Console.WriteLine(first);  // output: T
```

В предыдущем примере выражение `^e` имеет тип System.Index. В выражении `^e` результат `e` должен быть неявно преобразован в `int`.

Можно также использовать оператор `^` с оператором диапазона для создания диапазона индексов. См. сведения в руководстве по диапазонам и индексам.

Диапазон: оператор .

Оператор диапазона `..`, доступный в C# 8.0 и последующих версиях, определяет начало и конец диапазона индексов в качестве своих операндов. Левый операнд является *инклюзивным* началом диапазона. Правый операнд является *эксклюзивным* концом диапазона. Любой из операндов может быть индексом от начала или конца последовательности, как показано в следующем примере:

```
int[] numbers = new[] { 0, 10, 20, 30, 40, 50 };
int start = 1;
int amountToTake = 3;
int[] subset = numbers[start..(start + amountToTake)];
Display(subset);  // output: 10 20 30

int margin = 1;
int[] inner = numbers[margin..^margin];
Display(inner);  // output: 10 20 30 40

string line = "one two three";
int amountToTakeFromEnd = 5;
Range endIndices = ^amountToTakeFromEnd..^0;
string end = line[endIndices];
Console.WriteLine(end);  // output: three

void Display<T>(IEnumerable<T> xs) => Console.WriteLine(string.Join(" ", xs));
```

В предыдущем примере выражение `a..b` имеет тип System.Range. В выражении `a..b` результаты `a` и `b` должны быть неявно преобразованы в `int` или Index.

Можно проигнорировать любой из операндов оператора `..`, чтобы получить открытый диапазон:

- `a..` — это эквивалент `a..^0`
- `..b` — это эквивалент `0..b`
- `..` — это эквивалент `0..^0`

```
int[] numbers = new[] { 0, 10, 20, 30, 40, 50 };
int amountToDrop = numbers.Length / 2;

int[] rightHalf = numbers[amountToDrop..];
Display(rightHalf);  // output: 30 40 50

int[] leftHalf = numbers[..^amountToDrop];
Display(leftHalf);  // output: 0 10 20

int[] all = numbers[..];
Display(all);  // output: 0 10 20 30 40 50

void Display<T>(IEnumerable<T> xs) => Console.WriteLine(string.Join(" ", xs));
```

См. сведения в руководстве по диапазонам и индексам.

Возможность перегрузки оператора

Операторы `.`, `()`, `^` и `..` перегрузить нельзя. Оператор `[]` также считается неперегружаемым. Используйте индексаторы для поддержки индексирования с помощью определяемых пользователем типов.

Спецификация языка C#

Дополнительные сведения см. в следующих разделах статьи Спецификация языка C#:

- Доступ к членам
- Доступ к элементам
- Null-условный оператор
- Выражения вызова

См. сведения о индексах и диапазонах в примечании к функциям.

См. также

- справочник по C#
- Операторы и выражения C#
- Оператор ?? (оператор объединения со значением NULL)
- Оператор ::

Операторы проверки типа и выражения приведения (справочник по C#)

18.12.2021 • 5 minutes to read

Чтобы выполнить проверку или преобразование типов, можно использовать следующие операторы и выражения:

- оператор is проверяет, совместим ли тип среды выполнения определенного выражения с указанным типом;
- оператор as явным образом преобразует выражение в указанный тип, если тип среды выполнения совместим с этим типом;
- выражение приведения выполняет явное преобразование;
- оператор typeof получает экземпляр System.Type указанного типа.

Оператор is

Оператор `is` проверяет, совместим ли тип среды выполнения результата определенного выражения с указанным типом. Начиная с версии C# 7.0, оператор `is` также проверяет соответствие результата выражения указанному шаблону.

Выражение с оператором проверки типа `is` имеет следующий вид:

```
E is T
```

где `E` представляет выражение, возвращающее значение, а `T` содержит имя или параметр типа. `E` не может использоваться как анонимный метод или лямбда-выражение.

Оператор `is` возвращает `true`, если результат выражения имеет значение, отличное от NULL, и выполняется любое из следующих условий.

- Тип результата выражения в среде выполнения — `T`.
- Тип результата выражения в среде выполнения является производным от типа `T`, реализует интерфейс `T` или существует другое неявное преобразования ссылок из него в `T`.
- Тип результата выражения в среде выполнения является типом, допускающим значение NULL, имеющим базовый тип `T`, а Nullable<T>.HasValue — `true`.
- Существует упаковка-преобразование или распаковка-преобразование из типа результата выражения в среде выполнения в тип `T`.

Оператор `is` не учитывает заданные пользователем преобразования.

В следующем примере показано, что оператор `is` возвращает `true`, если тип результата выражения в среде выполнения является производным от указанного типа, то есть между этими типами существует преобразование ссылки:

```
public class Base { }

public class Derived : Base { }

public static class IsOperatorExample
{
    public static void Main()
    {
        object b = new Base();
        Console.WriteLine(b is Base);  // output: True
        Console.WriteLine(b is Derived);  // output: False

        object d = new Derived();
        Console.WriteLine(d is Base);  // output: True
        Console.WriteLine(d is Derived); // output: True
    }
}
```

В следующем примере показано, что оператор `is` учитывает упаковку-преобразование и распаковку-преобразование, но не учитывает числовые преобразования:

```
int i = 27;
Console.WriteLine(i is System.IFormattable);  // output: True

object iBoxed = i;
Console.WriteLine(iBoxed is int);  // output: True
Console.WriteLine(iBoxed is long);  // output: False
```

Дополнительные сведения о преобразованиях в C# см. в главе о преобразованиях в спецификации по языку C#.

Тестирование типов с сопоставлением шаблонов

Начиная с версии C# 7.0, оператор `is` также проверяет соответствие результата выражения указанному шаблону. В следующем примере показано, как использовать шаблон объявления для проверки типа выражения в среде выполнения:

```
int i = 23;
object iBoxed = i;
int? jNullable = 7;
if (iBoxed is int a && jNullable is int b)
{
    Console.WriteLine(a + b);  // output 30
}
```

Дополнительные сведения о поддерживаемых шаблонах см. в разделе Шаблоны.

Оператор as

Оператор `as` явным образом преобразует результат выражения в указанный ссылочный или поддерживающий значения NULL тип. Если такое преобразование невозможно, оператор `as` возвращает значение `null`. В отличие от выражения приведения, оператор `as` никогда не создает исключение.

Выражение имеет такой формат:

```
E as T
```

где `E` представляет выражение, возвращающее значение, а `T` содержит имя или параметр типа. Результат такого выражения аналогичен результату этого:

```
E is T ? (T)(E) : (T)null
```

за исключением того, что `E` вычисляется только один раз.

Оператор `as` рассматривает только преобразование ссылки, допускающие значение NULL преобразования, упаковку-преобразование и распаковку-преобразование. Оператор `as` нельзя использовать для определенного пользователем преобразования. Для этого используйте выражение приведения.

В следующем примере иллюстрируется использование оператора `as`.

```
IEnumerable<int> numbers = new[] { 10, 20, 30 };
IList<int> indexable = numbers as IList<int>;
if (indexable != null)
{
    Console.WriteLine(indexable[0] + indexable[indexable.Count - 1]);  // output: 40
}
```

NOTE

Как показано в примере выше, нужно сравнить результат выражения `as` со значением `null`, чтобы проверить, успешно ли выполнено преобразование. Начиная с версии C# 7.0, вы можете использовать оператор is как для проверки успешного выполнения преобразования, так и для сохранения результата в переменной, если преобразование выполнено успешно.

Выражение приведения

Выражение приведения в формате `(T)E` выполняет явное преобразование значения выражения `E` в тип `T`. Если явного преобразования из типа `E` в тип `T` не существует, возникает ошибка времени компиляции. Во время выполнения явное преобразование может завершиться сбоем, и выражение приведения может вызвать исключение.

Приведенный ниже пример демонстрирует явное числовое преобразование и преобразование ссылки:

```
double x = 1234.7;
int a = (int)x;
Console.WriteLine(a);   // output: 1234

IEnumerable<int> numbers = new int[] { 10, 20, 30 };
IList<int> list = (IList<int>)numbers;
Console.WriteLine(list.Count);  // output: 3
Console.WriteLine(list[1]);  // output: 20
```

Сведения о поддерживаемых явных преобразованиях см. в разделе о явных преобразованиях в спецификации по языку C#. Сведения о том, как определять пользовательские операторы явного или неявного преобразования, см. в разделе Операторы пользовательского преобразования.

Другие данные об использовании ()

Используйте скобки, чтобы вызвать метод или делегат.

Кроме того, с помощью круглых скобок можно настраивать порядок выполнения операций в выражении. Дополнительные сведения см. в разделе Операторы C#.

typeof - оператор

Оператор `typeof` получает экземпляр System.Type для указанного типа. Оператор `typeof` принимает в качестве аргумента имя типа или параметр типа, как показано в следующем примере:

```
void PrintType<T>() => Console.WriteLine(typeof(T));

Console.WriteLine(typeof(List<string>));
PrintType<int>();
PrintType<System.Int32>();
PrintType<Dictionary<int, char>>();
// Output:
// System.Collections.Generic.List`1[System.String]
// System.Int32
// System.Int32
// System.Collections.Generic.Dictionary`2[System.Int32,System.Char]
```

Аргумент не должны должен быть типом, требующим заметок к метаданным. Далее приведены примеры типов.

- `dynamic`
- `string?` (или любой ссылочный тип, допускающий значение NULL)

Эти типы не представлены непосредственно в метаданных. Типы включают в себя атрибуты, описывающие базовый тип. В обоих случаях можно использовать базовый тип. Вместо `dynamic` можно использовать `object`. Вместо `string?` можно использовать `string`.

Кроме того, оператор `typeof` можно использовать с несвязанными универсальными типами. В имени несвязанного универсального типа должно содержаться правильное количество запятых, то есть на одну меньше, чем число параметров этого типа. В следующем примере показано использование оператора `typeof` с несвязанным универсальным типом:

```
Console.WriteLine(typeof(Dictionary<,>));
// Output:
// System.Collections.Generic.Dictionary`2[TKey,TValue]
```

Оператор `typeof` не может принимать выражение в качестве аргумента. Чтобы получить экземпляр System.Type для типа среды выполнения результата выражения, используйте метод Object.GetType.

Тестирование типов с оператором `typeof`

Используйте оператор `typeof`, чтобы проверить совместимость типа среды выполнения результата выражения с указанным типом. В следующем примере показано различие между проверкой типов, выполненной с помощью оператора `typeof` и оператора is:

```
public class Animal { }

public class Giraffe : Animal { }

public static class TypeOfExample
{
    public static void Main()
    {
        object b = new Giraffe();
        Console.WriteLine(b is Animal);  // output: True
        Console.WriteLine(b.GetType() == typeof(Animal));  // output: False

        Console.WriteLine(b is Giraffe);  // output: True
        Console.WriteLine(b.GetType() == typeof(Giraffe));  // output: True
    }
}
```

Возможность перегрузки оператора

Операторы `is`, `as` и `typeof` перегрузить нельзя.

Определяемый пользователем тип нельзя использовать для перегрузки оператора `()`, но на его основе можно определить пользовательские преобразования типа и выполнить их с помощью выражения приведения. Дополнительные сведения см. в разделе Операторы пользовательского преобразования.

Спецификация языка C#

Дополнительные сведения см. в следующих разделах статьи Спецификация языка C#:

- Оператор is
- Оператор as
- Выражения приведения
- Оператор typeof

См. также

- справочник по C#
- Операторы и выражения C#
- Практическое руководство. Безопасное приведение с помощью сопоставления шаблонов, а также операторов is и as
- Универсальные шаблоны в .NET

Операторы пользовательского преобразования (справочник по C#)

18.12.2021 • 2 minutes to read

Пользовательский тип может определять неявное или явное пользовательское преобразование в другой тип или из него.

Неявные преобразования не требуют специального синтаксиса для вызова и могут происходить в различных ситуациях, например при вызовах методов и в назначениях. Предопределенные неявные преобразования на C# всегда завершаются успешно и никогда не вызывают исключение. Неявные пользовательские преобразования должны вести себя таким же образом. Если пользовательское преобразование может вызвать исключение или привести к потере данных, определите его как явное преобразование.

Пользовательские преобразования не рассматриваются операторами is и as. Используйте выражение приведения для вызова явного пользовательского преобразования.

Используйте ключевые слова `operator` и `implicit` или `explicit` для определения явного или неявного преобразования соответственно. Тип, который определяет преобразование, должен быть типом источника или целевого объекта этого преобразования. Преобразование между двумя пользовательскими типами можно определить одним из двух типов.

Следующий пример иллюстрирует, как определить неявное и явное преобразования:

```
using System;

public readonly struct Digit
{
    private readonly byte digit;

    public Digit(byte digit)
    {
        if (digit > 9)
        {
            throw new ArgumentOutOfRangeException(nameof(digit), "Digit cannot be greater than nine.");
        }
        this.digit = digit;
    }

    public static implicit operator byte(Digit d) => d.digit;
    public static explicit operator Digit(byte b) => new Digit(b);

    public override string ToString() => $"{digit}";
}

public static class UserDefinedConversions
{
    public static void Main()
    {
        var d = new Digit(7);

        byte number = d;
        Console.WriteLine(number);  // output: 7

        Digit digit = (Digit)number;
        Console.WriteLine(digit);  // output: 7
    }
}
```

Можно также использовать ключевое слово `operator` для перегрузки предопределенного оператора C#. Для получения дополнительной информации см. раздел Перегрузка операторов.

Спецификация языка C#

Дополнительные сведения см. в следующих разделах статьи Спецификация языка C#:

- Операторы преобразования
- Пользовательские преобразования
- Неявные преобразования
- Явные преобразования

См. также

- справочник по C#
- Операторы и выражения C#
- Перегрузка операторов
- Операторы приведения и тестирования типов
- Приведение и преобразование типов
- Рекомендации по разработке. Операторы преобразования
- Связанные пользовательские явные преобразования в C#

Операторы, связанные с указателем (справочник по C#)

18.12.2021 • 6 minutes to read

Для работы с указателями можно использовать следующие операторы:

- Унарный оператор `&` (address-of): для получения адреса переменной
- Унарный оператор `*` (косвенное обращение к указателю): для получения переменной, на которую указывает указатель
- Операторы `->` (доступ к членам) и `[]` (доступ к элементам)
- Арифметические операторы `+`, `-`, `++` и `--`
- Операторы сравнения `==`, `!=`, `<`, `>`, `<=` и `>=`

Сведения о типах указателей см. в разделе Типы указателей.

> **NOTE**
> Для всех операций с указателями требуется небезопасный контекст. Код, содержащий небезопасные блоки, должен компилироваться с параметром компилятора **AllowUnsafeBlocks**.

Оператор address-of &

Унарный оператор `&` возвращает адрес своего операнда:

```
unsafe
{
    int number = 27;
    int* pointerToNumber = &number;

    Console.WriteLine($"Value of the variable: {number}");
    Console.WriteLine($"Address of the variable: {(long)pointerToNumber:X}");
}
// Output is similar to:
// Value of the variable: 27
// Address of the variable: 6C1457DBD4
```

Операнд оператора `&` должен быть фиксированной переменной. *Фиксированные* переменные — это переменные в местах хранения, на которые не распространяется сборка мусора. В приведенном выше примере локальная переменная `number` — это фиксированная переменная, так как она находится в стеке. Переменные, которые находятся в местах хранения, обслуживаемых сборщиком мусора (например, перемещение), называются *перемещаемыми* переменными. Поля объекта и элементы массива являются примерами перемещаемых переменных. Вы можете получить адрес перемещаемой переменной, если зафиксируете или закрепите ее с помощью инструкции `fixed`. Полученный адрес действителен только в пределах блока инструкции `fixed`. В следующем примере показано, как использовать инструкцию `fixed` и оператор `&`:

```
unsafe
{
    byte[] bytes = { 1, 2, 3 };
    fixed (byte* pointerToFirst = &bytes[0])
    {
        // The address stored in pointerToFirst
        // is valid only inside this fixed statement block.
    }
}
```

Получить адрес константы или значения нельзя.

Дополнительные сведения о фиксированных и перемещаемых переменных см. в разделе Фиксированные и перемещаемые переменные в спецификации языка C#.

Бинарный оператор `&` вычисляет логическое И своих логических операндов или побитовое логическое И своих целочисленных операндов.

Оператор косвенного обращения указателя *

Унарный оператор косвенного обращения указателя `*` получает переменную, на которую указывает операнд. Он также называется оператором разыменования. Операнд оператора `*` должен иметь тип указателя.

```
unsafe
{
    char letter = 'A';
    char* pointerToLetter = &letter;
    Console.WriteLine($"Value of the `letter` variable: {letter}");
    Console.WriteLine($"Address of the `letter` variable: {(long)pointerToLetter:X}");

    *pointerToLetter = 'Z';
    Console.WriteLine($"Value of the `letter` variable after update: {letter}");
}
// Output is similar to:
// Value of the `letter` variable: A
// Address of the `letter` variable: DCB977DDF4
// Value of the `letter` variable after update: Z
```

Невозможно применить оператор `*` к выражению типа `void*`.

Бинарный оператор `*` вычисляет продукт своих числовых операндов.

Оператор доступа к члену указателя ->

Оператор `->` объединяет косвенное обращение к указателю и доступ к члену. То есть если `x` — это указатель типа `T*`, а `y` является доступным членом типа `T`, выражение формы

```
x->y
```

эквивалентно

```
(*x).y
```

В следующем примере иллюстрируется использование оператора `->`.

```
public struct Coords
{
    public int X;
    public int Y;
    public override string ToString() => $"({X}, {Y})";
}

public class PointerMemberAccessExample
{
    public static unsafe void Main()
    {
        Coords coords;
        Coords* p = &coords;
        p->X = 3;
        p->Y = 4;
        Console.WriteLine(p->ToString());  // output: (3, 4)
    }
}
```

Невозможно применить оператор `->` к выражению типа `void*`.

Оператор доступа к элементу указателя []

Для выражения `p` типа указателя доступ к элементу указателя формы `p[n]` вычисляется как `*(p + n)`, где `n` должен иметь тип, неявно преобразуемый в `int`, `uint`, `long` или `ulong`. Сведения о поведении оператора `+` с указателями см. в разделе Сложение целочисленного значения с указателем или его вычитание из указателя.

Следующий пример демонстрирует доступ к элементам массива с указателем и оператором `[]`:

```
unsafe
{
    char* pointerToChars = stackalloc char[123];

    for (int i = 65; i < 123; i++)
    {
        pointerToChars[i] = (char)i;
    }

    Console.Write("Uppercase letters: ");
    for (int i = 65; i < 91; i++)
    {
        Console.Write(pointerToChars[i]);
    }
}
// Output:
// Uppercase letters: ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

В предыдущем примере выражение `stackalloc` выделяет блок памяти в стеке.

> **NOTE**
> Оператор доступа к элементу указателя не проверяет ошибки за пределами области.

Нельзя использовать `[]` для доступа к элементу указателя с помощью выражения типа `void*`.

Можно также использовать оператор `[]` для доступа к элементу массива или индексатору.

Арифметические операторы указателя

Вы можете выполнить следующие арифметические операции с указателями:

- Сложение целочисленного значения с указателем или его вычитание из указателя
- Вычитание двух указателей
- Инкремент или декремент указателя

Невозможно выполнить эти операции с указателями типа `void*`.

Сведения о поддерживаемых арифметических операциях с числовыми типами см. в разделе Арифметические операторы.

Сложение целочисленного значения с указателем или его вычитание из указателя

Для указателя `p` типа `T*` и выражения `n` типа, неявно преобразуемого в `int`, `uint`, `long` или `ulong`, сложение и вычитание определяются следующим образом:

- Выражения `p + n` и `n + p` создают указатель типа `T*`, полученный добавлением `n * sizeof(T)` к адресу, предоставленному `p`.
- Выражение `p - n` создает указатель типа `T*`, полученный вычитанием `n * sizeof(T)` из адреса, предоставленного `p`.

Оператор `sizeof` получает размер типа в байтах.

В следующем примере показано использование оператора `+` с указателем:

```
unsafe
{
    const int Count = 3;
    int[] numbers = new int[Count] { 10, 20, 30 };
    fixed (int* pointerToFirst = &numbers[0])
    {
        int* pointerToLast = pointerToFirst + (Count - 1);

        Console.WriteLine($"Value {*pointerToFirst} at address {(long)pointerToFirst}");
        Console.WriteLine($"Value {*pointerToLast} at address {(long)pointerToLast}");
    }
}
// Output is similar to:
// Value 10 at address 1818345918136
// Value 30 at address 1818345918144
```

Вычитание указателей

Для двух указателей `p1` и `p2` типа `T*` выражение `p1 - p2` находит разность между адресами, предоставленными `p1` и `p2`, деленную на `sizeof(T)`. Результат имеет тип `long`. То есть `p1 - p2` вычисляется как `((long)(p1) - (long)(p2)) / sizeof(T)`.

В следующем примере показано вычитание указателей:

```
unsafe
{
    int* numbers = stackalloc int[] { 0, 1, 2, 3, 4, 5 };
    int* p1 = &numbers[1];
    int* p2 = &numbers[5];
    Console.WriteLine(p2 - p1);  // output: 4
}
```

Инкремент и декремент указателя

Оператор инкремента `++` добавляет 1 к своему операнду указателя. Оператор декремента `--` вычитает 1 из своего операнда указателя.

Оба оператора поддерживаются в двух формах: постфикс (`p++` и `p--`) и префикс (`++p` и `--p`). Результат `p++` и `p--` — это значение `p` *перед* операцией. Результат `++p` и `--p` — это значение `p` *после* операции.

В следующем примере показано поведение постфиксных и префиксных операторов инкремента:

```
unsafe
{
    int* numbers = stackalloc int[] { 0, 1, 2 };
    int* p1 = &numbers[0];
    int* p2 = p1;
    Console.WriteLine($"Before operation: p1 - {(long)p1}, p2 - {(long)p2}");
    Console.WriteLine($"Postfix increment of p1: {(long)(p1++)}");
    Console.WriteLine($"Prefix increment of p2: {(long)(++p2)}");
    Console.WriteLine($"After operation: p1 - {(long)p1}, p2 - {(long)p2}");
}
// Output is similar to
// Before operation: p1 - 816489946512, p2 - 816489946512
// Postfix increment of p1: 816489946512
// Prefix increment of p2: 816489946516
// After operation: p1 - 816489946516, p2 - 816489946516
```

Операторы сравнения указателей

Вы можете использовать операторы `==` , `!=` , `<` , `>` , `<=` и `>=` для сравнения операндов любого типа указателя, включая `void*` . Эти операторы сравнивают адреса, предоставленные двумя операндами, как если бы они были целыми числами без знака.

Сведения о поведении этих операторов для операндов других типов см. в статьях Операторы равенства и Операторы сравнения.

Приоритет операторов

В следующем списке операторы, связанные с указателями, упорядочены от самого высокого до самого низкого приоритета:

- Постфиксные операторы инкремента `x++` и декремента `x--` и операторы `->` и `[]`
- Префиксные операторы инкремента `++x` и декремента `--x` и операторы `&` и `*`
- Аддитивные операторы `+` и `-`
- Операторы сравнения `<` , `>` , `<=` и `>=`
- Операторы равенства `==` и `!=`

Используйте скобки `()` , чтобы изменить порядок вычисления, накладываемый приоритетом операторов.

Полный список операторов C#, упорядоченный по уровню приоритета, можно найти в разделе Приоритет операторов статьи Операторы C#.

Возможность перегрузки оператора

Определяемый пользователем тип не может перегружать операторы, связанные с указателем, `&` , `*` , `->` и `[]` .

Спецификация языка C#

Дополнительные сведения см. в следующих разделах статьи Спецификация языка C#:

- Фиксированные и перемещаемые переменные
- Оператор address-of
- Косвенное обращение к указателю
- Доступ к членам указателей
- Доступ к элементам указателей
- Расчеты указателей
- Инкремент и декремент указателя
- Сравнение указателей

См. также

- справочник по C#
- Операторы и выражения C#
- Типы указателей
- Ключевое слово unsafe
- Ключевое слово fixed
- stackalloc
- Оператор sizeof

Справочник по C#. Операторы присваивания

18.12.2021 • 2 minutes to read

Оператор присваивания `=` назначает значение расположенного в правой части операнда переменной, свойству или элементу индексатора, которые указаны в операнде слева. Результатом выражения присваивания является значение, назначенное расположенному слева операнду. Расположенный справа операнд должен иметь тот же тип, что и операнд слева, либо же допускать неявное преобразование в этот тип.

Оператор присваивания `=` имеет правую ассоциативность, то есть выражение формы:

```
a = b = c
```

вычисляется как

```
a = (b = c)
```

В следующем примере показано использование оператора присваивания с локальной переменной, свойством и элементом индексатора в качестве левого операнда:

```
var numbers = new List<double>() { 1.0, 2.0, 3.0 };

Console.WriteLine(numbers.Capacity);
numbers.Capacity = 100;
Console.WriteLine(numbers.Capacity);
// Output:
// 4
// 100

int newFirstElement;
double originalFirstElement = numbers[0];
newFirstElement = 5;
numbers[0] = newFirstElement;
Console.WriteLine(originalFirstElement);
Console.WriteLine(numbers[0]);
// Output:
// 1
// 5
```

Ссылочный оператор присваивания

Начиная с C# 7.3, вы можете использовать ссылочный оператор присваивания `= ref`, чтобы переназначить ссылочную локальную переменную или ссылочную локальную переменную только для чтения. В следующем примере иллюстрируется использование ссылочного оператора присваивания:

```
void Display(double[] s) => Console.WriteLine(string.Join(" ", s));

double[] arr = { 0.0, 0.0, 0.0 };
Display(arr);

ref double arrayElement = ref arr[0];
arrayElement = 3.0;
Display(arr);

arrayElement = ref arr[arr.Length - 1];
arrayElement = 5.0;
Display(arr);
// Output:
// 0 0 0
// 3 0 0
// 3 0 5
```

При использовании ссылочного оператора присваивания тип левого и правого операндов должен быть одинаковым.

Составное присваивание

Для бинарного оператора `op` выражение составного присваивания в форме

```
x op= y
```

эквивалентно

```
x = x op y
```

за исключением того, что `x` вычисляется только один раз.

Составное присваивание поддерживается арифметическими, логическими, побитовыми логическими операторами и операторами смещения.

Присваивание объединения со значением NULL

Начиная с C# 8.0 вы можете использовать оператор присваивания объединения со значением NULL `??=` для присваивания значения правого операнда левому операнду только в том случае, если левый операнд принимает значение `null`. Дополнительные сведения см. в статье Операторы ?? и ??=.

Возможность перегрузки оператора

Пользовательский тип не может перегружать оператор присваивания. Однако пользовательский тип может определять неявное преобразование в другой тип. Таким образом, значение пользовательского типа может быть присвоено переменной, свойству или элементу индексатора другого типа. Дополнительные сведения см. в разделе Операторы пользовательского преобразования.

Определяемый пользователем тип не может перегружать оператор составного присваивания явным образом. Но если пользовательский тип перегружает бинарный оператор `op`, существующий оператор `op=` также будет неявно перегружен.

Спецификация языка C#

Подробные сведения см. в разделе Assignment operators (Операторы присваивания) в спецификации

языка C#.

См. сведения о ссылочном операторе присваивания `= ref` в примечании к функциям.

См. также раздел

- справочник по C#
- Операторы и выражения C#
- Ключевое слово ref

Лямбда-выражения (Справочник по C#)

18.12.2021 • 12 minutes to read

Лямбда-выражение используется для создания анонимной функции. Используйте оператор объявления лямбда-выражения `=>` для отделения списка параметров лямбда-выражения от исполняемого кода. Лямбда-выражение может иметь одну из двух следующих форм:

- Лямбда выражения, имеющая выражение в качестве текста:

```
(input-parameters) => expression
```

- Лямбда оператора, имеющая блок операторов в качестве текста:

```
(input-parameters) => { <sequence-of-statements> }
```

Чтобы создать лямбда-выражение, необходимо указать входные параметры (если они есть) с левой стороны лямбда-оператора и блок выражений или операторов с другой стороны.

Лямбда-выражение может быть преобразовано в тип делегата. Тип делегата, в который может быть преобразовано лямбда-выражение, определяется типами его параметров и возвращаемым значением. Если лямбда-выражение не возвращает значение, оно может быть преобразовано в один из типов делегата `Action`; в противном случае его можно преобразовать в один из типов делегатов `Func`. Например, лямбда-выражение, которое имеет два параметра и не возвращает значение, можно преобразовать в делегат Action<T1,T2>. Лямбда-выражение, которое имеет два параметра и возвращает значение, можно преобразовать в делегат Func<T,TResult>. В следующем примере лямбда-выражение `x => x * x`, которое указывает параметр с именем `x` и возвращает значение `x` в квадрате, присваивается переменной типа делегата:

```
Func<int, int> square = x => x * x;
Console.WriteLine(square(5));
// Output:
// 25
```

Лямбда-выражения можно также преобразовать в типы дерева выражения, как показано в следующем примере:

```
System.Linq.Expressions.Expression<Func<int, int>> e = x => x * x;
Console.WriteLine(e);
// Output:
// x => (x * x)
```

Лямбда-выражения можно использовать в любом коде, для которого требуются экземпляры типов делегатов или деревьев выражений, например в качестве аргумента метода Task.Run(Action) для передачи кода, который должен выполняться в фоновом режиме. Можно также использовать лямбда-выражения при применении LINQ в C#, как показано в следующем примере:

```
int[] numbers = { 2, 3, 4, 5 };
var squaredNumbers = numbers.Select(x => x * x);
Console.WriteLine(string.Join(" ", squaredNumbers));
// Output:
// 4 9 16 25
```

При использовании синтаксиса на основе методов для вызова метода Enumerable.Select в классе System.Linq.Enumerable (например, в LINQ to Objects и LINQ to XML) параметром является тип делегата System.Func<T,TResult>. При вызове метода Queryable.Select в классе System.Linq.Queryable (например, в LINQ to SQL) типом параметра является тип дерева выражения `Expression<Func<TSource,TResult>>`. В обоих случаях можно использовать одно и то же лямбда-выражение для указания значения параметра. Поэтому оба вызова `Select` выглядят одинаково, хотя на самом деле объект, созданный из лямбда-выражения, имеет другой тип.

Выражения-лямбды

Лямбда-выражение с выражением с правой стороны оператора `=>` называется *выражением лямбда*. Выражения-лямбды возвращают результат выражения и принимают следующую основную форму.

```
(input-parameters) => expression
```

Текст выражения лямбды может состоять из вызова метода. Но при создании деревьев выражений, которые вычисляются вне контекста поддержки общеязыковой среды выполнения (CRL) .NET, например в SQL Server, вызовы методов не следует использовать в лямбда-выражениях. Эти методы не имеют смысла вне контекста среды CLR .NET.

Лямбды операторов

Лямбда-инструкция напоминает лямбда-выражение, за исключением того, что инструкции заключаются в фигурные скобки:

```
(input-parameters) => { <sequence-of-statements> }
```

Тело лямбды оператора может состоять из любого количества операторов; однако на практике обычно используется не более двух-трех.

```
Action<string> greet = name =>
{
    string greeting = $"Hello {name}!";
    Console.WriteLine(greeting);
};
greet("World");
// Output:
// Hello World!
```

Лямбда-инструкции нельзя использовать для создания деревьев выражений.

Входные параметры лямбда-выражения

Входные параметры лямбда-выражения заключаются в круглые скобки. Нулевое количество входных параметры задается пустыми скобками:

```
Action line = () => Console.WriteLine();
```

Если лямбда-выражение имеет только один входной параметр, круглые скобки необязательны:

```
Func<double, double> cube = x => x * x * x;
```

Два и более входных параметра разделяются запятыми:

```
Func<int, int, bool> testForEquality = (x, y) => x == y;
```

Иногда компилятор не может вывести типы входных параметров. Вы можете указать типы данных в явном виде, как показано в следующем примере:

```
Func<int, string, bool> isTooLong = (int x, string s) => s.Length > x;
```

Для входных параметров все типы нужно задать либо в явном, либо в неявном виде. В противном случае компилятор выдает ошибку CS0748.

Начиная с C# 9.0, вы можете использовать пустые переменные, чтобы указать два или более входных параметра лямбда-выражения, которые не используются в выражении:

```
Func<int, int, int> constant = (_, _) => 42;
```

Параметры пустой переменной лямбда-выражения полезны, если вы используете лямбда-выражение для указания обработчика событий.

> **NOTE**
> Если только один входной параметр имеет имя `_`, для обеспечения обратной совместимости `_` рассматривается как имя этого параметра в лямбда-выражении.

Асинхронные лямбда-выражения

С помощью ключевых слов async и await можно легко создавать лямбда-выражения и операторы, включающие асинхронную обработку. Например, в следующем примере Windows Forms содержится обработчик событий, который вызывает асинхронный метод `ExampleMethodAsync` и ожидает его.

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += button1_Click;
    }

    private async void button1_Click(object sender, EventArgs e)
    {
        await ExampleMethodAsync();
        textBox1.Text += "\r\nControl returned to Click event handler.\n";
    }

    private async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

Такой же обработчик событий можно добавить с помощью асинхронного лямбда-выражения. Чтобы добавить этот обработчик, поставьте модификатор `async` перед списком параметров лямбда-выражения, как показано в следующем примере:

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += async (sender, e) =>
        {
            await ExampleMethodAsync();
            textBox1.Text += "\r\nControl returned to Click event handler.\n";
        };
    }

    private async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

Дополнительные сведения о создании и использовании асинхронных методов см. в разделе Асинхронное программирование с использованием ключевых слов Async и Await.

Лямбда-выражения и кортежи

В C# 7.0 представлена встроенная поддержка кортежей. Кортеж можно ввести в качестве аргумента лямбда-выражения, и лямбда-выражение также может возвращать кортеж. В некоторых случаях компилятор C# использует определение типа для определения типов компонентов кортежа.

Кортеж определяется путем заключения в скобки списка его компонентов с разделителями-запятыми. В следующем примере кортеж с тремя компонентами используется для передачи последовательности чисел в лямбда-выражение. Оно удваивает каждое значение и возвращает кортеж с тремя компонентами, содержащий результат операций умножения.

```
Func<(int, int, int), (int, int, int)> doubleThem = ns => (2 * ns.Item1, 2 * ns.Item2, 2 * ns.Item3);
var numbers = (2, 3, 4);
var doubledNumbers = doubleThem(numbers);
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");
// Output:
// The set (2, 3, 4) doubled: (4, 6, 8)
```

Как правило, поля кортежи именуются как `Item1`, `Item2` и т. д. Тем не менее кортеж с именованными компонентами можно определить, как показано в следующем примере:

```
Func<(int n1, int n2, int n3), (int, int, int)> doubleThem = ns => (2 * ns.n1, 2 * ns.n2, 2 * ns.n3);
var numbers = (2, 3, 4);
var doubledNumbers = doubleThem(numbers);
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");
```

Дополнительные сведения о кортежах в C# см. в статье Типы кортежей.

Лямбда-выражения со стандартными операторами запросов

В LINQ to Objects, наряду с другими реализациями, есть входной параметр, тип которого принадлежит к семейству универсальных делегатов Func<TResult>. Эти делегаты используют параметры типа для определения количества и типов входных параметров, а также тип возвращаемого значения делегата. Делегаты `Func` полезны для инкапсуляции пользовательских выражений, которые применяются к каждому элементу в наборе исходных данных. В качестве примера рассмотрим следующий тип делегата Func<T,TResult>:

```
public delegate TResult Func<in T, out TResult>(T arg)
```

Экземпляр этого делегата можно создать как `Func<int, bool>`, где `int` — входной параметр, а `bool` — возвращаемое значение. Возвращаемое значение всегда указывается в последнем параметре типа. Например, `Func<int, string, bool>` определяет делегат с двумя входными параметрами, `int` и `string`, и типом возвращаемого значения `bool`. Следующий делегат `Func` при вызове возвращает логическое значение, которое показывает, равен ли входной параметр 5:

```
Func<int, bool> equalsFive = x => x == 5;
bool result = equalsFive(4);
Console.WriteLine(result);   // False
```

Лямбда-выражения также можно использовать, когда аргумент имеет тип Expression<TDelegate>, например в стандартных операторах запросов, которые определены в типе Queryable. При указании аргумента Expression<TDelegate> лямбда-выражение компилируется в дерево выражения.

В этом примере используется стандартный оператор запроса Count:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
int oddNumbers = numbers.Count(n => n % 2 == 1);
Console.WriteLine($"There are {oddNumbers} odd numbers in {string.Join(" ", numbers)}");
```

Компилятор может вывести тип входного параметра ввода; но его также можно определить явным образом. Данное лямбда-выражение подсчитывает указанные целые значения (`n`), которые при делении на два дают остаток 1.

В следующем примере кода показано, как создать последовательность, которая содержит все элементы

массива `numbers`, предшествующие 9, так как это первое число последовательности, не удовлетворяющее условию:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstNumbersLessThanSix = numbers.TakeWhile(n => n < 6);
Console.WriteLine(string.Join(" ", firstNumbersLessThanSix));
// Output:
// 5 4 1 3
```

В следующем примере показано, как указать несколько входных параметров путем их заключения в скобки. Этот метод возвращает все элементы в массиве `numbers` до того числа, значение которого меньше его порядкового номера в массиве:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstSmallNumbers = numbers.TakeWhile((n, index) => n >= index);
Console.WriteLine(string.Join(" ", firstSmallNumbers));
// Output:
// 5 4
```

Лямбда-выражения не используются непосредственно в выражениях запросов, но их можно использовать в вызовах методов в выражениях запросов, как показано в следующем примере:

```
var numberSets = new List<int[]>
{
    new[] { 1, 2, 3, 4, 5 },
    new[] { 0, 0, 0 },
    new[] { 9, 8 },
    new[] { 1, 0, 1, 0, 1, 0, 1, 0 }
};

var setsWithManyPositives =
    from numberSet in numberSets
    where numberSet.Count(n => n > 0) > 3
    select numberSet;

foreach (var numberSet in setsWithManyPositives)
{
    Console.WriteLine(string.Join(" ", numberSet));
}
// Output:
// 1 2 3 4 5
// 1 0 1 0 1 0 1 0
```

Определение типа в лямбда-выражениях

При написании лямбда-выражений обычно не требуется указывать тип входных параметров, так как компилятор может выводить этот тип на основе тела лямбда-выражения, типов параметров и других факторов, как описано в спецификации языка C#. Для большинства стандартных операторов запросов первой входное значение имеет тип элементов в исходной последовательности. При запросе `IEnumerable<Customer>` входная переменная считается объектом `Customer`, а это означает, что у вас есть доступ к его методам и свойствам:

```
customers.Where(c => c.City == "London");
```

Общие правила определения типа для лямбда-выражений формулируются следующим образом:

- лямбда-выражение должно содержать то же число параметров, что и тип делегата;

- каждый входной параметр в лямбда-выражении должен быть неявно преобразуемым в соответствующий параметр делегата;
- возвращаемое значение лямбда-выражения (если таковое имеется) должно быть неявно преобразуемым в возвращаемый тип делегата.

Естественный тип для лямбда-выражений

У лямбда-выражений нет типа, так как в системе общих типов изначально отсутствует такое понятие, как лямбда-выражения. Но применительно к лямбда-выражениям иногда бывает удобно оперировать понятием типа. Под неофициальным термином "тип" понимается тип делегата или тип Expression, в который преобразуется лямбда-выражение.

Начиная с C# 10 некоторые лямбда-выражения имеют *естественный тип*. Вам не придется объявлять тип делегата, например `Func<...>` или `Action<...>` для лямбда-выражения, потому что компилятор может вывести тип делегата из параметров и типа выражения. В качестве примера рассмотрим следующее объявление:

```
var parse = (string s) => int.Parse(s);
```

Компилятор может определить `parse` как `Func<string, int>`. В общем случае компилятор будет использовать доступный делегат `Func` или `Action`, если он существует. В противном случае он синтезирует типа делегата. Например, тип должен быть синтезирован, если лямбда-выражение имеет параметры `ref`. Если лямбда-выражение имеет естественный тип, его можно присвоить менее явному типу, например System.Object, или System.Delegate:

```
object parse = (string s) => int.Parse(s);   // Func<string, int>
Delegate parse = (string s) => int.Parse(s); // Func<string, int>
```

Группы методов (то есть имена методов без списков аргументов) с ровно одной перегрузкой имеют естественный тип:

```
var read = Console.Read; // Just one overload; Func<int> inferred
var write = Console.Write; // ERROR: Multiple overloads, can't choose
```

Если присвоить лямбда-выражение System.Linq.Expressions.LambdaExpression или System.Linq.Expressions.Expression, и лямбда имеет естественный тип делегата, выражение имеет естественный тип System.Linq.Expressions.Expression<TDelegate> с естественным типом делегата, используемым в качестве аргумента для параметра типа:

```
LambdaExpression parseExpr = (string s) => int.Parse(s); // Expression<Func<string, int>>
Expression parseExpr = (string s) => int.Parse(s);       // Expression<Func<string, int>>
```

У многих лямбда-выражений нет естественного типа. Рассмотрим следующее объявление:

```
var parse = s => int.Parse(s); // ERROR: Not enough type info in the lambda
```

Компилятор не может определить тип параметра для `s`. Если компилятор не может определить естественный тип, необходимо объявить тип:

```
Func<string, int> parse = s => int.Parse(s);
```

Объявленный тип возвращаемого значения

Как правило, тип возвращаемого значения лямбда-выражения является очевидным и легко выводится. Для некоторых выражений это не сработает:

```
var choose = (bool b) => b ? 1 : "two"; // ERROR: Can't infer return type
```

Начиная с C# 10 можно указать тип возвращаемого значения для лямбда-выражения перед параметрами. При указании явного типа возвращаемого значения параметры должны быть заключены в круглые скобки, чтобы не путать компилятор и других разработчиков:

```
var choose = object (bool b) => b ? 1 : "two"; // Func<bool, object>
```

Атрибуты

Начиная с C# 10 к лямбда-выражениям можно применять атрибуты. Атрибуты добавляются перед объявлением параметра. При наличии атрибутов список параметров лямбда-выражения должен быть заключен в круглые скобки:

```
Func<string, int> parse = [Example(1)] (s) => int.Parse(s);
var choose = [Example(2)][Example(3)] object (bool b) => b ? 1 : "two";
```

Можно применить любой атрибут, допустимый для AttributeTargets.Method.

Лямбда-выражения вызываются через базовый тип делегата. Это отличается от методов и локальных функций. Метод делегата `Invoke` не проверяет атрибуты в лямбда-выражении. При вызове лямбда-выражения атрибуты не оказывают никакого влияния. Атрибуты лямбда-выражений полезны для анализа кода и могут быть обнаружены с помощью отражения. Одним из последствий этого решения является тот факт, что System.Diagnostics.ConditionalAttribute невозможно применить к лямбда-выражению.

Запись внешних переменных и области видимости переменной в лямбда-выражениях

Лямбда-выражения могут ссылаться *на внешние переменные*. Это переменные в области метода, в котором определено лямбда-выражение, или области типа, который содержит лямбда-выражение. Переменные, полученные таким способом, сохраняются для использования в лямбда-выражениях, даже если бы в ином случае они оказались за границами области действия и уничтожились сборщиком мусора. Внешняя переменная должна быть определенным образом присвоена, прежде чем она сможет использоваться в лямбда-выражениях. В следующем примере демонстрируются эти правила.

```
public static class VariableScopeWithLambdas
{
    public class VariableCaptureGame
    {
        internal Action<int> updateCapturedLocalVariable;
        internal Func<int, bool> isEqualToCapturedLocalVariable;

        public void Run(int input)
        {
            int j = 0;

            updateCapturedLocalVariable = x =>
            {
                j = x;
                bool result = j > input;
                Console.WriteLine($"{j} is greater than {input}: {result}");
            };

            isEqualToCapturedLocalVariable = x => x == j;

            Console.WriteLine($"Local variable before lambda invocation: {j}");
            updateCapturedLocalVariable(10);
            Console.WriteLine($"Local variable after lambda invocation: {j}");
        }
    }

    public static void Main()
    {
        var game = new VariableCaptureGame();

        int gameInput = 5;
        game.Run(gameInput);

        int jTry = 10;
        bool result = game.isEqualToCapturedLocalVariable(jTry);
        Console.WriteLine($"Captured local variable is equal to {jTry}: {result}");

        int anotherJ = 3;
        game.updateCapturedLocalVariable(anotherJ);

        bool equalToAnother = game.isEqualToCapturedLocalVariable(anotherJ);
        Console.WriteLine($"Another lambda observes a new value of captured variable: {equalToAnother}");
    }
    // Output:
    // Local variable before lambda invocation: 0
    // 10 is greater than 5: True
    // Local variable after lambda invocation: 10
    // Captured local variable is equal to 10: True
    // 3 is greater than 5: False
    // Another lambda observes a new value of captured variable: True
}
```

Следующие правила применимы к области действия переменной в лямбда-выражениях.

- Захваченная переменная не будет уничтожена сборщиком мусора до тех пор, пока делегат, который на нее ссылается, не перейдет в статус подлежащего уничтожению при сборке мусора.
- Переменные, представленные в лямбда-выражении, невидимы в заключающем методе.
- Лямбда-выражение не может непосредственно захватывать параметры in, ref или out из заключающего метода.
- Оператор return в лямбда-выражении не вызывает возврат значения заключающим методом.
- Лямбда-выражение не может содержать операторы goto, break или continue, если целевой объект этого оператора перехода находится за пределами блока лямбда-выражения. Если целевой объект находится внутри блока, использование оператора перехода за пределами лямбда-выражения также

будет ошибкой.

Начиная с C# 9.0, вы можете применять модификатор `static` к лямбда-выражению для предотвращения непреднамеренного сохранения локальных переменных или состояния экземпляров лямбда-выражением:

```
Func<double, double> square = static x => x * x;
```

Статическое лямбда-выражение не может сохранять локальные переменные или состояние экземпляров из охватывающих областей, но может ссылаться на статические элементы и определения констант.

Спецификация языка C#

Дополнительные сведения см. в разделе Выражения анонимных функций в спецификации языка C#.

Дополнительные сведения о функциях, добавленных в C# 9.0, см. в следующих заметках о функциях:

- Параметры удаления лямбда-выражения
- Статические анонимные функции

См. также раздел

- справочник по C#
- Операторы и выражения C#
- Встроенный язык запросов LINQ
- Деревья выражений
- Локальные функции или лямбда-выражения
- Примеры запросов LINQ
- Пример XQuery
- Общие сведения о примерах LINQ

Шаблоны (справочник по C#)

18.12.2021 • 13 minutes to read

Возможность использовать сопоставление шаблонов впервые появилась в C# 7.0. С тех пор в каждой новой основной версии C# возможности сопоставления шаблонов расширяются. Сопоставление шаблонов поддерживают следующие выражения и операторы C#:

- Выражение `is`
- Оператор `switch`
- Выражение `switch` (появилось в C# 8.0)

В этих конструкциях можно сравнить входное выражение с любым из следующих шаблонов:

- Шаблон объявления: для проверки типа выражения в среде выполнения и, в случае равенства, присвоения объявленной переменной результата этого выражения. Впервые появился в C# 7.0.
- Шаблон типа: для проверки типа выражения в среде выполнения. Впервые появился в C# 9.0.
- Шаблон константы: для проверки того, равен ли результат выражения указанной константе. Впервые появился в C# 7.0.
- Реляционные шаблоны: для сравнения результата выражения с заданной константой. Впервые появился в C# 9.0.
- Логические шаблоны: для проверки того, соответствует ли выражение логической комбинации шаблонов. Впервые появился в C# 9.0.
- Шаблон свойства: для проверки того, соответствуют ли свойства или поля выражения вложенным шаблонам. Впервые появился в C# 8.0.
- Позиционный шаблон: для деконструкции результата выражения и проверки того, соответствуют ли результирующие значения вложенным шаблонам. Впервые появился в C# 8.0.
- Шаблон `var`: для сравнения любых выражений и присваивания результата сравнения объявленной переменной. Впервые появился в C# 7.0.
- Шаблон пустой переменной: для сравнения любых выражений. Впервые появился в C# 8.0.

Логические шаблоны, шаблоны свойств и позиционные шаблоны являются *рекурсивными* шаблонами. То есть, они могут содержать *вложенные* шаблоны.

Пример использования этих шаблонов для создания управляемого данными алгоритма можно посмотреть в разделе Учебник: использование сопоставления шаблонов для создания управляемых типами и управляемых данными алгоритмов.

Шаблоны объявления и шаблоны типов

Шаблоны объявления и шаблоны типов используются для проверки того, совместим ли с указанным типом тип определенного выражения в среде выполнения. С помощью шаблона объявления можно также объявить новую локальную переменную. Если шаблон объявления соответствует выражению, этой переменной присваивается результат преобразованного выражения, как показано в следующем примере:

```
object greeting = "Hello, World!";
if (greeting is string message)
{
    Console.WriteLine(message.ToLower());  // output: hello, world!
}
```

Начиная с C# 7.0, *шаблон объявления* с типом `T` соответствует выражению в том случае, если результат выражения имеет значение, отличное от NULL, и выполняется любое из следующих условий:

- Тип результата выражения в среде выполнения — `T`.
- Тип результата выражения в среде выполнения является производным от типа `T`, реализует интерфейс `T` или существует другое неявное преобразование ссылок из него в `T`. В следующем примере показаны два случая, когда данное условие истинно:

  ```
  var numbers = new int[] { 10, 20, 30 };
  Console.WriteLine(GetSourceLabel(numbers));  // output: 1

  var letters = new List<char> { 'a', 'b', 'c', 'd' };
  Console.WriteLine(GetSourceLabel(letters));  // output: 2

  static int GetSourceLabel<T>(IEnumerable<T> source) => source switch
  {
      Array array => 1,
      ICollection<T> collection => 2,
      _ => 3,
  };
  ```

  В предыдущем примере при первом вызове метода `GetSourceLabel` первый шаблон соответствует значению аргумента, так как тип этого аргумента (`int[]`) в среде выполнения является производным от типа Array. При втором вызове метода `GetSourceLabel` тип аргумента в среде выполнения (List<T>) не является производным от типа Array, но реализует интерфейс ICollection<T>.

- Тип результата выражения в среде выполнения является типом, допускающим значение NULL, и имеющим базовый тип `T`.
- Существует упаковка-преобразование или распаковка-преобразование из типа результата выражения в среде выполнения в тип `T`.

В следующем примере показаны два последних условия:

```
int? xNullable = 7;
int y = 23;
object yBoxed = y;
if (xNullable is int a && yBoxed is int b)
{
    Console.WriteLine(a + b);  // output: 30
}
```

Если требуется проверить только тип выражения, можно вместо имени переменной использовать пустую переменную `_`, как показано в следующем примере:

```
public abstract class Vehicle {}
public class Car : Vehicle {}
public class Truck : Vehicle {}

public static class TollCalculator
{
    public static decimal CalculateToll(this Vehicle vehicle) => vehicle switch
    {
        Car _ => 2.00m,
        Truck _ => 7.50m,
        null => throw new ArgumentNullException(nameof(vehicle)),
        _ => throw new ArgumentException("Unknown type of a vehicle", nameof(vehicle)),
    };
}
```

Начиная с C# 9.0, для этой цели можно использовать *шаблон типа*, как показано в следующем примере:

```
public static decimal CalculateToll(this Vehicle vehicle) => vehicle switch
{
    Car => 2.00m,
    Truck => 7.50m,
    null => throw new ArgumentNullException(nameof(vehicle)),
    _ => throw new ArgumentException("Unknown type of a vehicle", nameof(vehicle)),
};
```

Как и шаблон объявления, шаблон типа соответствует выражению, если результат выражения не равен NULL, а его тип в среде выполнения удовлетворяет любому из указанных выше условий.

Дополнительные сведения см. в разделах Шаблон объявления и Шаблон типа в примечаниях к предлагаемой функции.

Шаблон константы

Начиная с C# 7.0, вы можете использовать *шаблон константы* для проверки того, равен ли результат выражения заданной константе, как показано в следующем примере:

```
public static decimal GetGroupTicketPrice(int visitorCount) => visitorCount switch
{
    1 => 12.0m,
    2 => 20.0m,
    3 => 27.0m,
    4 => 32.0m,
    0 => 0.0m,
    _ => throw new ArgumentException($"Not supported number of visitors: {visitorCount}",
nameof(visitorCount)),
};
```

В шаблоне константы можно использовать любое константное выражение, например:

- целочисленный литерал или литерал с плавающей точкой;
- символьный или строковый литерал;
- логическое значение `true` или `false`;
- значение типа enum;
- имя объявленного поля с модификатором const или локального элемента.
- `null`

Используйте шаблон константы для проверки на `null`, как показано в следующем примере:

```
if (input is null)
{
    return;
}
```

Компилятор гарантирует, что при вычислении выражения `x is null` не будет вызван перегруженный пользователем оператор равенства `==`.

Начиная с C# 9.0, вы можете использовать шаблон константы `null` с отрицанием для проверки неравенства значению NULL, как показано в следующем примере:

```
if (input is not null)
{
    // ...
}
```

Дополнительные сведения см. в разделе Шаблон константы в примечании к предлагаемой функции.

Реляционные шаблоны

Начиная с C# 9.0, вы можете использовать *реляционный шаблон* для сравнения результата выражения с константой, как показано в следующем примере:

```
Console.WriteLine(Classify(13));  // output: Too high
Console.WriteLine(Classify(double.NaN));  // output: Unknown
Console.WriteLine(Classify(2.4));  // output: Acceptable

static string Classify(double measurement) => measurement switch
{
    < -4.0 => "Too low",
    > 10.0 => "Too high",
    double.NaN => "Unknown",
    _ => "Acceptable",
};
```

В реляционном шаблоне можно использовать любые операторы отношений — `<`, `>`, `<=`, `>=`. Правая часть реляционного шаблона должна быть константным выражением. Константное выражение может быть целым числом, числом с плавающей точкой, символом или иметь тип enum.

Чтобы проверить, находится ли результат выражения в определенном диапазоне, сопоставьте его с шаблоном конъюнкции (`and`), как показано в следующем примере:

```
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 3, 14)));  // output: spring
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 7, 19)));  // output: summer
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 2, 17)));  // output: winter

static string GetCalendarSeason(DateTime date) => date.Month switch
{
    >= 3 and < 6 => "spring",
    >= 6 and < 9 => "summer",
    >= 9 and < 12 => "autumn",
    12 or (>= 1 and < 3) => "winter",
    _ => throw new ArgumentOutOfRangeException(nameof(date), $"Date with unexpected month: {date.Month}."),
};
```

Если результат выражения — `null` или его не удается преобразовать в тип константы с помощью преобразования, допускающего значение NULL, или распаковки-преобразования, то реляционный

шаблон не соответствует выражению.

Дополнительные сведения см. в разделе Реляционные шаблоны в примечании к предлагаемой функции.

Логические шаблоны

Начиная с C# 9.0, вы можете использовать блоки объединения `not`, `and` и `or` для создания следующих *логических шаблонов*:

- *Шаблон отрицания* `not`, который соответствует выражению, если шаблон с отрицанием не соответствует выражению. В следующем примере показано, как можно инвертировать шаблон константы `null`, чтобы проверить, имеет ли выражение значение, отличное от NULL:

  ```
  if (input is not null)
  {
      // ...
  }
  ```

- *Шаблон конъюнкции* `and`, который соответствует выражению, если оба шаблона соответствуют этому выражению. В следующем примере показано, как можно объединить реляционные шаблоны, чтобы проверить, находится ли значение в определенном диапазоне:

  ```
  Console.WriteLine(Classify(13));  // output: High
  Console.WriteLine(Classify(-100));  // output: Too low
  Console.WriteLine(Classify(5.7));  // output: Acceptable

  static string Classify(double measurement) => measurement switch
  {
      < -40.0 => "Too low",
      >= -40.0 and < 0 => "Low",
      >= 0 and < 10.0 => "Acceptable",
      >= 10.0 and < 20.0 => "High",
      >= 20.0 => "Too high",
      double.NaN => "Unknown",
  };
  ```

- *Шаблон дизъюнкции* `or`, который соответствует выражению, если любой из шаблонов соответствует данному выражению, как показано в следующем примере:

  ```
  Console.WriteLine(GetCalendarSeason(new DateTime(2021, 1, 19)));  // output: winter
  Console.WriteLine(GetCalendarSeason(new DateTime(2021, 10, 9)));  // output: autumn
  Console.WriteLine(GetCalendarSeason(new DateTime(2021, 5, 11)));  // output: spring

  static string GetCalendarSeason(DateTime date) => date.Month switch
  {
      3 or 4 or 5 => "spring",
      6 or 7 or 8 => "summer",
      9 or 10 or 11 => "autumn",
      12 or 1 or 2 => "winter",
      _ => throw new ArgumentOutOfRangeException(nameof(date), $"Date with unexpected month:
  {date.Month}."),
  };
  ```

Как показано в предыдущем примере, блоки объединения в шаблоне можно использовать многократно.

Приоритет и порядок проверки

В следующем списке приведены методы объединения шаблонов заказов в порядке убывания приоритета:

- `not`

- `and`
- `or`

Чтобы явно задать приоритет, используйте круглые скобки, как показано в следующем примере:

```
static bool IsLetter(char c) => c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z');
```

> **NOTE**
> Порядок проверки шаблонов не определен. Во время выполнения можно сначала проверить правые вложенные шаблоны шаблонов `or` и `and`.

Дополнительные сведения см. в разделе Блоки объединения шаблонов в примечании к предлагаемой функции.

Шаблон свойства

Начиная с C# 8.0, вы можете использовать *шаблон свойства* для сопоставления свойств или полей выражения с вложенными шаблонами, как показано в следующем примере:

```
static bool IsConferenceDay(DateTime date) => date is { Year: 2020, Month: 5, Day: 19 or 20 or 21 };
```

Шаблон свойства соответствует выражению, если результат выражения не равен NULL, а каждый вложенный шаблон соответствует соответствующему свойству или полю результата выражения.

Вы также можете добавить проверку типа среды выполнения и объявление переменной в шаблон свойства, как показано в следующем примере:

```
Console.WriteLine(TakeFive("Hello, world!"));  // output: Hello
Console.WriteLine(TakeFive("Hi!"));  // output: Hi!
Console.WriteLine(TakeFive(new[] { '1', '2', '3', '4', '5', '6', '7' }));  // output: 12345
Console.WriteLine(TakeFive(new[] { 'a', 'b', 'c' }));  // output: abc

static string TakeFive(object input) => input switch
{
    string { Length: >= 5 } s => s.Substring(0, 5),
    string s => s,

    ICollection<char> { Count: >= 5 } symbols => new string(symbols.Take(5).ToArray()),
    ICollection<char> symbols => new string(symbols.ToArray()),

    null => throw new ArgumentNullException(nameof(input)),
    _ => throw new ArgumentException("Not supported input type."),
};
```

Шаблон свойства является рекурсивным шаблоном. Это значит, что любой шаблон можно использовать как вложенный шаблон. Используйте шаблон свойства для сопоставления элементов данных с вложенными шаблонами, как показано в следующем примере:

```
public record Point(int X, int Y);
public record Segment(Point Start, Point End);

static bool IsAnyEndOnXAxis(Segment segment) =>
    segment is { Start: { Y: 0 } } or { End: { Y: 0 } };
```

В предыдущем примере используются две возможности, доступные в C# 9.0 и более поздних версиях языка: блок объединения шаблонов `or` и типы записей.

Начиная с C# 10 можно ссылаться на вложенные свойства или поля в шаблоне свойства. Например, можно выполнить рефакторинг метода из предыдущего примера в следующий эквивалентный код:

```
static bool IsAnyEndOnXAxis(Segment segment) =>
    segment is { Start.Y: 0 } or { End.Y: 0 };
```

Дополнительные сведения см. в разделе Шаблон свойства в примечании к предлагаемой функции и примечание к предлагаемой функции Расширенные шаблоны свойств.

Позиционный шаблон

Начиная с C# 8.0, вы можете использовать *позиционный шаблон* для деконструкции результата выражения и сравнения результирующих значений с соответствующими вложенными шаблонами, как показано в следующем примере:

```
public readonly struct Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y) => (X, Y) = (x, y);

    public void Deconstruct(out int x, out int y) => (x, y) = (X, Y);
}

static string Classify(Point point) => point switch
{
    (0, 0) => "Origin",
    (1, 0) => "positive X basis end",
    (0, 1) => "positive Y basis end",
    _ => "Just a point",
};
```

В предыдущем примере тип выражения содержит метод Deconstruct, используемый для деконструкции результата выражения. Можно также сопоставлять выражения кортежных типов с позиционными шаблонами. Таким образом можно сопоставить несколько входных значений с различными шаблонами, как показано в следующем примере:

```
static decimal GetGroupTicketPriceDiscount(int groupSize, DateTime visitDate)
    => (groupSize, visitDate.DayOfWeek) switch
    {
        (<= 0, _) => throw new ArgumentException("Group size must be positive."),
        (_, DayOfWeek.Saturday or DayOfWeek.Sunday) => 0.0m,
        (>= 5 and < 10, DayOfWeek.Monday) => 20.0m,
        (>= 10, DayOfWeek.Monday) => 30.0m,
        (>= 5 and < 10, _) => 12.0m,
        (>= 10, _) => 15.0m,
        _ => 0.0m,
    };
```

В предыдущем примере используются реляционные и логические шаблоны, доступные в C# 9.0 и более поздних версиях языка.

В позиционном шаблоне можно использовать имена элементов кортежа и параметры метода `Deconstruct`, как показано в следующем примере:

```
var numbers = new List<int> { 1, 2, 3 };
if (SumAndCount(numbers) is (Sum: var sum, Count: > 0))
{
    Console.WriteLine($"Sum of [{string.Join(" ", numbers)}] is {sum}");  // output: Sum of [1 2 3] is 6
}

static (double Sum, int Count) SumAndCount(IEnumerable<int> numbers)
{
    int sum = 0;
    int count = 0;
    foreach (int number in numbers)
    {
        sum += number;
        count++;
    }
    return (sum, count);
}
```

Можно также расширить позиционный шаблон одним из следующих способов:

- Добавьте проверку типа среды выполнения и объявление переменной, как показано в следующем примере:

  ```
  public record Point2D(int X, int Y);
  public record Point3D(int X, int Y, int Z);

  static string PrintIfAllCoordinatesArePositive(object point) => point switch
  {
      Point2D (> 0, > 0) p => p.ToString(),
      Point3D (> 0, > 0, > 0) p => p.ToString(),
      _ => string.Empty,
  };
  ```

  В предыдущем примере используются позиционные записи, которые неявно обеспечивают выполнение метода `Deconstruct`.

- Используйте шаблон свойства в позиционном шаблоне, как показано в следующем примере:

  ```
  public record WeightedPoint(int X, int Y)
  {
      public double Weight { get; set; }
  }

  static bool IsInDomain(WeightedPoint point) => point is (>= 0, >= 0) { Weight: >= 0.0 };
  ```

- Можно объединить два предыдущих варианта, как показано в следующем примере:

  ```
  if (input is WeightedPoint (> 0, > 0) { Weight: > 0.0 } p)
  {
      // ..
  }
  ```

Позиционный шаблон является рекурсивным шаблоном. Это значит, что любой шаблон можно использовать как вложенный шаблон.

Дополнительные сведения см. в разделе Позиционный шаблон в примечании к предлагаемой функции.

Шаблон `var`

Начиная с C# 7.0, *шаблон* `var` можно использовать для сопоставления любого выражения, включая `null`, и присвоения результата сопоставления новой локальной переменной, как показано в следующем примере:

```
static bool IsAcceptable(int id, int absLimit) =>
    SimulateDataFetch(id) is var results
    && results.Min() >= -absLimit
    && results.Max() <= absLimit;

static int[] SimulateDataFetch(int id)
{
    var rand = new Random();
    return Enumerable
                .Range(start: 0, count: 5)
                .Select(s => rand.Next(minValue: -10, maxValue: 11))
                .ToArray();
}
```

Шаблон `var` полезно использовать, если в логическом выражении вам требуется временная переменная для хранения результатов промежуточных вычислений. Шаблон `var` также можно использовать, если требуется реализовать дополнительные проверки в условиях регистра `when` выражения или оператора `switch`, как показано в следующем примере:

```
public record Point(int X, int Y);

static Point Transform(Point point) => point switch
{
    var (x, y) when x < y => new Point(-x, y),
    var (x, y) when x > y => new Point(x, -y),
    var (x, y) => new Point(x, y),
};

static void TestTransform()
{
    Console.WriteLine(Transform(new Point(1, 2)));  // output: Point { X = -1, Y = 2 }
    Console.WriteLine(Transform(new Point(5, 2)));  // output: Point { X = 5, Y = -2 }
}
```

В предыдущем примере шаблон `var (x, y)` эквивалентен позиционному шаблону `(var x, var y)`.

В шаблоне `var` тип объявленной переменной равен установленному во время компиляции типу выражения, сопоставляемого с данным шаблоном.

Дополнительные сведения см. в разделе Шаблон var в примечании к предлагаемой функции.

Шаблон пустой переменной

Начиная с C# 8.0, *шаблон пустой переменной* `_` используется для сопоставления любых выражений, включая `null`, как показано в следующем примере:

```
Console.WriteLine(GetDiscountInPercent(DayOfWeek.Friday));  // output: 5.0
Console.WriteLine(GetDiscountInPercent(null));  // output: 0.0
Console.WriteLine(GetDiscountInPercent((DayOfWeek)10));  // output: 0.0

static decimal GetDiscountInPercent(DayOfWeek? dayOfWeek) => dayOfWeek switch
{
    DayOfWeek.Monday => 0.5m,
    DayOfWeek.Tuesday => 12.5m,
    DayOfWeek.Wednesday => 7.5m,
    DayOfWeek.Thursday => 12.5m,
    DayOfWeek.Friday => 5.0m,
    DayOfWeek.Saturday => 2.5m,
    DayOfWeek.Sunday => 2.0m,
    _ => 0.0m,
};
```

В предыдущем примере шаблон пустой переменной используется для обработки значения `null` и любых целочисленных значений, которые не соответствуют имеющимся членам перечисления DayOfWeek. Благодаря этому гарантируется, что выражение `switch` в приведенном примере сможет обработать все возможные входные значения. Если в выражении `switch` не используется шаблон пустой переменной и при этом ни один из шаблонов выражения не соответствует входным данным, среда выполнения генерирует исключение. Если выражение `switch` не обрабатывает все возможные входные значения, компилятор генерирует предупреждение.

Шаблон пустой переменной не может быть шаблоном в выражении `is` или операторе `switch`. В этих случаях для сопоставления выражений используйте шаблон `var` с пустой переменной: `var _`.

Дополнительные сведения см. в разделе Шаблон пустой переменной в примечании к предлагаемой функции.

Шаблон в круглых скобках

Начиная с C# 9.0, вы можете использовать круглые скобки вокруг любого шаблона. Как правило, это делается для того, чтобы подчеркнуть или изменить приоритет логических шаблонов, как показано в следующем примере:

```
if (input is not (float or double))
{
    return;
}
```

Спецификация языка C#

Дополнительные сведения см. в следующих примечаниях к предлагаемой функции.

- Сопоставление шаблонов в C# 7.0
- Рекурсивное сопоставление шаблонов (впервые представлено в C# 8.0)
- Изменения в сопоставлении шаблонов в C# 9.0
- Расширенные шаблоны свойств (C# 10)

См. также раздел

- справочник по C#
- Операторы и выражения C#
- Учебник. Использование сопоставления шаблонов для создания управляемых типами и управляемых

данными алгоритмов

Операторы + и +=. Справочник по C#

18.12.2021 • 2 minutes to read

Операторы `+` и `+=` поддерживаются встроенными целыми числовыми типами и числовыми типами с плавающей запятой, а также строковым типом и типамиделегатов.

Сведения об арифметическом операторе `+` см. в разделе Операторы унарного плюса и минуса и Оператор сложения + в статье Арифметические операторы.

Объединение строк

Если один или оба операнда имеют тип string, оператор `+` сцепляет строковые представления этих операндов (строковое представление `null` является пустой строкой):

```
Console.WriteLine("Forgot" + "white space");
Console.WriteLine("Probably the oldest constant: " + Math.PI);
Console.WriteLine(null + "Nothing to add.");
// Output:
// Forgotwhite space
// Probably the oldest constant: 3.14159265358979
// Nothing to add.
```

В C#, начиная с версии 6, реализован более удобный способ форматирования строк, который называется интерполяция строк:

```
Console.WriteLine($"Probably the oldest constant: {Math.PI:F2}");
// Output:
// Probably the oldest constant: 3.14
```

Начиная с C# 10, можно использовать интерполяцию строк для инициализации константной строки, если все выражения, используемые для заполнителей, также являются константными строками.

Объединение делегатов

Для операндов того же типа delegate оператор `+` возвращает новый экземпляр делегата, при вызове которого вызывается сначала левый, а затем правый операнд. Если какой-либо из операндов имеет значение `null`, оператор `+` возвращает значение другого операнда (это тоже может быть `null`). Следующий пример демонстрирует объединение делегатов с помощью оператора `+`:

```
Action a = () => Console.Write("a");
Action b = () => Console.Write("b");
Action ab = a + b;
ab();  // output: ab
```

Для удаления делегатов используйте оператор `-`.

См. дополнительные сведения о типах делегатов.

Оператор присваивания сложения (+=)

Выражение, использующее оператор `+=`, такое как

```
x += y
```

эквивалентно

```
x = x + y
```

за исключением того, что `x` вычисляется только один раз.

В следующем примере иллюстрируется использование оператора `+=`.

```
int i = 5;
i += 9;
Console.WriteLine(i);
// Output: 14

string story = "Start. ";
story += "End.";
Console.WriteLine(story);
// Output: Start. End.

Action printer = () => Console.Write("a");
printer();  // output: a

Console.WriteLine();
printer += () => Console.Write("b");
printer();  // output: ab
```

Можно также использовать оператор `+=`, который позволяет указать метод обработчика событий при подписке на событие. Дополнительные сведения см. в разделе Практическое руководство. Подписка и отмена подписки на события.

Возможность перегрузки оператора

Определяемый пользователем тип может перегружать оператор `+`. При перегрузке бинарного оператора `+` неявно перегружается и соответствующий оператор `+=`. Определяемый пользователем тип не может перегружать оператор `+=` явным образом.

Спецификация языка C#

См. дополнительные сведения об унарном операторе сложение и операторе сложения в спецификации языка C#.

См. также

- справочник по C#
- Операторы и выражения C#
- Сцепка нескольких строк
- События
- Арифметические операторы
- Операторы - и -=

Операторы - и -= (справочник по C#)

18.12.2021 • 2 minutes to read

Операторы `-` и `-=` поддерживаются встроенными целыми числовыми типами и числовыми типами с плавающей запятой, а также типами делегатов.

Сведения об арифметическом операторе `-` см. в разделе Операторы унарного плюса и минуса и Оператор вычитания - в статье Арифметические операторы.

Удаление делегатов

Для операндов одного и того же типа delegate оператор `-` возвращает экземпляр делегата, который вычисляется следующим образом:

- Если оба операнда не равны NULL и список вызовов правого операнда является соответствующим подчиненным списком списка вызовов левого операнда, результатом операции является новый список вызовов, который получается путем удаления записей правого операнда из списка вызовов левого операнда. Если список правого операнда соответствует нескольким смежным подчиненным спискам в списке левого операнда, удаляется только крайний правый совпадающий подсписок. Если удаление приводит к пустому списку, возвращается `null`.

  ```
  Action a = () => Console.Write("a");
  Action b = () => Console.Write("b");

  var abbaab = a + b + b + a + a + b;
  abbaab();  // output: abbaab
  Console.WriteLine();

  var ab = a + b;
  var abba = abbaab - ab;
  abba();  // output: abba
  Console.WriteLine();

  var nihil = abbaab - abbaab;
  Console.WriteLine(nihil is null);  // output: True
  ```

- Если список вызовов правого операнда не является соответствующим смежным подчиненным списком списка вызовов левого операнда, результатом операции является левый операнд. Так, удаление делегата, который не является частью многоадресного делегата, ни к чему не приводит и многоадресный делегат никак не меняется.

```
Action a = () => Console.Write("a");
Action b = () => Console.Write("b");

var abbaab = a + b + b + a + a + b;
var aba = a + b + a;

var first = abbaab - aba;
first();  // output: abbaab
Console.WriteLine();
Console.WriteLine(object.ReferenceEquals(abbaab, first));  // output: True

Action a2 = () => Console.Write("a");
var changed = aba - a;
changed();  // output: ab
Console.WriteLine();
var unchanged = aba - a2;
unchanged();  // output: aba
Console.WriteLine();
Console.WriteLine(object.ReferenceEquals(aba, unchanged));  // output: True
```

В предыдущем примере также показано, что во время удаления делегата сравниваются его экземпляры. Например, делегаты, созданные в результате оценки идентичных лямбда-выражений не будут равны. Дополнительные сведения о равенстве делегатов см. в разделе об операторах равенства делегатов в спецификации языка C#.

- Если левый операнд имеет значение `null`, результатом операции является `null`. Если правый операнд имеет значение `null`, результатом операции является левый операнд.

```
Action a = () => Console.Write("a");

var nothing = null - a;
Console.WriteLine(nothing is null);  // output: True

var first = a - null;
a();  // output: a
Console.WriteLine();
Console.WriteLine(object.ReferenceEquals(first, a));  // output: True
```

Объединить делегаты можно с помощью оператора `+`.

См. дополнительные сведения о типах делегатов.

Оператор присваивания вычитания (-=)

Выражение, использующее оператор `-=`, такое как

```
x -= y
```

эквивалентно

```
x = x - y
```

за исключением того, что `x` вычисляется только один раз.

В следующем примере иллюстрируется использование оператора `-=`.

```
int i = 5;
i -= 9;
Console.WriteLine(i);
// Output: -4

Action a = () => Console.Write("a");
Action b = () => Console.Write("b");
var printer = a + b + a;
printer();  // output: aba

Console.WriteLine();
printer -= a;
printer();  // output: ab
```

Можно также использовать оператор `-=`, который позволяет указать метод обработчика событий для удаления при отмене подписки на событие. Дополнительные сведения см. в разделе Практическое руководство. Подписка и отмена подписки на события.

Возможность перегрузки оператора

Определяемый пользователем тип может перегружать оператор `-`. При перегрузке бинарного оператора `-` неявно перегружается и соответствующий оператор `-=`. Определяемый пользователем тип не может перегружать оператор `-=` явным образом.

Спецификация языка C#

Дополнительные сведения см. в разделе Оператор унарного минуса и Оператор вычитания в спецификации языка C#.

См. также раздел

- справочник по C#
- Операторы и выражения C#
- События
- Арифметические операторы
- Операторы + и +=

Оператор ?: (справочник по C#)

18.12.2021 • 2 minutes to read

Условный оператор `?:`, также называемый тернарным, вычисляет логическое выражение и в зависимости от полученного значения `true` или `false` возвращает результат одного из двух соответствующих выражений, как показано в следующем примере.

```
string GetWeatherDisplay(double tempInCelsius) => tempInCelsius < 20.0 ? "Cold." : "Perfect!";

Console.WriteLine(GetWeatherDisplay(15));  // output: Cold.
Console.WriteLine(GetWeatherDisplay(27));  // output: Perfect!
```

Как показано в предыдущем примере, синтаксис условного оператора выглядит следующим образом.

```
condition ? consequent : alternative
```

Выражение `condition` должно принимать значение `true` или `false`. Если `condition` принимает значение `true`, вычисляется выражение `consequent`, а результат становится результатом операции. Если `condition` принимает значение `false`, вычисляется выражение `alternative`, а результат становится результатом операции. Вычисляется только выражение `consequent` или `alternative`.

Начиная с C# 9.0 условные выражения имеют целевой тип. Это значит, что если известен целевой тип условного выражения, типы `consequent` и `alternative` должны быть неявно преобразованы в целевой тип, как показано в следующем примере:

```
var rand = new Random();
var condition = rand.NextDouble() > 0.5;

int? x = condition ? 12 : null;

IEnumerable<int> xs = x is null ? new List<int>() { 0, 1 } : new int[] { 2, 3 };
```

Если целевой тип условного выражения неизвестен (например, при использовании ключевого слова `var`) или используются C# 8.0 и более ранние версии, типы `consequent` и `alternative` должны быть одинаковыми, либо должно поддерживаться неявное преобразование из одного типа в другой:

```
var rand = new Random();
var condition = rand.NextDouble() > 0.5;

var x = condition ? 12 : (int?)null;
```

Условный оператор имеет правую ассоциативность, то есть выражение формы.

```
a ? b : c ? d : e
```

вычисляется как

```
a ? b : (c ? d : e)
```

TIP

Вы можете использовать следующий мнемонический прием, чтобы запомнить, как оценивается условный оператор:

```
is this condition true ? yes : no
```

Условное выражение REF

Начиная с версии C# 7.2 локальные переменные REF и предназначенные только для чтения локальные переменные REF можно присваивать условным образом с использованием условного выражения REF. Кроме того, условное выражение REF можно использовать как возвращаемое ссылочное значение или как аргумент метода `ref`.

Для условного выражения REF используется следующий синтаксис:

```
condition ? ref consequent : ref alternative
```

Подобно исходному условному оператору, условное выражение REF вычисляет только одно из двух выражений: `consequent` или `alternative`.

Для условного выражения REF типы `consequent` и `alternative` должны совпадать. Условные выражения REF не имеют целевого типа.

В следующем примере иллюстрируется использование условного выражения REF:

```
var smallArray = new int[] { 1, 2, 3, 4, 5 };
var largeArray = new int[] { 10, 20, 30, 40, 50 };

int index = 7;
ref int refValue = ref ((index < 5) ? ref smallArray[index] : ref largeArray[index - 5]);
refValue = 0;

index = 2;
((index < 5) ? ref smallArray[index] : ref largeArray[index - 5]) = 100;

Console.WriteLine(string.Join(" ", smallArray));
Console.WriteLine(string.Join(" ", largeArray));
// Output:
// 1 2 100 4 5
// 10 20 0 40 50
```

Условный оператор и оператор `if`

Если использовать условный оператор вместо оператора `if`, может получиться более лаконичный код в случаях, когда необходимо условно вычислить значение. В следующем примере иллюстрируются два вида классификации целого числа как положительного или отрицательного:

```
int input = new Random().Next(-5, 5);

string classify;
if (input >= 0)
{
    classify = "nonnegative";
}
else
{
    classify = "negative";
}

classify = (input >= 0) ? "nonnegative" : "negative";
```

Возможность перегрузки оператора

Определяемый пользователем тип не может перегружать условный оператор.

Спецификация языка C#

Дополнительные сведения см. в разделе Условный оператор статьи Предварительная спецификация C# 6.0.

Дополнительные сведения о функциях, добавленных в C# 7.2 и более поздние версии, см. в следующих заметках о функциях.

- Условные выражения REF (C# 7.2)
- Условное выражение с целевым типом (C# 9.0)

См. также

- справочник по C#
- Операторы и выражения C#
- Оператор if
- Операторы ?. и ?[]
- Операторы ?? и ??=
- Ключевое слово ref

! — оператор (допускающий значение NULL) (справочник по C#)

18.12.2021 • 2 minutes to read

Унарный постфиксный оператор `!`, доступный в C# 8.0 и более поздних версиях, может допускать или не допускать значение NULL. При включенном контексте аннотаций с поддержкой значения NULL можно использовать оператор, допускающий значение NULL, чтобы объявить, что выражение `x` для ссылочного типа, допускающего значение NULL, не равно `null`: `x!`. Унарный префиксный оператор `!` является оператором логического отрицания.

Оператор, допускающий NULL, ни на что не влияет во время выполнения. Он влияет только на статический анализ потока компилятора путем изменения состояния NULL выражения. Во время выполнения выражение `x!` сравнивается с результатом базового выражения `x`.

Дополнительные сведения о ссылочных типах, допускающих значения NULL, см. в разделе Ссылочные типы, допускающие значение NULL.

Примеры

Один из вариантов использования оператора, допускающего значение NULL, — тестирование логики проверки аргументов. Например, рассмотрим следующий класс.

```
#nullable enable
public class Person
{
    public Person(string name) => Name = name ?? throw new ArgumentNullException(nameof(name));

    public string Name { get; }
}
```

С помощью платформы тестирования MSTest можно создать следующий тест для логики проверки в конструкторе:

```
[TestMethod, ExpectedException(typeof(ArgumentNullException))]
public void NullNameShouldThrowTest()
{
    var person = new Person(null!);
}
```

Без оператора, допускающего значение NULL, компилятор создает следующее предупреждение для предыдущего кода: `Warning CS8625: Cannot convert null literal to non-nullable reference type`. Используя оператор, допускающий значение NULL, вы сообщаете компилятору, что передача `null` ожидается и что о ней не нужно предупреждать.

Можно также использовать оператор, допускающий значение NULL, если вы точно знаете, что выражение не может быть `null`, но компилятор не распознает это. В следующем примере, если метод `IsValid` возвращает `true`, его аргумент не является `null`, и вы можете безопасно отменить его ссылку:

```
public static void Main()
{
    Person? p = Find("John");
    if (IsValid(p))
    {
        Console.WriteLine($"Found {p!.Name}");
    }
}

public static bool IsValid(Person? person)
    => person is not null && person.Name is not null;
```

Без оператора, допускающего значение NULL, компилятор создает следующее предупреждение для кода `p.Name`: `Warning CS8602: Dereference of a possibly null reference`.

Если вы можете изменить метод `IsValid`, можно использовать атрибут NotNullWhen, чтобы сообщить компилятору, что аргумент метода `IsValid` не может быть `null`, когда метод возвращает `true`:

```
public static void Main()
{
    Person? p = Find("John");
    if (IsValid(p))
    {
        Console.WriteLine($"Found {p.Name}");
    }
}

public static bool IsValid([NotNullWhen(true)] Person? person)
    => person is not null && person.Name is not null;
```

В предыдущем примере не нужно использовать оператор, допускающий значение NULL, поскольку компилятор содержит достаточно информации, чтобы определить, что `p` не может быть `null` внутри `if`. См. сведения об атрибутах, позволяющих указать дополнительную информацию о состоянии NULL для переменной, в руководстве по включению в API атрибутов для определения ожидаемых значений NULL.

Спецификация языка C#

См. сведения об операторе, допускающем значение NULL, в черновике спецификации по ссылочным типам допускающим значение NULL.

См. также

- справочник по C#
- Операторы и выражения C#
- Учебник. Разработка с использованием ссылочных типов, допускающих значение NULL

?? и ??= (справочник по C#)

18.12.2021 • 2 minutes to read

Оператор объединения с NULL `??` возвращает значение своего операнда слева, если его значение не равно `null`. В противном случае он вычисляет операнд справа и возвращает его результат. Оператор `??` не выполняет оценку своего операнда справа, если его операнд слева имеет значение, отличное от NULL.

Начиная с C# 8.0 можно использовать оператор присваивания объединения со значением NULL `??=` для присваивания значения правого операнда левому операнду только в том случае, если левый операнд принимает значение `null`. Оператор `??=` не выполняет оценку своего операнда справа, если его операнд слева имеет значение, отличное от NULL.

```
List<int> numbers = null;
int? a = null;

(numbers ??= new List<int>()).Add(5);
Console.WriteLine(string.Join(" ", numbers));  // output: 5

numbers.Add(a ??= 0);
Console.WriteLine(string.Join(" ", numbers));  // output: 5 0
Console.WriteLine(a);  // output: 0
```

Левый операнд оператора `??=` должен быть переменной, свойством или элементом индексатора.

В C# 7.3 и более ранних версий левый операнд оператора `??` должен иметь либо ссылочный тип, либо тип значения, допускающий значение NULL. Начиная с C# 8.0 это требование заменяется следующим: тип левого операнда операторов `??` и `??=` не может быть типом значения, не допускающим значение NULL. В частности, начиная с версии C# 8.0, можно использовать операторы объединения со значением NULL с неограниченными параметрами типа:

```
private static void Display<T>(T a, T backup)
{
    Console.WriteLine(a ?? backup);
}
```

Операторы объединения со значением NULL являются правоассоциативными. То есть выражения в форме

```
a ?? b ?? c
d ??= e ??= f
```

вычисляются как

```
a ?? (b ?? c)
d ??= (e ??= f)
```

Примеры

Операторы `??` и `??=` могут быть полезны в таких случаях:

- В выражениях с NULL-условными операторами ?. и ?[] можно использовать оператор `null`, чтобы задать альтернативное выражение для оценки на случай, если результат выражения с NULL-условной операцией будет равен `??`.

```
double SumNumbers(List<double[]> setsOfNumbers, int indexOfSetToSum)
{
    return setsOfNumbers?[indexOfSetToSum]?.Sum() ?? double.NaN;
}

var sum = SumNumbers(null, 0);
Console.WriteLine(sum);  // output: NaN
```

- Когда вы работаете с типами, допускающими значение NULL, и вам нужно указать значение базового типа значения, используйте оператор `??` для указания значения, возвращаемого в том случае, если значение типа, допускающего значение NULL, равно `null`.

```
int? a = null;
int b = a ?? -1;
Console.WriteLine(b);  // output: -1
```

  Используйте метод Nullable<T>.GetValueOrDefault(), если значение, которое будет использоваться, когда значение типа, допускающего значение NULL, равно `null`, должно быть значением по умолчанию базового типа.

- Начиная с версии C# 7.0, можно сократить код проверки аргументов, используя выражение `throw` в качестве правого операнда оператора `??`:

```
public string Name
{
    get => name;
    set => name = value ?? throw new ArgumentNullException(nameof(value), "Name cannot be null");
}
```

  В предыдущем примере также демонстрируются способы определения свойства с помощью членов, заданных выражениями.

- Начиная с версии C# 8.0, можно использовать оператор `??=` для замены кода формы

```
if (variable is null)
{
    variable = expression;
}
```

  на новый код:

```
variable ??= expression;
```

Возможность перегрузки оператора

Операторы `??` и `??=` не могут быть перегружены.

Спецификация языка C#

Дополнительные сведения об операторе `??` см. в разделе об операторе объединения со значением

NULL в спецификации языка C#.

См. сведения об операторе `??=` в примечании к предлагаемой функции.

См. также раздел

- справочник по C#
- Операторы и выражения C#
- Операторы ?. и ?[]
- Оператор ?

Оператор => (справочник по C#)

18.12.2021 • 2 minutes to read

Токен `=>` поддерживается в двух формах: в виде лямбда-оператора и в виде разделителя имени члена и реализации члена в определении тела выражения.

Лямбда-оператор

В лямбда-выражениях лямбда-оператор `=>` используется для отделения входных параметров с левой стороны от тела лямбда-выражения с правой стороны.

В следующих примерах используется функция LINQ с синтаксисом метода для демонстрации применения лямбда-выражений:

```
string[] words = { "bot", "apple", "apricot" };
int minimalLength = words
  .Where(w => w.StartsWith("a"))
  .Min(w => w.Length);
Console.WriteLine(minimalLength);   // output: 5

int[] numbers = { 4, 7, 10 };
int product = numbers.Aggregate(1, (interim, next) => interim * next);
Console.WriteLine(product);   // output: 280
```

Входные параметры лямбда-выражений строго типизируются во время компиляции. Если компилятор может выводить типы входных параметров, как в предыдущем примере, вы можете опустить объявления типа. Если требуется указать тип входных параметров, это необходимо делать для каждого такого параметра, как показано в следующем примере:

```
int[] numbers = { 4, 7, 10 };
int product = numbers.Aggregate(1, (int interim, int next) => interim * next);
Console.WriteLine(product);   // output: 280
```

В следующем примере показано, как определить лямбда-выражение без входных параметров:

```
Func<string> greet = () => "Hello, World!";
Console.WriteLine(greet());
```

Дополнительные сведения см. в разделе Лямбда-выражения.

Определения тела выражения

Определение тела выражения имеет следующий общий синтаксис:

```
member => expression;
```

где `expression` — любое допустимое выражение. Тип возвращаемого значения `expression` должен быть неявно преобразуемым в тип возвращаемого значения элемента. Если элемент:

- имеет тип возвращаемого значения `void` или

- является:
  - Конструктор
  - Метод завершения
  - Метод доступа к свойству или индексатору `set`

то `expression` должен быть *выражением оператора*. Поскольку результат выражения отбрасывается, возвращаемым типом этого выражения может быть любой тип.

В следующем примере приводится определение тела выражения для метода `Person.ToString`:

```
public override string ToString() => $"{fname} {lname}".Trim();
```

Это сокращенная версия следующего определения метода:

```
public override string ToString()
{
   return $"{fname} {lname}".Trim();
}
```

Начиная с версии C# 6, поддерживаются определения тела выражения для методов, операторов и доступных только для чтения свойств. Начиная с версии C# 7.0, поддерживаются определения тела выражения для конструкторов, методов завершения, методов доступа свойств и индексаторов.

Дополнительные сведения см. в разделе Элементы, воплощающие выражение.

Возможность перегрузки оператора

Оператор `=>` перегрузить нельзя.

Спецификация языка C#

См. сведения о лямбда-операторе в разделе Выражения анонимных функций в спецификации языка C#.

См. также

- справочник по C#
- Операторы и выражения C#

Оператор :: (справочник по C#)

18.12.2021 • 2 minutes to read

Используйте квалификатор псевдонима пространства имен `::` для получения доступа к элементам пространства имен, обозначенного псевдонимом. Квалификатор `::` используется только между двумя идентификаторами. Левый идентификатор может быть любым из следующих псевдонимов:

- Псевдоним пространства имен, созданный с помощью директивы псевдонима using:

```
using forwinforms = System.Drawing;
using forwpf = System.Windows;

public class Converters
{
    public static forwpf::Point Convert(forwinforms::Point point) => new forwpf::Point(point.X,
point.Y);
}
```

- Псевдоним extern.

- Псевдоним `global`, который является псевдонимом глобального пространства имен. Глобальное пространство имен — это пространство имен, которое содержит пространства имен и типы, не объявленные в именованном пространстве имен. При использовании с квалификатором `::` псевдоним `global` всегда ссылается на глобальное пространство имен, даже если существует определяемый пользователем псевдоним пространства имен `global`.

  В следующем примере используется псевдоним `global` для доступа к пространству имен .NET System, которое является элементом глобального пространства имен. Без псевдонима `global` доступ к определяемому пользователем пространству имен `System`, которое является элементом пространства имен `MyCompany.MyProduct`, будет осуществляться следующим образом:

```
namespace MyCompany.MyProduct.System
{
    class Program
    {
        static void Main() => global::System.Console.WriteLine("Using global alias");
    }

    class Console
    {
        string Suggestion => "Consider renaming this class";
    }
}
```

> **NOTE**
> Ключевое слово `global` является псевдонимом глобального пространства имен, только если оно является левым идентификатором квалификатора `::`.

Можно также использовать маркер `.` для получения доступа к элементам пространства имен, обозначенного псевдонимом. При этом маркер `.` также используется для доступа к элементам типа. Квалификатор `::` гарантирует, что его левый идентификатор всегда ссылается на псевдоним

пространства имен, даже если существует тип или пространство имен с таким же именем.

Спецификация языка C#

Дополнительные сведения см. в описании квалификаторов псевдонимов пространства имен в спецификации языка C#.

См. также

- справочник по C#
- Операторы и выражения C#

Оператор await (справочник по C#)

18.12.2021 • 3 minutes to read

Оператор `await` приостанавливает вычисление включающего асинхронного метода до завершения асинхронной операции, представленной его операндом. После завершения асинхронной операции оператор `await` возвращает результат операции, если таковой имеется. Когда оператор `await` применяется к операнду, который представляет уже завершенную операцию, он возвращает результат операции немедленно без приостановки включающего метода. Оператор `await` не блокирует поток, который вычисляет асинхронный метод. Когда оператор `await` приостанавливает включающий метод async, элемент управления возвращается вызывающему объекту метода.

В следующем примере метод HttpClient.GetByteArrayAsync возвращает экземпляр `Task<byte[]>`, который представляет асинхронную операцию, создающую массив байтов после завершения. До завершения операции оператор `await` приостанавливает метод `DownloadDocsMainPageAsync`. Когда `DownloadDocsMainPageAsync` приостанавливается, управление возвращается методу `Main`, который является вызывающим объектом `DownloadDocsMainPageAsync`. Метод `Main` выполняется до тех пор, пока ему не потребуется результат асинхронной операции, выполняемой методом `DownloadDocsMainPageAsync`. Когда GetByteArrayAsync получает все байты, вычисляется остальная часть метода `DownloadDocsMainPageAsync`. После этого вычисляется остальная часть метода `Main`.

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

public class AwaitOperator
{
    public static async Task Main()
    {
        Task<int> downloading = DownloadDocsMainPageAsync();
        Console.WriteLine($"{nameof(Main)}: Launched downloading.");

        int bytesLoaded = await downloading;
        Console.WriteLine($"{nameof(Main)}: Downloaded {bytesLoaded} bytes.");
    }

    private static async Task<int> DownloadDocsMainPageAsync()
    {
        Console.WriteLine($"{nameof(DownloadDocsMainPageAsync)}: About to start downloading.");

        var client = new HttpClient();
        byte[] content = await client.GetByteArrayAsync("https://docs.microsoft.com/en-us/");

        Console.WriteLine($"{nameof(DownloadDocsMainPageAsync)}: Finished downloading.");
        return content.Length;
    }
}
// Output similar to:
// DownloadDocsMainPageAsync: About to start downloading.
// Main: Launched downloading.
// DownloadDocsMainPageAsync: Finished downloading.
// Main: Downloaded 27700 bytes.
```

В предыдущем примере используется асинхронный метод `Main`, который доступен, начиная с версии C# 7.1. Дополнительные сведения см. в описании оператора await в методе Main.

> **NOTE**
> Общие сведения об асинхронном программировании см. в разделе Асинхронное программирование с использованием ключевых слов async и await. Асинхронное программирование `async` и `await` следует асинхронной модели, основанной на задачах.

Оператор `await` можно использовать только в методе, лямбда-выражении или в анонимном методе, изменяемом ключевым словом async. В асинхронном методе нельзя использовать оператор `await` в теле синхронной функции, внутри блока инструкции lock и в ненадежном контексте.

Операнд оператора `await` обычно имеет один из следующих типов .NET: Task, Task<TResult>, ValueTask или ValueTask<TResult>. Однако любое ожидаемое выражение может быть операндом оператора `await`. Дополнительные сведения см. в разделе об ожидаемых выражениях в спецификации языка C#.

Тип выражения `await t` является `TResult`, если тип выражения `t` является Task<TResult> или ValueTask<TResult>. Если тип `t` является Task или ValueTask, тип `await t` является `void`. В обоих случаях, если `t` вызывает исключение, `await t` воспроизводит это исключение. Дополнительные сведения об обработке исключений см. в разделе Исключения в асинхронных методах в статье о try-catch.

Ключевые слова `async` и `await` доступны, начиная с версии C# 5.

Асинхронные потоки и освобождаемые объекты

Начиная с версии C# 8.0 вы можете работать с асинхронными потоками и освобождаемыми объектами.

Вы можете использовать инструкцию `await foreach` для работы с асинхронным потоком данных. Дополнительные сведения см. в разделе Инструкция `foreach` статьи Инструкции итерации и разделе Асинхронные потоки статьи Новые возможности в C# 8.0.

Инструкция `await using` используется для работы с асинхронно освобождаемым объектом, то есть с объектом типа, который реализует интерфейс IAsyncDisposable. Дополнительные сведения см. в разделе Использование асинхронных освобождаемых объектов статьи Реализация метода DisposeAsync.

Оператор await в методе Main

Начиная с версии C# 7.1 метод `Main`, который является точкой входа приложения, может возвращать `Task` или `Task<int>`, позволяя использовать оператор `await` в теле. В более ранних версиях C#, чтобы убедиться, что метод `Main` ожидает завершения асинхронной операции, можно получить значение свойства Task<TResult>.Result экземпляра Task<TResult>, возвращаемого соответствующим асинхронным методом. Для асинхронных операций, которые не создают значение, можно вызвать метод Task.Wait. См. сведения о том, как выбрать версию языка в описании версий C#.

Спецификация языка C#

Дополнительные сведения см. в разделе об ожидаемых выражениях в спецификации языка C#.

См. также

- справочник по C#
- Операторы и выражения C#
- async
- Асинхронная модель программирования
- Асинхронное программирование

- Подробный обзор асинхронного программирования
- Пошаговое руководство. Получение доступа к Интернету с помощью async и await
- Учебник. Создание и использование асинхронных потоков с использованием C# 8.0 и .NET Core 3.0

Выражения default value (справочник по C#)

18.12.2021 • 2 minutes to read

Выражение default value создает значение по умолчанию для типа. Выражения default value бывают двух видов: вызов оператора default и литерал default.

Также используется ключевое слово `default` в качестве метки варианта по умолчанию в инструкции `switch`.

оператор default

Оператор `default` принимает в качестве аргумента имя типа или параметр типа, как показано в следующем примере:

```
Console.WriteLine(default(int));  // output: 0
Console.WriteLine(default(object) is null);  // output: True

void DisplayDefaultOf<T>()
{
    var val = default(T);
    Console.WriteLine($"Default value of {typeof(T)} is {(val == null ? "null" : val.ToString())}.");
}

DisplayDefaultOf<int?>();
DisplayDefaultOf<System.Numerics.Complex>();
DisplayDefaultOf<System.Collections.Generic.List<int>>();
// Output:
// Default value of System.Nullable`1[System.Int32] is null.
// Default value of System.Numerics.Complex is (0, 0).
// Default value of System.Collections.Generic.List`1[System.Int32] is null.
```

Литерал default

Начиная с C# 7.1, можно использовать литерал `default` для создания значения по умолчанию для типа, если компилятор может вывести тип выражения. Литеральное выражение `default` создает то же значение, что и выражение `default(T)`, где `T` является выведенным типом. Литерал `default` можно использовать в любом из следующих случаев:

- при назначении или инициализации переменной;
- в объявлении значения по умолчанию для необязательного параметра метода;
- в вызове метода для предоставления значения аргумента;
- в инструкции `return` или в качестве выражения в элементе в тексте выражения.

Ниже приведен пример применения литерала `default`.

```
T[] InitializeArray<T>(int length, T initialValue = default)
{
    if (length < 0)
    {
        throw new ArgumentOutOfRangeException(nameof(length), "Array length must be nonnegative.");
    }

    var array = new T[length];
    for (var i = 0; i < length; i++)
    {
        array[i] = initialValue;
    }
    return array;
}

void Display<T>(T[] values) => Console.WriteLine($"[ {string.Join(", ", values)} ]");

Display(InitializeArray<int>(3));  // output: [ 0, 0, 0 ]
Display(InitializeArray<bool>(4, default));  // output: [ False, False, False, False ]

System.Numerics.Complex fillValue = default;
Display(InitializeArray(3, fillValue));  // output: [ (0, 0), (0, 0), (0, 0) ]
```

Спецификация языка C#

Дополнительные сведения см. в разделе о выражениях значения по умолчанию в спецификации языка C#.

Дополнительные сведения о литерале `default` см. в примечании к предлагаемой функции.

См. также

- справочник по C#
- Операторы и выражения C#
- Значения по умолчанию типов C#
- Универсальные шаблоны в .NET

Справочник по C#. Оператор delegate

18.12.2021 • 2 minutes to read

Оператор `delegate` создает анонимный метод, который можно преобразовать в тип делегата:

```
Func<int, int, int> sum = delegate (int a, int b) { return a + b; };
Console.WriteLine(sum(3, 4));  // output: 7
```

NOTE

Начиная с C# 3, лямбда-выражения позволяют быстрее и проще создавать анонимные функции. С помощью оператора `=>` создайте лямбда-выражение:

```
Func<int, int, int> sum = (a, b) => a + b;
Console.WriteLine(sum(3, 4));  // output: 7
```

См. подробнее о возможностях лямбда-выражений (например, об использовании внешних переменных).

При использовании оператора `delegate` вам, возможно, нужно будет пропустить список параметров. В таком случае созданный анонимный метод можно будет преобразовать в тип делегата с любым списком параметров, как показано в примере ниже:

```
Action greet = delegate { Console.WriteLine("Hello!"); };
greet();

Action<int, double> introduce = delegate { Console.WriteLine("This is world!"); };
introduce(42, 2.7);

// Output:
// Hello!
// This is world!
```

Это единственная функция анонимных методов, которая не поддерживается лямбда-выражениями. Во всех остальных случаях лямбда-выражение является предпочтительным способом написания встроенного кода.

Начиная с C# 9.0, можно использовать отмены, чтобы указать два или более входных параметра анонимного метода, которые не используются методом:

```
Func<int, int, int> constant = delegate (int _, int _) { return 42; };
Console.WriteLine(constant(3, 4));  // output: 42
```

Если только один параметр имеет имя `_`, для обеспечения обратной совместимости `_` рассматривается как имя этого параметра в анонимном методе.

Кроме того, начиная с C# 9.0 можно использовать модификатор `static` в объявлении анонимного метода:

```
Func<int, int, int> sum = static delegate (int a, int b) { return a + b; };
Console.WriteLine(sum(10, 4));  // output: 14
```

Статический анонимный метод не может записывать локальные переменные или состояние экземпляра из охватывающих областей.

Вы также можете использовать ключевое слово `delegate` для объявления типа делегата.

Спецификация языка C#

Дополнительные сведения см. в разделе Выражения анонимных функций в спецификации языка C#.

См. также

- справочник по C#
- Операторы и выражения C#
- => оператор

Оператор is (справочник по C#)

18.12.2021 • 2 minutes to read

Оператор `is` проверяет, совместим ли результат выражения с указанным типом. Сведения об операторе тестирования типов `is` см. в соответствующем разделе статьи об операторах приведения и тестирования типов.

Начиная с версии C# 7.0 можно также использовать оператор `is`, чтобы сопоставить выражение с шаблоном, как показано в следующем примере:

```
static bool IsFirstFridayOfOctober(DateTime date) =>
    date is { Month: 10, Day: <=7, DayOfWeek: DayOfWeek.Friday };
```

В предыдущем примере оператор `is` сопоставляет выражение с шаблоном свойства (доступно в C# 8.0 и последующих версий) с вложенной константой и реляционными (доступны в C# 9.0 и последующих версий) шаблонами.

Оператор `is` может быть полезен в таких случаях:

- Проверка типа результата выражения в среде выполнения, как показано в следующем примере:

  ```
  int i = 34;
  object iBoxed = i;
  int? jNullable = 42;
  if (iBoxed is int a && jNullable is int b)
  {
      Console.WriteLine(a + b);  // output 76
  }
  ```

  В предыдущем примере показано использование шаблона объявления.

- Проверка значения `null`, как показано в следующем примере:

  ```
  if (input is null)
  {
      return;
  }
  ```

  Когда выражение сопоставляется с `null`, компилятор гарантирует, что ни один из перегруженных пользователем операторов `==` или `!=` не вызывается.

- Начиная с версии C# 9.0 можно использовать шаблон отрицания для проверки неравенства значению NULL, как показано в следующем примере:

  ```
  if (result is not null)
  {
      Console.WriteLine(result.ToString());
  }
  ```

> **NOTE**
> Полный список шаблонов, поддерживаемых оператором `is`, см. в разделе Шаблоны.

Спецификация языка C#

Подробные сведения см. в разделе The is operator (Оператор is) спецификации языка C# и в следующих предложениях для языка C#:

- Сопоставление шаблонов
- Сопоставление шаблонов с универсальными шаблонами

См. также раздел

- справочник по C#
- Операторы и выражения C#
- Шаблоны
- Учебник. Использование сопоставления шаблонов для создания управляемых типами и управляемых данными алгоритмов
- Операторы приведения и тестирования типов

Выражение nameof (справочник по C#)

18.12.2021 • 2 minutes to read

Выражение `nameof` создает имя, тип или элемент переменной в качестве строковой константы:

```
Console.WriteLine(nameof(System.Collections.Generic));  // output: Generic
Console.WriteLine(nameof(List<int>));  // output: List
Console.WriteLine(nameof(List<int>.Count));  // output: Count
Console.WriteLine(nameof(List<int>.Add));  // output: Add

var numbers = new List<int> { 1, 2, 3 };
Console.WriteLine(nameof(numbers));  // output: numbers
Console.WriteLine(nameof(numbers.Count));  // output: Count
Console.WriteLine(nameof(numbers.Add));  // output: Add
```

Как показано в предыдущем примере, при использовании типа и пространства имен созданное имя не является полным.

Если используются буквальные идентификаторы, символ `@` не является частью имени, как показано в следующем примере.

```
var @new = 5;
Console.WriteLine(nameof(@new));  // output: new
```

Значение выражения `nameof` вычисляется во время компиляции, и это не влияет на время выполнения.

С помощью выражения `nameof` можно сделать код проверки аргументов более удобным:

```
public string Name
{
    get => name;
    set => name = value ?? throw new ArgumentNullException(nameof(value), $"{nameof(Name)} cannot be null");
}
```

Выражение `nameof` доступно в C# версии 6 и выше.

Спецификация языка C#

См. подробнее о выражениях nameof в спецификации языка C#.

См. также

- справочник по C#
- Операторы и выражения C#
- CallerArgumentExpression

Оператор new (справочник по C#)

18.12.2021 • 2 minutes to read

Оператор `new` создает экземпляр типа.

Ключевое слово `new` можно также использовать как модификатор объявления члена или ограничение универсального типа.

Вызов конструктора

Для создания экземпляра типа обычно вызывается один из конструкторов этого типа с помощью оператора `new`:

```
var dict = new Dictionary<string, int>();
dict["first"] = 10;
dict["second"] = 20;
dict["third"] = 30;

Console.WriteLine(string.Join("; ", dict.Select(entry => $"{entry.Key}: {entry.Value}")));
// Output:
// first: 10; second: 20; third: 30
```

Для создания экземпляра объекта и его инициализации в одном операторе можно использовать инициализатор объекта или элементов с оператором `new`, как в следующем примере:

```
var dict = new Dictionary<string, int>
{
    ["first"] = 10,
    ["second"] = 20,
    ["third"] = 30
};

Console.WriteLine(string.Join("; ", dict.Select(entry => $"{entry.Key}: {entry.Value}")));
// Output:
// first: 10; second: 20; third: 30
```

Начиная с версии C# 9.0 выражения вызова конструктора имеют целевой тип. То есть, если известен целевой тип выражения, вы можете опустить имя типа, как показано в следующем примере:

```
List<int> xs = new();
List<int> ys = new(capacity: 10_000);
List<int> zs = new() { Capacity = 20_000 };

Dictionary<int, List<int>> lookup = new()
{
    [1] = new() { 1, 2, 3 },
    [2] = new() { 5, 8, 3 },
    [5] = new() { 1, 0, 4 }
};
```

Как показано в предыдущем примере, выражение `new` с целевым типом всегда указывается в скобках.

Если целевой тип выражения `new` неизвестен (например, если вы используете ключевое слово `var`), нужно указать имя типа.

создание массива

С помощью оператора `new` можно также создать экземпляр массива, как показано в следующем примере:

```
var numbers = new int[3];
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;

Console.WriteLine(string.Join(", ", numbers));
// Output:
// 10, 20, 30
```

Чтобы создать экземпляр массива и заполнить его элементами в одном операторе, используйте синтаксис инициализации массива. В следующем примере показаны различные способы сделать это:

```
var a = new int[3] { 10, 20, 30 };
var b = new int[] { 10, 20, 30 };
var c = new[] { 10, 20, 30 };
Console.WriteLine(c.GetType());  // output: System.Int32[]
```

Дополнительные сведения см. в руководстве по работе с массивами.

Создание экземпляров анонимных типов

Чтобы создать экземпляр анонимного типа, используйте оператор `new` и синтаксис инициализации объекта:

```
var example = new { Greeting = "Hello", Name = "World" };
Console.WriteLine($"{example.Greeting}, {example.Name}!");
// Output:
// Hello, World!
```

Уничтожение экземпляров типа

Уничтожать ранее созданные экземпляры типа необязательно. Экземпляры как ссылочных типов, так и типов значений уничтожаются автоматически. Экземпляры типов значений уничтожаются, как только уничтожается содержащий их контекст. Экземпляры ссылочных типов уничтожаются сборщиком мусора в неуказанное время после удаления последней ссылки на них.

Для экземпляров типа, которые содержат неуправляемые ресурсы, например дескриптор файла, рекомендуется использовать детерминированную очистку, чтобы как можно скорее высвободить эти ресурсы. Дополнительные сведения см. в справке по API System.IDisposable и статье об операторе using.

Возможность перегрузки оператора

Пользовательский тип не может перегружать оператор `new`.

Спецификация языка C#

Дополнительные сведения см. в разделе Оператор newспецификация языка C#.

Подробные сведения о выражении `new` с целевым типом см. в примечании к предлагаемой функции.

См. также

- справочник по C#
- Операторы и выражения C#
- Инициализаторы объектов и коллекций

Справочник по C#. Оператор sizeof

18.12.2021 • 2 minutes to read

Оператор `sizeof` позволяет получать число байт, занятых переменной заданного типа. Аргумент оператора `sizeof` должен быть именем неуправляемого типа или параметром типа, к которому применяется ограничение, указывающее, что он является неуправляемым типом.

Для использования оператора `sizeof` требуется небезопасный контекст. Однако выражения, приведенные в следующей таблице, вычисляются во время компиляции для получения соответствующих константных значений и им не требуется небезопасный контекст.

ВЫРАЖЕНИЕ	КОНСТАНТА
`sizeof(sbyte)`	1
`sizeof(byte)`	1
`sizeof(short)`	2
`sizeof(ushort)`	2
`sizeof(int)`	4
`sizeof(uint)`	4
`sizeof(long)`	8
`sizeof(ulong)`	8
`sizeof(char)`	2
`sizeof(float)`	4
`sizeof(double)`	8
`sizeof(decimal)`	16
`sizeof(bool)`	1

Небезопасный контекст также не требуется, если операнд оператора `sizeof` является именем перечисляемого типа.

В следующем примере иллюстрируется использование оператора `sizeof`.

```
using System;

public struct Point
{
    public Point(byte tag, double x, double y) => (Tag, X, Y) = (tag, x, y);

    public byte Tag { get; }
    public double X { get; }
    public double Y { get; }
}

public class SizeOfOperator
{
    public static void Main()
    {
        Console.WriteLine(sizeof(byte));  // output: 1
        Console.WriteLine(sizeof(double));  // output: 8

        DisplaySizeOf<Point>();  // output: Size of Point is 24
        DisplaySizeOf<decimal>();  // output: Size of System.Decimal is 16

        unsafe
        {
            Console.WriteLine(sizeof(Point*));  // output: 8
        }
    }

    static unsafe void DisplaySizeOf<T>() where T : unmanaged
    {
        Console.WriteLine($"Size of {typeof(T)} is {sizeof(T)}");
    }
}
```

Оператор `sizeof` возвращает число байт, выделяемых средой CLR в управляемой памяти. Для типов структуры в это значение входит любое заполнение, как показано в предыдущем примере. Результат использования оператора `sizeof` может отличаться от результата работы метода Marshal.SizeOf, который возвращает размер типа в *неуправляемой* памяти.

Спецификация языка C#

Дополнительные сведения см. в разделе Оператор sizeofспецификации языка C#.

См. также раздел

- справочник по C#
- Операторы и выражения C#
- Операторы, связанные с указателем
- Типы указателей
- Типы, связанные с памятью и диапазонами
- Универсальные шаблоны в .NET

выражение stackalloc (справочник по C#)

18.12.2021 • 2 minutes to read

Выражение `stackalloc` выделяет блок памяти в стеке. Выделенный в стеке блок памяти, который создает этот метод, автоматически удаляется по завершении выполнения метода. Вы не можете явным образом освободить память, выделенную `stackalloc`. Выделенный в стеке блок памяти не подвергается сборке мусора, поэтому его не нужно закреплять с помощью инструкции `fixed`.

Результат выполнения выражения `stackalloc` можно присвоить переменной любого из следующих типов:

- Начиная с версии C# 7.2, System.Span<T> или System.ReadOnlySpan<T>, как показано в следующем примере:

  ```
  int length = 3;
  Span<int> numbers = stackalloc int[length];
  for (var i = 0; i < length; i++)
  {
      numbers[i] = i;
  }
  ```

  Нет необходимости использовать небезопасный контекст при назначении выделенного в стеке блока памяти переменной Span<T> или ReadOnlySpan<T>.

  При работе с такими типами вы можете использовать выражение `stackalloc` в условном выражении или выражении присваивания, как показано в следующем примере:

  ```
  int length = 1000;
  Span<byte> buffer = length <= 1024 ? stackalloc byte[length] : new byte[length];
  ```

  Начиная с версии C# 8.0, можно использовать выражение `stackalloc` внутри других выражений, если разрешена переменная Span<T> или ReadOnlySpan<T>, как показано в следующем примере:

  ```
  Span<int> numbers = stackalloc[] { 1, 2, 3, 4, 5, 6 };
  var ind = numbers.IndexOfAny(stackalloc[] { 2, 4, 6, 8 });
  Console.WriteLine(ind);  // output: 1
  ```

  NOTE

  Мы рекомендуем везде, где это возможно, использовать для работы с выделенной в стеке памятью типы Span<T> или ReadOnlySpan<T>.

- Тип указателя, как показано в следующем примере.

```
unsafe
{
    int length = 3;
    int* numbers = stackalloc int[length];
    for (var i = 0; i < length; i++)
    {
        numbers[i] = i;
    }
}
```

Как демонстрирует пример выше, при работе с типами указателей необходимо использовать контекст `unsafe`.

В случае типов указателей можно использовать выражение `stackalloc` только в объявлении локальной переменной для инициализации переменной.

Объем доступной памяти в стеке ограничен. При выделении слишком большого объема памяти в стеке возникает исключение StackOverflowException. Чтобы избежать этого, следуйте приведенным ниже правилам.

- Ограничьте объем памяти, выделенный `stackalloc`. Например, если предполагаемый размер буфера меньше определенного предела, то выделяется память в стеке. В противном случае используйте массив требуемой длины, как показано в следующем коде:

```
const int MaxStackLimit = 1024;
Span<byte> buffer = inputLength <= MaxStackLimit ? stackalloc byte[inputLength] : new
byte[inputLength];
```

> **NOTE**
> Поскольку объем доступной памяти на стеке зависит от среды, в которой выполняется код, при определении фактического предельного значения следует использовать консервативное значение.

- Старайтесь не использовать `stackalloc` в циклах. Выделяйте блок памяти за пределами цикла и используйте его повторно внутри цикла.

Содержимое только что выделенной памяти не определено. Его следует инициализировать перед использованием. Например, вы можете использовать метод Span<T>.Clear, который задает для всех элементов значение по умолчанию типа `T`.

Начиная с версии C# 7.3, вы можете использовать синтаксис инициализатора массива, чтобы определить содержимое для только что выделенной памяти. В следующем примере показано несколько способов сделать это:

```
Span<int> first = stackalloc int[3] { 1, 2, 3 };
Span<int> second = stackalloc int[] { 1, 2, 3 };
ReadOnlySpan<int> third = stackalloc[] { 1, 2, 3 };
```

В выражении `stackalloc T[E]` `T` должен иметь неуправляемый тип, а `E` — неотрицательное значение int.

Безопасность

При использовании `stackalloc` в среде CLR автоматически включается контроль переполнения буфера. Если буфер переполнен, процесс незамедлительно прерывается — это позволяет минимизировать риск

исполнения вредоносного кода.

Спецификация языка C#

См. сведения о выделении памяти в стеке в спецификации языка C# и разрешении `stackalloc` во вложенных контекстах в примечании.

См. также

- справочник по C#
- Операторы и выражения C#
- Операторы, связанные с указателем
- Типы указателей
- Типы, связанные с памятью и диапазонами
- Правила stackalloc

Выражение switch (справочник по C#)

18.12.2021 • 2 minutes to read

Начиная с версии C# 8.0, вы можете использовать `switch`, чтобы вычислить какое-то одно выражение из списка кандидатов, основываясь на сопоставлении шаблонов по входному выражению. Дополнительные сведения об операторе `switch`, который поддерживает семантику, аналогичную `switch`, в контексте оператора, см. в разделе об операторе `switch` статьи Операторы выбора.

В следующем примере демонстрируется выражение `switch`, которое преобразует значения `enum`, представляющие визуальные направления на интерактивной карте, в соответствующие стороны света.

```
public static class SwitchExample
{
    public enum Direction
    {
        Up,
        Down,
        Right,
        Left
    }

    public enum Orientation
    {
        North,
        South,
        East,
        West
    }

    public static Orientation ToOrientation(Direction direction) => direction switch
    {
        Direction.Up    => Orientation.North,
        Direction.Right => Orientation.East,
        Direction.Down  => Orientation.South,
        Direction.Left  => Orientation.West,
        _ => throw new ArgumentOutOfRangeException(nameof(direction), $"Not expected direction value:
{direction}"),
    };

    public static void Main()
    {
        var direction = Direction.Right;
        Console.WriteLine($"Map view direction is {direction}");
        Console.WriteLine($"Cardinal orientation is {ToOrientation(direction)}");
        // Output:
        // Map view direction is Right
        // Cardinal orientation is East
    }
}
```

В предыдущем примере показаны основные элементы выражения `switch`.

- Выражение, за которым следует ключевое слово `switch`. В предыдущем примере это параметр метода `direction`.
- *Ветви выражения* `switch`, разделенные запятыми. Каждая ветвь выражения `switch` содержит *шаблон*, необязательное *охранное условие*, маркер `=>` и *выражение*.

В предыдущем примере выражение `switch` использует следующие шаблоны.

- Шаблон константы для обработки определенных значений перечисления `Direction`.
- Шаблон пустой переменной используется для обработки любых целочисленных значений, которые не имеют соответствующих элементов в перечислении `Direction` (например, `(Direction)10`). Это делает выражение `switch` исчерпывающим.

> **IMPORTANT**
> Сведения о шаблонах, поддерживаемых выражением `switch` и дополнительные примеры см. в статье Шаблоны.

Результатом выражения `switch` является значение выражения первой ветви `switch`, чей шаблон соответствует входному выражению, а охранное условие, если таковое есть, принимает значение `true`. Ветви выражения `switch` вычисляются в том порядке, в котором они приведены в тексте.

Когда выбрать идущую ниже ветвь выражения `switch` невозможно, так как идущая выше ветвь `switch` соответствует всем значениям этого выражения, компилятор выдает ошибку.

Охранные условия

Шаблон может недостаточно полно выражать условие вычисления выражения ветви. В этом случае можно использовать охранное условие. Это дополнительное условие, которое должно выполняться помимо соответствия шаблону. Условие случая должно быть логическим выражением. Охранное условие указывается после ключевого слова `when`, идущего за шаблоном, как показано в следующем примере.

```
public readonly struct Point
{
    public Point(int x, int y) => (X, Y) = (x, y);

    public int X { get; }
    public int Y { get; }
}

static Point Transform(Point point) => point switch
{
    { X: 0, Y: 0 }                    => new Point(0, 0),
    { X: var x, Y: var y } when x < y => new Point(x + y, y),
    { X: var x, Y: var y } when x > y => new Point(x - y, y),
    { X: var x, Y: var y }            => new Point(2 * x, 2 * y),
};
```

В предыдущем примере используются шаблоны свойств с вложенными шаблонами var.

Неисчерпывающие выражения switch

Если ни один из шаблонов выражения `switch` не соответствует входному значению, среда выполнения выдает исключение. В .NET Core 3.0 и более поздних версиях исключением является System.Runtime.CompilerServices.SwitchExpressionException. В .NET Framework исключением является InvalidOperationException. Если выражение `switch` не обрабатывает все возможные входные значения, компилятор генерирует предупреждение.

> **TIP**
> Чтобы гарантировать, что выражение `switch` обработает все возможные входные значения, укажите ветвь выражения `switch` с шаблоном пустой переменной.

Спецификация языка C#

Дополнительные сведения см. в разделе о выражении `switch` в этой заметке о предлагаемых функциях.

См. также раздел

- справочник по C#
- Операторы и выражения C#
- Шаблоны
- Учебник. Использование сопоставления шаблонов для создания управляемых типами и управляемых данными алгоритмов
- Инструкция `switch`

Операторы true и false (справочник по C#)

18.12.2021 • 2 minutes to read

Оператор `true` возвращает значение bool `true`, указывая, что его операнд имеет значение true. Оператор `false` возвращает значение `bool``true`, указывая, что его операнд имеет значение false. Операторы `true` и `false` не обязательно дополняют друг друга. То есть оба оператора `true` и `false` могут вернуть значение `bool``false` для одного операнда. Если тип определяет один из двух операторов, он должен также определять другой оператор.

> **TIP**
>
> Используйте тип `bool?`, если вам нужно использовать трехзначную логику (например, при работе с базами данных, которые поддерживают трехзначный логический тип). В C# предоставляются операторы `&` и `|`, которые поддерживают троичную логику с операндами `bool?`. См. подробнее о логических операторах, поддерживающих значение NULL в описании логических операторов.

Логические выражения

Тип с определенным оператором `true` может быть типом результата управляющего условного выражения в операторах if, do, while и for и условном операторе `?:`. Дополнительные сведения см. в описании логических выражений в спецификации языка C#.

Пользовательские условные логические операторы

Если тип с определенными операторами `true` и `false` перегружает логические операторы OR `|` или AND `&` определенным образом, условные логические операторы OR `||` или AND `&&` соответственно могут вычисляться для операндов этого типа. Дополнительные сведения см. в разделе Пользовательские условные логические операторы в Спецификации языка C#.

Пример

В следующем примере представлен тип, который определяет операторы `true` и `false`. Этот тип также перегружает логический оператор И `&` таким образом, что оператор `&&` может быть вычислен для операндов этого типа.

```
using System;

public struct LaunchStatus
{
    public static readonly LaunchStatus Green = new LaunchStatus(0);
    public static readonly LaunchStatus Yellow = new LaunchStatus(1);
    public static readonly LaunchStatus Red = new LaunchStatus(2);

    private int status;

    private LaunchStatus(int status)
    {
        this.status = status;
    }

    public static bool operator true(LaunchStatus x) => x == Green || x == Yellow;
    public static bool operator false(LaunchStatus x) => x == Red;

    public static LaunchStatus operator &(LaunchStatus x, LaunchStatus y)
    {
        if (x == Red || y == Red || (x == Yellow && y == Yellow))
        {
            return Red;
        }

        if (x == Yellow || y == Yellow)
        {
            return Yellow;
        }

        return Green;
    }

    public static bool operator ==(LaunchStatus x, LaunchStatus y) => x.status == y.status;
    public static bool operator !=(LaunchStatus x, LaunchStatus y) => !(x == y);

    public override bool Equals(object obj) => obj is LaunchStatus other && this == other;
    public override int GetHashCode() => status;
}

public class LaunchStatusTest
{
    public static void Main()
    {
        LaunchStatus okToLaunch = GetFuelLaunchStatus() && GetNavigationLaunchStatus();
        Console.WriteLine(okToLaunch ? "Ready to go!" : "Wait!");
    }

    static LaunchStatus GetFuelLaunchStatus()
    {
        Console.WriteLine("Getting fuel launch status...");
        return LaunchStatus.Red;
    }

    static LaunchStatus GetNavigationLaunchStatus()
    {
        Console.WriteLine("Getting navigation launch status...");
        return LaunchStatus.Yellow;
    }
}
```

Обратите внимание на поведение сокращенного вычисления оператора `&&`. Когда метод `GetFuelLaunchStatus` возвращает `LaunchStatus.Red`, правый операнд оператора `&&` не вычисляется. Это обусловлено тем, что `LaunchStatus.Red` имеет значение false. Результат логического оператора И не зависит от значения правого операнда. Выходные данные примера могут быть следующими:

```
Getting fuel launch status...
Wait!
```

См. также раздел

- справочник по C#
- Операторы и выражения C#

Выражение with (справочник по C#)

18.12.2021 • 3 minutes to read

Выражение `with`, доступное в C# 9.0 и более поздних версиях, создает копию своего операнда с измененными заданными свойствами и полями:

```
using System;

public class WithExpressionBasicExample
{
    public record NamedPoint(string Name, int X, int Y);

    public static void Main()
    {
        var p1 = new NamedPoint("A", 0, 0);
        Console.WriteLine($"{nameof(p1)}: {p1}");  // output: p1: NamedPoint { Name = A, X = 0, Y = 0 }

        var p2 = p1 with { Name = "B", X = 5 };
        Console.WriteLine($"{nameof(p2)}: {p2}");  // output: p2: NamedPoint { Name = B, X = 5, Y = 0 }

        var p3 = p1 with
            {
                Name = "C",
                Y = 4
            };
        Console.WriteLine($"{nameof(p3)}: {p3}");  // output: p3: NamedPoint { Name = C, X = 0, Y = 4 }

        Console.WriteLine($"{nameof(p1)}: {p1}");  // output: p1: NamedPoint { Name = A, X = 0, Y = 0 }

        var apples = new { Item = "Apples", Price = "1.19" };
        Console.WriteLine($"original apples: {apples}");
        var saleApples = apples with { Price = "0.79" };
        Console.WriteLine($"sale apples: {saleApples}");

    }
}
```

Как показано в предыдущем примере, для указания элементов для изменения и их новых значений используется синтаксис инициализатора объектов.

В C# 9.0 левый операнд в выражении `with` должен иметь тип записи. Начиная с C# 10, левый операнд выражения `with` также может иметь тип структуры или анонимный тип.

Результат выражения `with` имеет тот же тип среды выполнения, что и операнд выражения, как показано в следующем примере:

```
using System;

public class InheritanceExample
{
    public record Point(int X, int Y);
    public record NamedPoint(string Name, int X, int Y) : Point(X, Y);

    public static void Main()
    {
        Point p1 = new NamedPoint("A", 0, 0);
        Point p2 = p1 with { X = 5, Y = 3 };
        Console.WriteLine(p2 is NamedPoint);  // output: True
        Console.WriteLine(p2);  // output: NamedPoint { X = 5, Y = 3, Name = A }

    }
}
```

При использовании элемента ссылочного типа копируется только ссылка на экземпляр элемента при копировании операнда. Как скопированный, так и исходный операнды имеют доступ к одному экземпляру ссылочного типа. В следующем примере продемонстрировано такое поведение.

```
using System;
using System.Collections.Generic;

public class ExampleWithReferenceType
{
    public record TaggedNumber(int Number, List<string> Tags)
    {
        public string PrintTags() => string.Join(", ", Tags);
    }

    public static void Main()
    {
        var original = new TaggedNumber(1, new List<string> { "A", "B" });

        var copy = original with { Number = 2 };
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B

        original.Tags.Add("C");
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B, C

    }
}
```

Семантика пользовательской копии

Любой тип класса записи имеет *конструктор копии*. Это конструктор с одним параметром содержащего типа записи. Он копирует состояние своего аргумента в новый экземпляр записи. При вычислении выражения `with` вызывается конструктор копий для создания нового экземпляра записи на основе исходной записи. После этого новый экземпляр обновляется в соответствии с указанными изменениями. По умолчанию конструктор копий является неявным, то есть созданным компилятором. Если необходимо настроить семантику копирования записей, явно объявите конструктор копии с требуемым поведением. В следующем примере предыдущий пример обновляется явным образом с помощью конструктора копии. Новое поведение копирования записи заключается в копировании элементов списка вместо ссылки на список:

```
using System;
using System.Collections.Generic;

public class UserDefinedCopyConstructorExample
{
    public record TaggedNumber(int Number, List<string> Tags)
    {
        protected TaggedNumber(TaggedNumber original)
        {
            Number = original.Number;
            Tags = new List<string>(original.Tags);
        }

        public string PrintTags() => string.Join(", ", Tags);
    }

    public static void Main()
    {
        var original = new TaggedNumber(1, new List<string> { "A", "B" });

        var copy = original with { Number = 2 };
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B

        original.Tags.Add("C");
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B
    }
}
```

Настроить семантику копирования для типов структуры нельзя.

Спецификация языка C#

Дополнительные сведения см. в следующих разделах примечания к предлагаемой функции записи:

- Выражение `with`
- Копирование и клонирование членов

См. также раздел

- справочник по C#
- Операторы и выражения C#
- Записи
- Типы структур

Перегрузка операторов (справочник по C#)

18.12.2021 • 3 minutes to read

Определяемый пользователем тип может перегружать предопределенный оператор C#. То есть тип может указать пользовательскую реализацию операции, если один или оба операнда принадлежат этому типу. В разделе Перегружаемые операторы показано, какие операторы C# можно перегружать.

Для объявления оператора используйте ключевое слово `operator`. Объявление оператора должно соответствовать следующим правилам:

- Оно должно включать `public` и модификатор `static`.
- У унарного оператора один входной параметр. У бинарного оператора два входных параметра. В каждом случае хотя бы один параметр должен иметь тип `T` или `T?`, где `T` — тип, который содержит объявление оператора.

В следующем примере определяется упрощенная структура, представляющая рациональное число. Структура перегружает некоторые арифметические операторы:

```
using System;

public readonly struct Fraction
{
    private readonly int num;
    private readonly int den;

    public Fraction(int numerator, int denominator)
    {
        if (denominator == 0)
        {
            throw new ArgumentException("Denominator cannot be zero.", nameof(denominator));
        }
        num = numerator;
        den = denominator;
    }

    public static Fraction operator +(Fraction a) => a;
    public static Fraction operator -(Fraction a) => new Fraction(-a.num, a.den);

    public static Fraction operator +(Fraction a, Fraction b)
        => new Fraction(a.num * b.den + b.num * a.den, a.den * b.den);

    public static Fraction operator -(Fraction a, Fraction b)
        => a + (-b);

    public static Fraction operator *(Fraction a, Fraction b)
        => new Fraction(a.num * b.num, a.den * b.den);

    public static Fraction operator /(Fraction a, Fraction b)
    {
        if (b.num == 0)
        {
            throw new DivideByZeroException();
        }
        return new Fraction(a.num * b.den, a.den * b.num);
    }

    public override string ToString() => $"{num} / {den}";
}

public static class OperatorOverloading
{
    public static void Main()
    {
        var a = new Fraction(5, 4);
        var b = new Fraction(1, 2);
        Console.WriteLine(-a);   // output: -5 / 4
        Console.WriteLine(a + b);  // output: 14 / 8
        Console.WriteLine(a - b);  // output: 6 / 8
        Console.WriteLine(a * b);  // output: 5 / 8
        Console.WriteLine(a / b);  // output: 10 / 4
    }
}
```

Вы можете расширить предыдущий пример, определив неявное преобразование из `int` в `Fraction`. Затем перегруженные операторы будут поддерживать аргументы этих двух типов. То есть можно будет добавить целое число к дроби и получить дробь.

Можно также использовать ключевое слово `operator` для определения пользовательского преобразования типа. Дополнительные сведения см. в разделе Операторы пользовательского преобразования.

Перегружаемые операторы

Следующая таблица содержит сведения о возможности перегрузки операторов C#:

ОПЕРАТОРЫ	ВОЗМОЖНОСТЬ ПЕРЕГРУЗКИ
+x, -x, !x, ~x, ++, --, true, false	Эти унарные операторы могут быть перегружены.
x + y, x - y, x * y, x / y, x % y, x & y, x | y, x ^ y, x << y, x >> y, x == y, x != y, x < y, x > y, x <= y, x >= y	Эти бинарные операторы могут быть перегружены. Некоторые операторы должны перегружаться парами; дополнительные сведения см. в примечаниях после этой таблицы.
x && y, x || y	Условный логический оператор не может быть перегружен. При этом, если тип с перегруженными операторами `true` и `false` также перегружает оператор `&` или `|`, оператор `&&` или `||`, соответственно, может быть применен для операндов этого типа. Дополнительные сведения см. в разделе Пользовательские условные логические операторы в Спецификации языка C#.
a[i], `a?[i]`	Доступ к элементам не считается перегружаемым оператором, но вы можете определить индексатор.
(T)x	Оператор приведения невозможно перегрузить, но можно определить пользовательские преобразования типа и выполнять его с помощью выражения приведения. Дополнительные сведения см. в разделе Операторы пользовательского преобразования.
+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=	Составные операторы присваивания не могут быть перегружены явным образом. Однако при перегрузке бинарного оператора соответствующий составной оператор присваивания (если таковой имеется) также неявно перегружается. Например, `+=` вычисляется с помощью `+`, который может быть перегружен.
^x, x = y, x.y, `x?.y`, c ? t : f, x ?? y, x ??= y, x..y, x->y, =>, f(x), as, await, checked, unchecked, default, delegate, is, nameof, new, sizeof, stackalloc, switch, typeof, with	Эти операторы не могут быть перегружены.

NOTE

Операторы сравнения должны перегружаться парами. То есть при перегрузке оператора из пары другой оператор тоже должен перегружаться. Ниже приведены эти пары:

- Операторы `==` и `!=`
- Операторы `<` и `>`
- Операторы `<=` и `>=`

Спецификация языка C#

Дополнительные сведения см. в следующих разделах статьи Спецификация языка C#:

- Перегрузка операторов
- Инструкции

См. также

- справочник по C#
- Операторы и выражения C#
- Операторы пользовательского преобразования
- Рекомендации по разработке. Перегрузки операторов
- Рекомендации по разработке. Операторы равенства
- Почему перегруженные операторы всегда являются статическими в C#?

Операторы итераций (справочник по C#)

18.12.2021 • 5 minutes to read

Следующие операторы многократно выполняют инструкцию или блок инструкций:

- Оператор `for`: выполняет тело, а заданное логическое выражение принимает значение `true`.
- Оператор `foreach`: перечисляет элементы коллекции и выполняет тело для каждого элемента коллекции.
- Оператор `do`: условно выполняет тело один или несколько раз.
- Оператор `while`: условно выполняет тело ноль или более раз.

Цикл можно прервать в любой момент в теле оператора итерации с помощью оператора break. Кроме того, можно перейти к следующей итерации в цикле с помощью оператора continue.

Инструкция `for`

Оператор `for` выполняет оператор или блок операторов, пока определенное логическое выражение равно значению `true`. В следующем примере показана инструкция `for`, выполняющая тело пока целочисленный счетчик меньше трех:

```
for (int i = 0; i < 3; i++)
{
    Console.Write(i);
}
// Output:
// 012
```

В предыдущем примере показаны элементы оператора `for`:

- Раздел *инициализатора*, который выполняется только один раз перед входом в цикл. Как правило, в этом разделе объявляется и инициализируется локальная переменная цикла. Доступ к объявленной переменной извне оператора `for` невозможен.

  В разделе *инициализатора* в предыдущем примере объявляется и инициализируется целочисленная переменная-счетчик:

  ```
  int i = 0
  ```

- Раздел *условия*, в котором определяется, следует ли выполнять следующую итерацию в цикле. Если для него получено значение `true` или значение отсутствует, выполняется следующая итерация; в противном случае цикл завершается. Раздел *условия* должен быть логическим выражением.

  В разделе *условия* в предыдущем примере проверяется, меньше ли трех значение счетчика.

  ```
  i < 3
  ```

- Раздел *итератора*, который определяет, что происходит после каждого выполнения тела цикла.

  Раздел *итератора* в предыдущем примере увеличивает значение счетчика:

```
i++
```

- Тело цикла которое должно быть оператором или блоком операторов.

Раздел итератора может содержать ноль или более следующих выражений оператора, разделенных запятыми:

- префиксное или постфиксное выражение приращения, такое как `++i` или `i++`
- префиксное или постфиксное выражение декремента, такое как `--i` или `i--`
- присваивание
- вызов метода
- выражение await
- создание объекта с помощью оператора new

Если переменная цикла не объявлена в разделе инициализатора, в разделе инициализатора можно также использовать ноль или более выражений из предыдущего списка. В следующем примере показано несколько менее распространенных вариантов использования разделов инициализатора и итератора: присваивание значения внешней переменной цикла в разделе инициализатора, вызов метода в разделах инициализатора и итератора и изменение значения двух переменных в разделе итератора.

```
int i;
int j = 3;
for (i = 0, Console.WriteLine($"Start: i={i}, j={j}"); i < j; i++, j--, Console.WriteLine($"Step: i={i}, j=
{j}"))
{
    //...
}
// Output:
// Start: i=0, j=3
// Step: i=1, j=2
// Step: i=2, j=1
```

Все разделы оператора `for` необязательны. Например, в следующем коде определяется бесконечный цикл `for`:

```
for ( ; ; )
{
    //...
}
```

Инструкция `foreach`

Оператор `foreach` выполняет оператор или блок операторов для каждого элемента в экземпляре типа, который реализует интерфейс System.Collections.IEnumerable или System.Collections.Generic.IEnumerable<T>, как показано в следующем примере.

```
var fibNumbers = new List<int> { 0, 1, 1, 2, 3, 5, 8, 13 };
foreach (int element in fibNumbers)
{
    Console.Write($"{element} ");
}
// Output:
// 0 1 1 2 3 5 8 13
```

Оператор `foreach` не ограничен этими типами. Его можно использовать с экземпляром любого типа, который удовлетворяет следующим условиям:

- Тип имеет открытый метод без параметров `GetEnumerator`. Начиная с C# 9.0 метод `GetEnumerator` может быть методом расширения типа.
- тип возвращаемого значения метода `GetEnumerator` должен содержать открытое свойство `Current` и открытый метод `MoveNext` без параметров с типом возвращаемого значения `bool`.

В следующем примере показано использование оператора `foreach` с экземпляром типа System.Span<T>, который не реализует интерфейс:

```
Span<int> numbers = new int[] { 3, 14, 15, 92, 6 };
foreach (int number in numbers)
{
    Console.Write($"{number} ");
}
// Output:
// 3 14 15 92 6
```

Начиная с версии C# 7.3, если свойство перечислителя `Current` возвращает ссылочное возвращаемое значение (`ref T`, где `T` — это тип элемента коллекции), вы можете объявить переменную итерации с модификатором `ref` или `ref readonly`, как показано в следующем примере.

```
Span<int> storage = stackalloc int[10];
int num = 0;
foreach (ref int item in storage)
{
    item = num++;
}
foreach (ref readonly var item in storage)
{
    Console.Write($"{item} ");
}
// Output:
// 0 1 2 3 4 5 6 7 8 9
```

Если оператор `foreach` применяется к `null`, возникает исключение NullReferenceException. Если исходная коллекция инструкции `foreach` пуста, тело оператора `foreach` не выполняется и пропускается.

await foreach

Начиная с C# 8.0, можно применять оператор `await foreach` для использования асинхронного потока данных, то есть типа коллекции, реализующего интерфейс IAsyncEnumerable<T>. Каждую итерацию цикла можно приостановить, пока будет осуществляться асинхронное извлечение следующего элемента. В следующем примере показано использование оператора `await foreach`.

```
await foreach (var item in GenerateSequenceAsync())
{
    Console.WriteLine(item);
}
```

Оператор `await foreach` можно также использовать с экземпляром любого типа, который удовлетворяет следующим условиям:

- Тип имеет открытый метод без параметров `GetAsyncEnumerator`. Этот метод может быть методом расширения типа.
- Тип возвращаемого значения метода `GetAsyncEnumerator` имеет открытое свойство `Current` и

открытый метод без параметров `MoveNextAsync`, тип возвращаемого значения которого — `Task<bool>`, `ValueTask<bool>` или любой другой подтверждающий ожидание тип, метод ожидания которого `GetResult` возвращает значение `bool`.

Элементы потока по умолчанию обрабатываются в захваченном контексте. Чтобы отключить захват контекста, используйте метод расширения TaskAsyncEnumerableExtensions.ConfigureAwait. Дополнительные сведения о контекстах синхронизации и захвате текущего контекста см. в статье Использование асинхронного шаблона, основанного на задачах. Дополнительные сведения об асинхронных потоках см. в разделе Асинхронные потоки статьи Новые возможности в C# 8.0.

Тип переменной итерации

Можно использовать ключевое слово `var`, чтобы компилятор мог определить тип переменной итерации в операторе `foreach`, как показано в следующем коде:

```
foreach (var item in collection) { }
```

Можно также явно указать тип переменной итерации, как показано в следующем коде:

```
IEnumerable<T> collection = new T[5];
foreach (V item in collection) { }
```

В предыдущей форме тип `T` элемента коллекции должен быть неявно или явно преобразован в тип `V` переменной итерации. Если явное преобразование из `T` в `V` завершается ошибкой во время выполнения, оператор `foreach` выдает исключение InvalidCastException. Например, если `T` является незапечатанным типом класса, `V` может быть любым типом интерфейса, даже тем, который `T` не реализует. Во время выполнения тип элемента коллекции может быть производным от `T` и фактически реализовать `V`. В противном случае возникает InvalidCastException.

Инструкция `do`

Оператор `do` выполняет оператор или блок операторов, пока определенное логическое выражение равно значению `true`. Так как это выражение оценивается после каждого выполнения цикла, цикл `do` выполняется один или несколько раз. Это отличает его от цикла while, который выполняется от нуля до нескольких раз.

В следующем примере показано применение оператора `do`.

```
int n = 0;
do
{
    Console.Write(n);
    n++;
} while (n < 5);
// Output:
// 01234
```

Инструкция `while`

Оператор `while` выполняет оператор или блок операторов, пока определенное логическое выражение равно значению `true`. Так как это выражение оценивается перед каждым выполнением цикла, цикл `while` выполняется ноль или несколько раз. Это отличает его от цикла do, который выполняется от одного до нескольких раз.

В следующем примере показано применение оператора `while`.

```
int n = 0;
while (n < 5)
{
    Console.Write(n);
    n++;
}
// Output:
// 01234
```

Спецификация языка C#

Дополнительные сведения см. в следующих разделах статьи Спецификация языка C#:

- Оператор `for`
- Оператор `foreach`
- Оператор `do`
- Оператор `while`

Дополнительные сведения о функциях, добавленных в C# 8.0 и более поздние версии, см. в следующих заметках о функциях.

- Асинхронные потоки (C# 8.0)
- Поддержка расширения `GetEnumerator` для циклов `foreach` (C# 9.0)

См. также

- справочник по C#
- Использование оператора foreach с массивами
- Итераторы

Операторы выбора. Справочник по C#

18.12.2021 • 4 minutes to read

Следующие операторы выбирают операторы, которые нужно выполнить, из нескольких возможных вариантов на основе значения выражения:

- Оператор `if` выбирает оператор, который нужно выполнить, на основе значения логического выражения.
- Оператор `switch` выбирает список операторов, который нужно выполнить, на основе соответствия шаблона выражению.

Инструкция `if`

Оператор `if` может быть в любой из следующих двух форм:

- Оператор `if` с частью `else` выбирает для выполнения один из двух операторов на основе значения логического выражения, как показано в следующем примере:

```
DisplayWeatherReport(15.0);  // Output: Cold.
DisplayWeatherReport(24.0);  // Output: Perfect!

void DisplayWeatherReport(double tempInCelsius)
{
    if (tempInCelsius < 20.0)
    {
        Console.WriteLine("Cold.");
    }
    else
    {
        Console.WriteLine("Perfect!");
    }
}
```

- Оператор `if` без части `else` выполняется, только если логическое выражение имеет значение `true`, как показано в следующем примере:

```
DisplayMeasurement(45);  // Output: The measurement value is 45
DisplayMeasurement(-3);  // Output: Warning: not acceptable value! The measurement value is -3

void DisplayMeasurement(double value)
{
    if (value < 0 || value > 100)
    {
        Console.Write("Warning: not acceptable value! ");
    }

    Console.WriteLine($"The measurement value is {value}");
}
```

Операторы `if` можно вкладывать для проверки нескольких условий, как показано в следующем примере:

```
DisplayCharacter('f');  // Output: A lowercase letter: f
DisplayCharacter('R');  // Output: An uppercase letter: R
DisplayCharacter('8');  // Output: A digit: 8
DisplayCharacter(',');  // Output: Not alphanumeric character: ,

void DisplayCharacter(char ch)
{
    if (char.IsUpper(ch))
    {
        Console.WriteLine($"An uppercase letter: {ch}");
    }
    else if (char.IsLower(ch))
    {
        Console.WriteLine($"A lowercase letter: {ch}");
    }
    else if (char.IsDigit(ch))
    {
        Console.WriteLine($"A digit: {ch}");
    }
    else
    {
        Console.WriteLine($"Not alphanumeric character: {ch}");
    }
}
```

В контексте выражения можно использовать условный оператор `?:` для вычисления одного из двух выражений на основе значения логического выражения.

Инструкция `switch`

Оператор `switch` выбирает список операторов, который нужно выполнить, на основе соответствия шаблона выражению соответствия, как показано в следующем примере:

```
DisplayMeasurement(-4);  // Output: Measured value is -4; too low.
DisplayMeasurement(5);  // Output: Measured value is 5.
DisplayMeasurement(30);  // Output: Measured value is 30; too high.
DisplayMeasurement(double.NaN);  // Output: Failed measurement.

void DisplayMeasurement(double measurement)
{
    switch (measurement)
    {
        case < 0.0:
            Console.WriteLine($"Measured value is {measurement}; too low.");
            break;

        case > 15.0:
            Console.WriteLine($"Measured value is {measurement}; too high.");
            break;

        case double.NaN:
            Console.WriteLine("Failed measurement.");
            break;

        default:
            Console.WriteLine($"Measured value is {measurement}.");
            break;
    }
}
```

В предыдущем примере оператор `switch` использует следующие шаблоны:

- Реляционные шаблоны — для сравнения результата выражения с константой.
- Шаблон константы — для проверки того, равен ли результат выражения константе.

> **IMPORTANT**
>
> Сведения о шаблонах, поддерживаемых оператором `switch`, см. в статье Шаблоны.

В предыдущем примере также показана ветвь `default`. Ветвь `default` указывает операторы, которые нужно выполнить, если выражение не соответствует ни одному другому шаблону. Если выражение не соответствует ни одному шаблону, а ветви `default` нет, управление передается оператору `switch`.

Оператор `switch` выполняет *список операторов* в первом *разделе switch*, *шаблон ветви* которого соответствует выражению соответствия, а условие ветви, если оно есть, равно `true`. Оператор `switch` оценивает шаблоны ветвей в порядке, в котором они указаны в коде, сверху вниз. Компилятор создает ошибку, если оператор `switch` содержит недостижимую ветвь. Это ветвь, которая уже была обработана в верхнем регистре или шаблон которой невозможно сопоставить.

> **NOTE**
>
> Ветвь `default` может находиться в любом месте в операторе `switch`. Независимо от ее расположения ветвь `default` всегда вычисляется последней и только в том случае, если не будет совпадений ни с одним из других шаблонов ветвей.

Можно указать несколько шаблонов ветвей для одного раздела оператора `switch`, как показано в следующем примере:

```
DisplayMeasurement(-4);  // Output: Measured value is -4; out of an acceptable range.
DisplayMeasurement(50);  // Output: Measured value is 50.
DisplayMeasurement(132);  // Output: Measured value is 132; out of an acceptable range.

void DisplayMeasurement(int measurement)
{
    switch (measurement)
    {
        case < 0:
        case > 100:
            Console.WriteLine($"Measured value is {measurement}; out of an acceptable range.");
            break;

        default:
            Console.WriteLine($"Measured value is {measurement}.");
            break;
    }
}
```

Внутри оператора `switch` управление не может передаваться от одного раздела switch к следующему. Как показано в примерах этого раздела, обычно в конце каждого раздела switch используется оператор `break` для передачи управления из оператора `switch`. Вы также можете использовать операторы return и throw, чтобы передать управление из оператора `switch`. Чтобы имитировать такое поведение ветви и передать управление другому разделу switch, можно использовать оператор `goto`.

В контексте оператора можно использовать оператор `switch`, чтобы вычислить одно выражение из списка кандидатов на основе сопоставления шаблона с выражением.

Охранные условия

Шаблон ветви может недостаточно полно выражать условие выполнения раздела switch. В этом случае

можно использовать *условие ветви*. Это дополнительное условие, которое должно выполняться помимо соответствия шаблону. Условие случая должно быть логическим выражением. Охранное условие указывается после ключевого слова `when`, идущего за шаблоном, как показано в следующем примере.

```
DisplayMeasurements(3, 4);  // Output: First measurement is 3, second measurement is 4.
DisplayMeasurements(5, 5);  // Output: Both measurements are valid and equal to 5.

void DisplayMeasurements(int a, int b)
{
    switch ((a, b))
    {
        case (> 0, > 0) when a == b:
            Console.WriteLine($"Both measurements are valid and equal to {a}.");
            break;

        case (> 0, > 0):
            Console.WriteLine($"First measurement is {a}, second measurement is {b}.");
            break;

        default:
            Console.WriteLine("One or both measurements are not valid.");
            break;
    }
}
```

В предыдущем примере используются шаблоны положений со вложенными шаблонами отношений.

Поддержка версий языка

Начиная с C# 7.0, оператор `switch` поддерживает сопоставление шаблонов.

В C# 6 и более ранних версиях используется оператор `switch` со следующими ограничениями:

- Выражение соответствия должно иметь один из следующих типов: char, string, bool, целочисленный числовой тип или тип перечисления.
- В метках `case` допускаются только константные выражения.

Спецификация языка C#

Дополнительные сведения см. в следующих разделах статьи Спецификация языка C#:

- Оператор `if`
- Оператор `switch`

Дополнительные сведения о функциях, появившихся в C# 7.0 и более поздних версиях, см. в следующих заметках о функциях:

- Оператор switch (сопоставление шаблонов для C# 7.0)

См. также

- справочник по C#
- Условный оператор `?:`
- Логические операторы
- Шаблоны
- Выражение `switch`

Операторы перехода (Справочник по C#)

18.12.2021 • 5 minutes to read

Следующие операторы передают управление без дополнительных условий:

- Оператор `break` завершает выполнение ближайшего внешнего оператора итерации или оператора `switch`.
- Оператор `continue` начинает новую итерацию ближайшего внешнего оператора итерации.
- Оператор `return` завершает выполнение метода, в котором он присутствует, и возвращает управление вызывавшему методу.
- Оператор `goto` передает управление оператору, помеченному меткой.

Сведения об операторе `throw`, который создает исключение и без дополнительных условий передает управление, см. здесь.

Инструкция `break`

Оператор `break` завершает выполнение ближайшего внешнего оператора итерации (то есть цикла `for`, `foreach`, `while` или `do`) или оператора `switch`. Оператор `break` передает управление оператору, который расположен после завершенного оператора (если таковой есть).

```
int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
foreach (int number in numbers)
{
    if (number == 3)
    {
        break;
    }

    Console.Write($"{number} ");
}
Console.WriteLine();
Console.WriteLine("End of the example.");
// Output:
// 0 1 2
// End of the example.
```

Во вложенных циклах оператор `break` завершает только самый внутренний цикл, содержащий его, как показано в следующем примере:

```
for (int outer = 0; outer < 5; outer++)
{
    for (int inner = 0; inner < 5; inner++)
    {
        if (inner > outer)
        {
            break;
        }

        Console.Write($"{inner} ");
    }
    Console.WriteLine();
}
// Output:
// 0
// 0 1
// 0 1 2
// 0 1 2 3
// 0 1 2 3 4
```

При использовании оператора `switch` внутри цикла оператор `break` в конце раздела switch передает управление только из оператора `switch`. Цикл, содержащий оператор `switch`, не затрагивается, как показано в следующем примере:

```
double[] measurements = { -4, 5, 30, double.NaN };
foreach (double measurement in measurements)
{
    switch (measurement)
    {
        case < 0.0:
            Console.WriteLine($"Measured value is {measurement}; too low.");
            break;

        case > 15.0:
            Console.WriteLine($"Measured value is {measurement}; too high.");
            break;

        case double.NaN:
            Console.WriteLine("Failed measurement.");
            break;

        default:
            Console.WriteLine($"Measured value is {measurement}.");
            break;
    }
}
// Output:
// Measured value is -4; too low.
// Measured value is 5.
// Measured value is 30; too high.
// Failed measurement.
```

Инструкция `continue`

Оператор `continue` начинает новую итерацию ближайшего внешнего оператора итерации (то есть цикла `for`, `foreach`, `while` или `do`), как показано в следующем примере:

```
for (int i = 0; i < 5; i++)
{
    Console.Write($"Iteration {i}: ");

    if (i < 3)
    {
        Console.WriteLine("skip");
        continue;
    }

    Console.WriteLine("done");
}
// Output:
// Iteration 0: skip
// Iteration 1: skip
// Iteration 2: skip
// Iteration 3: done
// Iteration 4: done
```

Инструкция `return`

Оператор `return` завершает выполнение метода, в котором он присутствует, и возвращает управление и результат функции (при его наличии) вызывавшему методу.

Если член функции не вычисляет значение, используется оператор `return` без выражения, как показано в следующем примере:

```
Console.WriteLine("First call:");
DisplayIfNecessary(6);

Console.WriteLine("Second call:");
DisplayIfNecessary(5);

void DisplayIfNecessary(int number)
{
    if (number % 2 == 0)
    {
        return;
    }

    Console.WriteLine(number);
}
// Output:
// First call:
// Second call:
// 5
```

Как показано в предыдущем примере, для досрочного завершения члена функции обычно используется оператор `return` без выражения. Если член функции не содержит оператор `return`, он завершается после выполнения последнего оператора.

Если член функции вычисляет значение, используется оператор `return` с выражением, как показано в следующем примере:

```
double surfaceArea = CalculateCylinderSurfaceArea(1, 1);
Console.WriteLine($"{surfaceArea:F2}"); // output: 12.57

double CalculateCylinderSurfaceArea(double baseRadius, double height)
{
    double baseArea = Math.PI * baseRadius * baseRadius;
    double sideArea = 2 * Math.PI * baseRadius * height;
    return 2 * baseArea + sideArea;
}
```

Если оператор `return` содержит выражение, это выражение должно допускать неявное преобразование в тип возвращаемого значения члена функции, если только он не является асинхронным. В случае с `async` функцией выражение должно допускать неявное преобразование в аргумент типа Task<TResult> или ValueTask<TResult> в зависимости от типа возвращаемого значения функции. Если тип возвращаемого значения функции `async` — Task или ValueTask, используется оператор `return` без выражения.

По умолчанию оператор `return` возвращает значение выражения. Начиная с версии C# 7.0 можно возвращать ссылку на переменную. Для этого используйте оператор `return` с ключевым словом `ref`, как показано в следующем примере:

```
var xs = new int[] { 10, 20, 30, 40 };
ref int found = ref FindFirst(xs, s => s == 30);
found = 0;
Console.WriteLine(string.Join(" ", xs));  // output: 10 20 0 40

ref int FindFirst(int[] numbers, Func<int, bool> predicate)
{
    for (int i = 0; i < numbers.Length; i++)
    {
        if (predicate(numbers[i]))
        {
            return ref numbers[i];
        }
    }
    throw new InvalidOperationException("No element satisfies the given condition.");
}
```

Дополнительные сведения о возвращаемых ссылках см. в статье Возвращаемые ссылочные значения и ссылочные локальные переменные.

Инструкция `goto`

Оператор `goto` передает управление оператору, помеченному меткой, как показано в следующем примере:

```
var matrices = new Dictionary<string, int[][]>
{
    ["A"] = new[]
    {
        new[] { 1, 2, 3, 4 },
        new[] { 4, 3, 2, 1 }
    },
    ["B"] = new[]
    {
        new[] { 5, 6, 7, 8 },
        new[] { 8, 7, 6, 5 }
    },
};

CheckMatrices(matrices, 4);

void CheckMatrices(Dictionary<string, int[][]> matrixLookup, int target)
{
    foreach (var (key, matrix) in matrixLookup)
    {
        for (int row = 0; row < matrix.Length; row++)
        {
            for (int col = 0; col < matrix[row].Length; col++)
            {
                if (matrix[row][col] == target)
                {
                    goto Found;
                }
            }
        }
        Console.WriteLine($"Not found {target} in matrix {key}.");
        continue;

    Found:
        Console.WriteLine($"Found {target} in matrix {key}.");
    }
}
// Output:
// Found 4 in matrix A.
// Not found 4 in matrix B.
```

Как показано в предыдущем примере, для выхода из вложенного цикла можно использовать оператор `goto`.

TIP

При работе с вложенными циклами рассмотрите возможность рефакторинга отдельных циклов в отдельные методы. В результате вы можете получить упрощенный, более удобочитаемый код без оператора `goto`.

В операторе `switch` можно также использовать оператор `goto` для передачи управления в раздел switch с меткой константы case, как показано в следующем примере:

```
using System;

public enum CoffeChoice
{
    Plain,
    WithMilk,
    WithIceCream,
}

public class GotoInSwitchExample
{
    public static void Main()
    {
        Console.WriteLine(CalculatePrice(CoffeChoice.Plain));  // output: 10.0
        Console.WriteLine(CalculatePrice(CoffeChoice.WithMilk));  // output: 15.0
        Console.WriteLine(CalculatePrice(CoffeChoice.WithIceCream));  // output: 17.0
    }

    private static decimal CalculatePrice(CoffeChoice choice)
    {
        decimal price = 0;
        switch (choice)
        {
            case CoffeChoice.Plain:
                price += 10.0m;
                break;

            case CoffeChoice.WithMilk:
                price += 5.0m;
                goto case CoffeChoice.Plain;

            case CoffeChoice.WithIceCream:
                price += 7.0m;
                goto case CoffeChoice.Plain;
        }
        return price;
    }
}
```

В операторе `switch` можно также использовать оператор `goto default;` для передачи управления в раздел switch с меткой `default`.

Если метка с данным именем не существует в текущем члене функции или если оператор `goto` находится вне области метки, возникает ошибка времени компиляции. Это значит, что оператор `goto` нельзя использовать для передачи управления из текущего члена функции или во вложенную область, например в блок `try`.

Спецификация языка C#

Дополнительные сведения см. в следующих разделах статьи Спецификация языка C#:

- Оператор `break`
- Оператор `continue`
- Оператор `return`
- Оператор `goto`

См. также

- справочник по C#
- Инструкция `yield`

Специальные символы в C#

18.12.2021 • 2 minutes to read

Специальные символы — это стандартные контекстно-зависимые символы, которые изменяют элемент программы (строковый литерал, идентификатор или имя атрибута), к которому они добавляются. C# поддерживает следующие специальные символы:

- @, символ буквального идентификатора.
- $, символ интерполируемой строки.

Этот раздел включает только токены, не являющиеся операторами. См. раздел Операторы, чтобы просмотреть список всех операторов.

См. также

- Справочник по C#
- Руководство по программированию на C#

$ — интерполяция строк (справочные материалы по C#)

18.12.2021 • 5 minutes to read

Специальный знак `$` идентифицирует строковый литерал как *интерполированную строку*. Интерполированная строка — это строковый литерал, который может содержать *выражения интерполяции*. При разрешении интерполированной строки в результирующую элементы с выражениями интерполяции заменяются строковыми представлениями результатов выражений. Эта функция доступна начиная с C# 6.

Интерполяция строк предоставляет легко читаемый и удобный синтаксис для форматирования строк. Его проще читать, чем составное форматирование строк. Сравните следующий пример, в котором используются обе эти возможности для получения одинаковых выходных данных:

```
string name = "Mark";
var date = DateTime.Now;

// Composite formatting:
Console.WriteLine("Hello, {0}! Today is {1}, it's {2:HH:mm} now.", name, date.DayOfWeek, date);
// String interpolation:
Console.WriteLine($"Hello, {name}! Today is {date.DayOfWeek}, it's {date:HH:mm} now.");
// Both calls produce the same output that is similar to:
// Hello, Mark! Today is Wednesday, it's 19:40 now.
```

Структура интерполированной строки

Для определения строкового литерала в качестве интерполированной строки добавьте к началу символ `$`. Между `$` и `"` в начале строкового литерала не может быть пробела.

Структура элемента с выражением интерполяции выглядит следующим образом:

```
{<interpolationExpression>[,<alignment>][:<formatString>]}
```

Элементы в квадратных скобках являются необязательными. Каждый из элементов описан в таблице ниже:

ЭЛЕМЕНТ	ОПИСАНИЕ
`interpolationExpression`	Выражение, создающее форматируемый результат. Строковое представление `null` является String.Empty.
`alignment`	Константное выражение, значение которого определяет минимальное количество символов в строковом представлении результата выражения. Если оно положительное, строковое представление выравнивается по правому краю, если отрицательное — по левому краю. Дополнительные сведения см. в разделе Компонент выравнивания.

ЭЛЕМЕНТ	ОПИСАНИЕ
`formatString`	Строка форматирования, поддерживаемая типом результата выражения. Дополнительные сведения см. в разделе Компонент строки форматирования.

В следующем примере используются необязательные компоненты форматирования, описанные выше:

```
Console.WriteLine($"|{"Left",-7}|{"Right",7}|");

const int FieldWidthRightAligned = 20;
Console.WriteLine($"{Math.PI,FieldWidthRightAligned} - default formatting of the pi number");
Console.WriteLine($"{Math.PI,FieldWidthRightAligned:F3} - display only three decimal digits of the pi
number");
// Expected output is:
// |Left   |  Right|
//     3.14159265358979 - default formatting of the pi number
//                3.142 - display only three decimal digits of the pi number
```

Начиная с C# 10, вы можете использовать интерполяцию строк для инициализации постоянной строки. Все выражения, используемые для заполнителей, должны быть постоянными строками. Иными словами, все *выражения интерполяции* должны быть строками и константами времени компиляции.

Специальные символы

Чтобы включить в текст, создаваемый интерполированной строкой, фигурную скобку "{" или "}", используйте две фигурные скобки — "{{" или "}}". Подробнее см. в разделе Экранирование фигурных скобок.

Так как двоеточие (:) имеет особое значение в элементе выражения интерполяции, чтобы использовать условный оператор в выражении интерполяции, заключите это выражение в круглые скобки.

В следующем примере показано добавление фигурной скобки в строку результатов. В нем также показано, как использовать условный оператор:

```
string name = "Horace";
int age = 34;
Console.WriteLine($"He asked, \"Is your name {name}?\", but didn't wait for a reply :-{{");
Console.WriteLine($"{name} is {age} year{(age == 1 ? "" : "s")} old.");
// Expected output is:
// He asked, "Is your name Horace?", but didn't wait for a reply :-{
// Horace is 34 years old.
```

Интерполированная строка verbatim начинается с символа `$`, за которым следует символ `@`. Дополнительные сведения о буквальных строках см. в статьях о строках и буквальном идентификаторе.

> **NOTE**
> Начиная с C# 8.0 маркеры `$` и `@` можно использовать в любом порядке: `$@"..."` и `@$"..."` являются допустимыми интерполированными строками verbatim. В более ранних версиях C# маркер `$` должен располагаться перед маркером `@`.

Неявные преобразования и указание реализации `IFormatProvider`

Существует три неявных преобразования из интерполированных строк.

1. Преобразование интерполированной строки в экземпляр String. Строка является результатом разрешения интерполированной строки. Все элементы выражений интерполяции заменяются правильно форматированными представлениями строк их результатов. Это преобразование использует CurrentCulture для форматирования результатов выражений.

2. Преобразование интерполированной строки в экземпляр FormattableString, представляющий строку составного формата, а также форматируемые результаты выражения. Это позволяет создавать несколько результирующих строк с содержимым для конкретного языка из одного экземпляра FormattableString. Для этого вызовите один из следующих методов:

   - Перегрузка ToString(), которая формирует результирующую строку для CurrentCulture.
   - Метод Invariant, который формирует результирующую строку для InvariantCulture.
   - Метод ToString(IFormatProvider), который формирует результирующую строку для указанного языка и региональных параметров.

   Метод ToString(IFormatProvider) предоставляет пользовательские реализации интерфейса IFormatProvider, который поддерживает настраиваемое форматирование. Дополнительные сведения см. в статье Типы форматирования в .NET, разделе Настраиваемое форматирование с использованием интерфейса ICustomFormatter.

3. Преобразование интерполированной строки в экземпляр IFormattable, который также позволяет создавать несколько результирующих строк с содержимым для конкретного языка из одного экземпляра IFormattable.

В следующем примере используется неявное преобразование в FormattableString для создания результирующих строк для конкретного языка:

```
double speedOfLight = 299792.458;
FormattableString message = $"The speed of light is {speedOfLight:N3} km/s.";

System.Globalization.CultureInfo.CurrentCulture = System.Globalization.CultureInfo.GetCultureInfo("nl-NL");
string messageInCurrentCulture = message.ToString();

var specificCulture = System.Globalization.CultureInfo.GetCultureInfo("en-IN");
string messageInSpecificCulture = message.ToString(specificCulture);

string messageInInvariantCulture = FormattableString.Invariant(message);

Console.WriteLine($"{System.Globalization.CultureInfo.CurrentCulture,-10} {messageInCurrentCulture}");
Console.WriteLine($"{specificCulture,-10} {messageInSpecificCulture}");
Console.WriteLine($"{"Invariant",-10} {messageInInvariantCulture}");
// Expected output is:
// nl-NL      The speed of light is 299.792,458 km/s.
// en-IN      The speed of light is 2,99,792.458 km/s.
// Invariant  The speed of light is 299,792.458 km/s.
```

Дополнительные ресурсы

Если у вас нет опыта работы с интерполяцией строк, ознакомьтесь с интерактивным учебником по интерполяции строк в C#. Вы можете также просмотреть другой учебник по интерполяции строк в C#. В этом учебнике показано, как использовать интерполированные строки для создания форматированных строк.

Компиляция интерполированных строк

Если интерполированная строка имеет тип `string`, обычно она преобразуется в вызов метода String.Format. Компилятор может заменить String.Format на String.Concat, если проанализированное поведение будет эквивалентно объединению.

Если интерполированная строка имеет тип IFormattable или FormattableString, компилятор создает вызов метода FormattableStringFactory.Create.

Начиная с C# 10 при использовании интерполированной строки компилятор проверяет, назначена ли эта строка типу, отвечающему требованиям *шаблона обработчика интерполированных строк*. *Обработчик интерполированных строк* — это настраиваемый тип, который преобразует интерполированную строку в строку. Обработчик интерполированных строк — это расширенный сценарий, который обычно используется для повышения производительности. Вы можете узнать о требованиях для создания обработчика интерполированных строк в спецификации языка для улучшения интерполированных строк. Вы можете создать одну из них, выполнив инструкции из учебника по обработчику интерполированных строк в разделе C# "Новые возможности". В .NET 6 при использовании интерполированной строки для аргумента типа `string` эта строка обрабатывается с помощью System.Runtime.CompilerServices.DefaultInterpolatedStringHandler.

> **NOTE**
> Одним из побочных эффектов обработчиков интерполированных строк является то, что пользовательский обработчик, включая System.Runtime.CompilerServices.DefaultInterpolatedStringHandler, может не оценивать все выражения, используемые в качестве заполнителей в интерполированной строке при всех условиях. Это означает, что побочные эффекты в этих выражениях могут не возникнуть.

Спецификация языка C#

Дополнительные сведения см. в разделе Интерполированные строкиспецификации языка C#.

См. также

- справочник по C#
- Специальные символы C#
- Строки
- Строки стандартных числовых форматов
- Составное форматирование
- String.Format

@ (справочник по C#)

18.12.2021 • 2 minutes to read

Специальный символ `@` является буквальным идентификатором. Его можно применять в следующих случаях.

1. Чтобы использовать ключевые слова C# в качестве идентификаторов. Символ `@` предшествует элементу кода, который компилятор должен интерпретировать как идентификатор, а не ключевое слово C#. В следующем примере символ `@` используется для определения идентификатора `for`, используемого в цикле `for`.

   ```
   string[] @for = { "John", "James", "Joan", "Jamie" };
   for (int ctr = 0; ctr < @for.Length; ctr++)
   {
      Console.WriteLine($"Here is your gift, {@for[ctr]}!");
   }
   // The example displays the following output:
   //     Here is your gift, John!
   //     Here is your gift, James!
   //     Here is your gift, Joan!
   //     Here is your gift, Jamie!
   ```

2. Чтобы указать, что строковый литерал следует интерпретировать буквально. Символ `@` в этом случае определяет *буквальный строковый литерал*. Простые escape-последовательности (например, `"\\"` для обратной косой черты), шестнадцатеричные escape-последовательности (например, `"\x0041"` для прописной буквы A) и escape-последовательности Юникода (например, `"\u0041"` для прописной буквы A) интерпретируются буквально. Только escape-последовательность кавычек (`""`) не интерпретируется буквально. Она создает одну двойную кавычку. Кроме того, в случае буквальной интерполированной строки escape-последовательности фигурных скобок (`{{` и `}}`) не интерпретируются буквально; они создают одну фигурную скобку. В следующем примере определяются два идентичных пути к файлам: один с помощью обычного строкового литерала, а другой — с помощью буквального строкового литерала. Это один из наиболее распространенных способов использования буквальных строковых литералов.

   ```
   string filename1 = @"c:\documents\files\u0066.txt";
   string filename2 = "c:\\documents\\files\\u0066.txt";

   Console.WriteLine(filename1);
   Console.WriteLine(filename2);
   // The example displays the following output:
   //     c:\documents\files\u0066.txt
   //     c:\documents\files\u0066.txt
   ```

   В следующем примере показан результат определения обычного строкового литерала и буквального строкового литерала, содержащий идентичные последовательности символов.

```
string s1 = "He said, \"This is the last \u0063hance\x0021\"";
string s2 = @"He said, ""This is the last \u0063hance\x0021""";

Console.WriteLine(s1);
Console.WriteLine(s2);
// The example displays the following output:
//     He said, "This is the last chance!"
//     He said, "This is the last \u0063hance\x0021"
```

3. Чтобы позволить компилятору различать атрибуты в случае конфликта имен. Атрибут класса, который наследует от Attribute. Имя его типа, как правило, содержит суффикс **Attribute**, несмотря на то, что компилятор не выполняет это соглашение. Затем на атрибут можно ссылаться в коде либо по его полному имени типа (например, `[InfoAttribute]`), либо по сокращенному имени (например, `[Info]`). Однако если два сокращенных имени типов совпадают и одно имя типа содержит суффикс **Attribute**, а другое — нет, возникает конфликт имен. Например, приведенный ниже код не может быть скомпилирован, так как компилятору не удается определить, какой атрибут — `Info` или `InfoAttribute` — применен к классу `Example`. См. CS1614 для получения дополнительных сведений.

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class Info : Attribute
{
   private string information;

   public Info(string info)
   {
      information = info;
   }
}

[AttributeUsage(AttributeTargets.Method)]
public class InfoAttribute : Attribute
{
   private string information;

   public InfoAttribute(string info)
   {
      information = info;
   }
}

[Info("A simple executable.")] // Generates compiler error CS1614. Ambiguous Info and InfoAttribute.
// Prepend '@' to select 'Info' ([@Info("A simple executable.")]). Specify the full name
'InfoAttribute' to select it.
public class Example
{
   [InfoAttribute("The entry point.")]
   public static void Main()
   {
   }
}
```

См. также

- Справочник по C#
- Руководство по программированию на C#
- Специальные символы в C#

Зарезервированные атрибуты: атрибуты уровня сборки

18.12.2021 • 2 minutes to read

Большинство атрибутов применяется к определенным элементам языка, таким как классы или методы. Тем не менее некоторые атрибуты являются глобальными — они применяются ко всей сборке или модулю. Например, атрибут AssemblyVersionAttribute можно использовать для встраивания сведений о версии в сборку, например следующим образом:

```
[assembly: AssemblyVersion("1.0.0.0")]
```

Глобальные атрибуты отображаются в исходном коде после любых директив `using` верхнего уровня и перед всеми объявлениями типов, модулей или пространств имен. Глобальные атрибуты могут содержаться в нескольких исходных файлах, однако эти файлы должны быть скомпилированы за один проход компиляции. В Visual Studio глобальные атрибуты добавляются в файл AssemblyInfo.cs в проектах .NET Framework. Эти атрибуты не добавляются в проекты .NET Core.

Атрибуты сборки — это значения, которые предоставляют сведения о сборке. Они делятся на следующие категории:

- Атрибуты удостоверения сборки
- Информационные атрибуты
- Атрибуты манифеста сборки

Атрибуты удостоверения сборки

Три атрибута (со строгим именем, если оно применимо) определяют удостоверение сборки: имя, версию, язык и региональные параметры. Эти атрибуты формируют полное имя сборки и являются обязательными при ссылке на нее в коде. Атрибуты можно использовать для задания версии сборки и языка и региональных параметров. Однако значение имени задается компилятором, в интегрированной среде разработки Visual Studio, в диалоговом окне сведений о сборке или в компоновщике сборок (Al.exe) при создании сборки. Имя сборки основано на манифесте сборки. Атрибут AssemblyFlagsAttribute указывает, могут ли сосуществовать несколько копий сборки.

В следующей таблице приведены атрибуты удостоверения.

АТРИБУТ	ЦЕЛЬ
AssemblyVersionAttribute	Задает версию сборки.
AssemblyCultureAttribute	Указывает, какой язык и региональные параметры поддерживает сборка.
AssemblyFlagsAttribute	Указывает, поддерживает ли сборка параллельное выполнение на одном компьютере, в одном процессе или в одном домене приложения.

Информационные атрибуты

Информационные атрибуты можно использовать для предоставления дополнительных сведений о компании или продукте в сборке. В следующей таблице показаны информационные атрибуты, определенные в пространстве имен System.Reflection.

АТРИБУТ	ЦЕЛЬ
AssemblyProductAttribute	Указывает имя продукта для манифеста сборки.
AssemblyTrademarkAttribute	Указывает товарный знак для манифеста сборки.
AssemblyInformationalVersionAttribute	Определяет сведения о версии для манифеста сборки.
AssemblyCompanyAttribute	Указывает имя организации для манифеста сборки.
AssemblyCopyrightAttribute	Определяет настраиваемый атрибут, задающий уведомление об авторских правах для манифеста сборки.
AssemblyFileVersionAttribute	Задает определенный номер версии для ресурса версии файла Win32.
CLSCompliantAttribute	Указывает, соответствует ли сборка спецификации CLS.

Атрибуты манифеста сборки

Атрибуты манифеста сборки можно использовать для предоставления сведений в манифесте сборки. К ним относится заголовок, описание, псевдоним по умолчанию и конфигурация. В следующей таблице показаны атрибуты манифеста сборки, определенные в пространстве имен System.Reflection.

АТРИБУТ	ЦЕЛЬ
AssemblyTitleAttribute	Указывает название сборки для манифеста сборки.
AssemblyDescriptionAttribute	Указывает описание сборки для манифеста сборки.
AssemblyConfigurationAttribute	Указывает конфигурацию сборки (например, "Розница" или "Отладка") для манифеста сборки.
AssemblyDefaultAliasAttribute	Определяет понятный псевдоним по умолчанию для манифеста сборки.

Зарезервированные атрибуты: определение сведений о вызывающем объекте

18.12.2021 • 3 minutes to read

С помощью информационных атрибутов можно получить сведения о вызывающем объекте метода. Можно получить путь к файлу исходного кода, номер строки в исходном коде и имя члена вызывающего объекта. Для получения этих сведений используются атрибуты, которые применяются к необязательным параметрам. Каждый необязательный параметр задает значение по умолчанию. В следующей таблице перечислены информационные атрибуты вызывающего объекта, которые определены в пространстве имен System.Runtime.CompilerServices:

АТРИБУТ	ОПИСАНИЕ	TYPE
CallerFilePathAttribute	Полный путь исходного файла, содержащего вызывающий объект. Это путь во время компиляции.	`String`
CallerLineNumberAttribute	Номер строки в исходном файле, из которого вызывается метод.	`Integer`
CallerMemberNameAttribute	Имя свойства или метода вызывающего объекта.	`String`
CallerArgumentExpressionAttribute	Строковое представление выражения аргумента.	`String`

Эта информация поможет вам с трассировкой и отладкой, а также созданием средств диагностики. В следующем примере показано, как использовать информационные атрибуты вызывающего объекта. При каждом вызове метода `TraceMessage` сведения о вызывающем объекте подставляются в качестве аргументов необязательных параметров.

```
public void DoProcessing()
{
    TraceMessage("Something happened.");
}

public void TraceMessage(string message,
        [CallerMemberName] string memberName = "",
        [CallerFilePath] string sourceFilePath = "",
        [CallerLineNumber] int sourceLineNumber = 0)
{
    Trace.WriteLine("message: " + message);
    Trace.WriteLine("member name: " + memberName);
    Trace.WriteLine("source file path: " + sourceFilePath);
    Trace.WriteLine("source line number: " + sourceLineNumber);
}

// Sample Output:
//  message: Something happened.
//  member name: DoProcessing
//  source file path: c:\Visual Studio Projects\CallerInfoCS\CallerInfoCS\Form1.cs
//  source line number: 31
```

Для каждого необязательного параметра указывается явное значение по умолчанию. Нельзя применять информационные атрибуты вызывающего объекта к параметрам, которые не были указаны как необязательные. Информационные атрибуты вызывающего объекта не делают параметр необязательным. Вместо этого они влияют на значение по умолчанию, которое передается, если аргумент был опущен. Информационные значения вызывающего объекта передаются как литералы в IL во время компиляции. В отличие от результатов свойства StackTrace для исключений, результаты не затрагиваются запутыванием кода. Можно явно передать необязательные аргументы, чтобы управлять сведениями о вызывающем объекте или скрывать сведения о вызывающем объекте.

Имена членов

Можно использовать атрибут `CallerMemberName`, чтобы не указывать имя члена в виде аргумента `String` вызываемому методу. Этот прием позволяет избежать проблемы, заключающейся в том, что **операция рефакторинга и переименования** не изменяет значений `String`. Это особенно полезно при выполнении следующих задач:

- Использование процедур трассировки и диагностики.
- Реализация интерфейса INotifyPropertyChanged при привязке данных. Этот интерфейс позволяет свойству объекта уведомлять привязанный элемент управления об изменении свойства. Элемент управления может отображать обновленные сведения. Если атрибут `CallerMemberName` не используется, необходимо указать имя свойства как литерал.

В следующей таблице представлены имена членов, возвращаемых при использовании атрибута `CallerMemberName`.

ФРАГМЕНТ КОДА, В ПРЕДЕЛАХ КОТОРОГО ПРОИСХОДЯТ ВЫЗОВЫ	РЕЗУЛЬТАТ ИМЕНИ ЧЛЕНА
Метод, свойство или событие	Имя метода, свойства или события, из которого происходил вызов.
Конструктор	Строка ".ctor"
Статический конструктор	Строка ".cctor"
Метод завершения	Строка "Finalize"
Определяемые пользователем операторы и преобразования	Созданное имя члена, например, "op_Addition".
Конструктора атрибута	Имя метода или свойства, к которому применяется атрибут. Если атрибут — любой элемент внутри члена (например, параметр, возвращаемое значение или параметр универсального типа), то результат — имя члена, который связан с этим элементом.
Нет содержащего члена (например, уровень сборки или атрибуты, примененные к типам)	Значение необязательного параметра по умолчанию.

Выражения аргументов

Если требуется, чтобы выражение передавалось в качестве аргумента, используйте System.Runtime.CompilerServices.CallerArgumentExpressionAttribute. Библиотеки диагностики могут предоставить дополнительные сведения о *выражениях*, передаваемых в аргументы. Предоставляя выражение, запускающее диагностику, помимо имени параметра разработчики получают

дополнительные сведения об условии, вызвавшем диагностику. Эти дополнительные сведения упрощают устранение проблемы.

В следующем примере показано, как можно предоставить подробные сведения об аргументе, если он является недопустимым:

```
public static void ValidateArgument(string parameterName, bool condition,
[CallerArgumentExpression("condition")] string? message=null)
{
    if (!condition)
    {
        throw new ArgumentException($"Argument failed validation: <{message}>", parameterName);
    }
}
```

Его можно вызвать, как показано в следующем примере:

```
public void Operation(Action func)
{
    Utilities.ValidateArgument(nameof(func), func is not null);
    func();
}
```

Выражение, используемое для `condition`, внедряется компилятором в аргумент `message`. Когда разработчик вызывает `Operation` с аргументом `null`, следующее сообщение сохраняется в `ArgumentException`:

```
Argument failed validation: <func is not null>
```

Этот атрибут позволяет создавать диагностические служебные программы, предоставляющие дополнительные сведения. Разработчики могут быстрее понять, какие изменения необходимы. Можно также использовать CallerArgumentExpressionAttribute для определения того, какое выражение использовалось в качестве получателя для методов расширения. Следующий метод выбирает последовательность через регулярные интервалы. Если в последовательности слишком мало элементов для этой частоты, возникает ошибка:

```
public static IEnumerable<T> Sample<T>(this IEnumerable<T> sequence, int frequency,
    [CallerArgumentExpression("sequence")] string? message = null)
{
    if (sequence.Count() < frequency)
        throw new ArgumentException($"Expression doesn't have enough elements: {message}",
nameof(sequence));
    int i = 0;
    foreach (T item in sequence)
    {
        if (i++ % frequency == 0)
            yield return item;
    }
}
```

Этот метод можно вызвать следующим образом:

```
sample = Enumerable.Range(0, 10).Sample(100);
```

В предыдущем примере выдается исключение ArgumentException со следующим сообщением:

```
Expression doesn't have enough elements: Enumerable.Range(0, 10) (Parameter 'sequence')
```

См. также

- Именованные и необязательные аргументы
- System.Reflection
- Attribute
- Атрибуты

Атрибуты для статического анализа состояния со значением NULL

18.12.2021 • 14 minutes to read

В контексте, допускающем значения NULL, компилятор выполняет статический анализ кода для определения *состояния со значением NULL* всех переменных ссылочного типа.

- *not-null*: статический анализ определяет, что переменной присвоено значение, отличное от NULL.
- *maybe-null*: статический анализ не может определить, что переменной присвоено значение, отличное от NULL.

Эти состояния позволяют компилятору предоставлять предупреждения, если вы можете разыменовать указатель на NULL, вызывая исключение System.NullReferenceException. Эти атрибуты предоставляют компилятору семантическую информацию о *состоянии со значением NULL* аргументов, возвращаемых значений и элементах объекта на основе состояния аргументов и возвращаемых значений. Компилятор предоставляет более точные предупреждения, если к API правильно добавлены эти семантические сведения.

В этой статье дается краткое описание каждого из атрибутов ссылочного типа, допускающего значение NULL, и способа их применения.

Начнем с примера. Представьте себе, что библиотека содержит указанный ниже API для получения строки ресурса. Этот метод изначально был написан до C# 8.0 и заметок о допустимости значения NULL:

```
bool TryGetMessage(string key, out string message)
{
    if (_messageMap.ContainsKey(key))
        message = _messageMap[key];
    else
        message = null;
    return message != null;
}
```

В предыдущем примере применяется знакомый шаблон `Try*` в .NET. Для этого API существует два ссылочных параметра: `key` и `message`. У этого API есть следующие правила, относящиеся к *состоянию со значением NULL* этих параметров:

- Вызывающие объекты не должны передавать `null` в качестве аргумента для `key`.
- Вызывающие объекты могут передавать переменную, значение которой равно `null`, в качестве аргумента для `message`.
- Если метод `TryGetMessage` возвращает `true`, значение `message` не равно NULL. Если возвращаемое значение равно `false,` , значение `message` равно NULL.

Правило для `key` может быть выражено более лаконично в C# 8.0: `key` должен быть ссылочным типом, не допускающим значения NULL. Параметр `message` является более сложным. Он допускает переменную, которая использует `null` в качестве аргумента, но гарантирует, что в случае успеха аргумент `out` не будет иметь значение `null`. В таких случаях требуется более широкое описание ожиданий. Атрибут `NotNullWhen`, описанный ниже, описывает *состояние со значением NULL* для аргумента, используемого для параметра `message`.

NOTE

При добавлении этих атрибутов компилятор получает дополнительные сведения о правилах для API. Если вызывающий код компилируется в контексте, допускающем значение NULL, и вызывающие объекты нарушают эти правила, компилятор выведет соответствующие предупреждения. Эти атрибуты не используются для выполнения дополнительных проверок в вашей реализации.

ATTRIBUTE	КАТЕГОРИЯ	ЗНАЧЕНИЕ
AllowNull	Precondition	Параметр, поле или свойство, не допускающие значения NULL, могут иметь значение NULL.
DisallowNull	Precondition	Параметр, поле или свойство, допускающие значения NULL, не должны иметь значение NULL.
MaybeNull	Постусловие	Параметр, поле, свойство или возвращаемое значение, не допускающие значения NULL, могут иметь значение NULL.
NotNull	Постусловие	Параметр, поле, свойство или возвращаемое значение, допускающие значения NULL, не будут иметь значение NULL.
MaybeNullWhen	Условное постусловие	Аргумент, не допускающий значение NULL, может иметь значение NULL, если метод возвращает указанное значение `bool`.
NotNullWhen	Условное постусловие	Аргумент, допускающий значение NULL, не будет иметь значение NULL, если метод возвращает указанное значение `bool`.
NotNullIfNotNull	Условное постусловие	Возвращаемое значение, свойство или аргумент не равны NULL, если аргумент для указанного параметра не равен NULL.
MemberNotNull	Вспомогательные методы для методов и свойств	Если метод возвращает значение, включенный в список элемент не будет иметь значение NULL.
MemberNotNullWhen.	Вспомогательные методы для методов и свойств	Если метод возвращает указанное значение `bool`, включенный в список элемент не будет иметь значение NULL.
DoesNotReturn	Недостижимый код	Метод или свойство никогда не возвращает значение. Другими словами, он всегда создает исключение.

ATTRIBUTE	КАТЕГОРИЯ	ЗНАЧЕНИЕ
DoesNotReturnIf	Недостижимый код	Этот метод или свойство никогда не возвращает значение, если связанный параметр `bool` имеет указанное значение.

Приведенные выше сведения лишь кратко описывают каждый атрибут. В следующих разделах более подробно описывается поведение и значение этих атрибутов.

Предусловия: `AllowNull` и `DisallowNull`

Рассмотрим свойство для чтения и записи, которое никогда не возвращает `null`, так как имеет разумное значение по умолчанию. Вызывающие объекты передают `null` методу доступа set при задании ему этого значения по умолчанию. Например, рассмотрим систему обмена сообщениями, запрашивающую имя экрана в комнате чата. Если имя не указано, система создает случайное значение.

```
public string ScreenName
{
    get => _screenName;
    set => _screenName = value ?? GenerateRandomScreenName();
}
private string _screenName;
```

При компиляции предыдущего кода в контексте, игнорирующем допустимость значения NULL, не возникает никаких проблем. После включения ссылочных типов, допускающих значения NULL, свойство `ScreenName` преобразуется в ссылку, не допускающую значение NULL. Это верно для метода доступа `get`: он никогда не возвращает `null`. Вызывающим объектам не нужно проверять возвращаемое свойство на наличие значения `null`. Но теперь при задании свойству значения `null` создается предупреждение. Чтобы включить поддержку этого типа кода, добавьте атрибут System.Diagnostics.CodeAnalysis.AllowNullAttribute к свойству, как показано в следующем коде.

```
[AllowNull]
public string ScreenName
{
    get => _screenName;
    set => _screenName = value ?? GenerateRandomScreenName();
}
private string _screenName = GenerateRandomScreenName();
```

Чтобы использовать этот и другие атрибуты, описанные в этой статье, может потребоваться добавить директиву `using` для пространства имен System.Diagnostics.CodeAnalysis. Атрибут применяется к свойству,а не методу доступа `set`. Атрибут `AllowNull` позволяет определить *предусловия* и применяется только к аргументам. Метод доступа `get` имеет возвращаемое значение, но у него нет параметров. Таким образом, атрибут `AllowNull` применяется только к методу доступа `set`.

В предыдущем примере показано, что следует искать при добавлении атрибута `AllowNull` к аргументу.

1. Общий контракт для этой переменной заключается в том, что она не должна принимать значение `null`, поэтому нужен ссылочный тип, не допускающий значение NULL.
2. Существуют ситуации, когда вызывающий объект передается в `null` в качестве аргумента, хотя это не самый распространенный способ использования.

Чаще всего этот атрибут будет необходим для свойств или аргументов `in`, `out` и `ref`. Атрибут `AllowNull` лучше всего применять в ситуации, когда переменная обычно имеет значение, отличное от

NULL, но необходимо разрешить `null` в качестве предусловия.

Сравните этот вариант со сценариями использования `DisallowNull`: этот атрибут используется для указания того, что аргумент ссылочного типа, допускающего значение NULL, не должен иметь значение `null`. Рассмотрим свойство, где `null` является значением по умолчанию, но клиенты могут задать для него только значение, отличное от NULL. Рассмотрим следующий код.

```
public string ReviewComment
{
    get => _comment;
    set => _comment = value ?? throw new ArgumentNullException(nameof(value), "Cannot set to null");
}
string _comment;
```

Предыдущий код лучше всего демонстрирует ваше намерение выразить то, что атрибут `ReviewComment` может принимать значение `null`, но ему невозможно задать значение `null`. В коде, допускающем значения NULL, эту концепцию можно представить вызывающим объектам более четко с помощью System.Diagnostics.CodeAnalysis.DisallowNullAttribute.

```
[DisallowNull]
public string? ReviewComment
{
    get => _comment;
    set => _comment = value ?? throw new ArgumentNullException(nameof(value), "Cannot set to null");
}
string? _comment;
```

В контексте, допускающем значения NULL, метод доступа `ReviewComment` `get` может возвращать значение по умолчанию, равное `null`. Компилятор предупреждает, что перед доступом необходимо проверить это значение. Более того, он предупреждает вызывающие объекты о том, что это может быть `null`, им не следует явно задавать его равным `null`. Атрибут `DisallowNull` также задает *предварительное условие*. Он не влияет на метод доступа `get`. Атрибут `DisallowNull` используется при отслеживании следующих моментов.

1. Переменная может принимать значение `null` в основных сценариях, часто при первом создании экземпляра.
2. Переменной не следует явно задавать значение `null`.

Эти ситуации распространены в коде, который изначально *игнорировал допустимость значений NULL*. Может быть так, что свойства объекта задаются в двух отдельных операциях инициализации. Может быть так, что некоторые свойства задаются только после завершения какой-либо асинхронной операции.

Атрибуты `AllowNull` и `DisallowNull` позволяют указать, что предусловия для переменных могут не соответствовать заметкам, допускающим значение NULL, для этих переменных. Они предоставляют более подробные сведения о характеристиках API. Эта информация помогает вызывающим объектам правильно использовать API. Помните, что для указания предусловий используются следующие атрибуты.

- AllowNull. Аргумент, не допускающий значение NULL, может принимать значение NULL.
- DisallowNull. Аргумент, допускающий значение NULL, никогда не должен принимать значение NULL.

Постусловия: `MaybeNull` и `NotNull`

Предположим, у вас есть метод со следующей сигнатурой.

```
public Customer FindCustomer(string lastName, string firstName)
```

Вероятно, вы написали такой метод для возвращения значения `null` в случае, если искомое имя не найдено. `null` четко указывает, что запись не найдена. В этом примере вы, вероятно, измените тип возвращаемого значения с `Customer` на `Customer?`. Объявление возвращаемого значения как ссылочного типа, допускающего значение NULL, четко указывает намерение этого API:

```
public Customer? FindCustomer(string lastName, string firstName)
```

По причинам, описанным в разделе Допустимость значений NULL для универсальных методов, этот метод может не создавать статический анализ, соответствующий API. У вас может быть универсальный метод, который действует по аналогичному шаблону.

```
public T Find<T>(IEnumerable<T> sequence, Func<T, bool> predicate)
```

Если искомый элемент не найден, метод возвращает `null`. Можно уточнить, что метод возвращает значение `null`, если элемент не найден, добавив заметку `MaybeNull` к возвращаемому значению метода:

```
[return: MaybeNull]
public T Find<T>(IEnumerable<T> sequence, Func<T, bool> predicate)
```

Предыдущий код информирует вызывающие объекты о том, что возвращаемое значение на самом деле *может* быть равно NULL. Он также информирует компилятор о том, что метод может возвращать выражение `null`, даже если тип не допускает значения NULL. При наличии универсального метода, возвращающего экземпляр своего параметра типа, `T`, с помощью атрибута `NotNull` можно указать, что он никогда не возвращает `null`.

Можно также указать, что возвращаемое значение или аргумент не равны NULL, даже если используется ссылочный тип, допускающий значение NULL. Следующий метод является вспомогательным и выдает исключение, если первым аргументом является `null`.

```
public static void ThrowWhenNull(object value, string valueExpression = "")
{
    if (value is null) throw new ArgumentNullException(nameof(value), valueExpression);
}
```

Эту подпрограмму можно вызвать следующим образом.

```
public static void LogMessage(string? message)
{
    ThrowWhenNull(message, $"{nameof(message)} must not be null");

    Console.WriteLine(message.Length);
}
```

После включения ссылочных типов, допускающих значения NULL, необходимо убедиться, что предыдущий код компилируется без вывода предупреждений. Когда метод возвращает значение, параметр `value` гарантированно не будет равен NULL. Однако можно вызвать `ThrowWhenNull` с пустой ссылкой. Можно сделать `value` ссылочным типом, допускающим значение NULL, и добавить постусловие `NotNull` в объявление параметра.

```
public static void ThrowWhenNull([NotNull] object? value, string valueExpression = "")
{
    _ = value ?? throw new ArgumentNullException(nameof(value), valueExpression);
    // other logic elided
```

В предыдущем коде показано четкое выражение имеющегося контракта. Вызывающие объекты могут передавать переменную со значением `null`, но аргумент гарантированно не будет равен NULL, если метод возвращает значение без выдачи исключений.

Для указания безусловных постусловий используются следующие атрибуты.

- MaybeNull. Возвращаемое значение, не допускающее значение NULL, может быть равно NULL.
- NotNull. Возвращаемое значение, допускающее значение NULL, никогда не будет равно NULL.

Условные постусловия: `NotNullWhen`, `MaybeNullWhen`, и `NotNullIfNotNull`

Вам, скорее всего, известен метод String.IsNullOrEmpty(String) `string`. Этот метод возвращает `true`, если аргумент имеет значение NULL или является пустой строкой. Вот форма проверки значений NULL. Если метод возвращает `false`, вызывающим объектам не требуется выполнять проверку значений NULL. Чтобы сделать такой метод методом, допускающим значения NULL, необходимо задать для аргумента ссылочный тип, допускающий значение NULL, и добавить атрибут `NotNullWhen`.

```
bool IsNullOrEmpty([NotNullWhen(false)] string? value)
```

В этом случае компилятор знает, что любой код, в котором возвращаемое значение равно `false`, не должен проходить проверку значений NULL. Для статического анализа добавление атрибута означает, что `IsNullOrEmpty` выполняет необходимую проверку значений NULL: когда он возвращает `false`, аргумент не равен `null`.

```
string? userInput = GetUserInput();
if (!string.IsNullOrEmpty(userInput))
{
    int messageLength = userInput.Length; // no null check needed.
}
// null check needed on userInput here.
```

Метод String.IsNullOrEmpty(String) будет помечен для .NET Core 3.0, как показано выше. В вашей базе кода могут быть аналогичные методы, которые проверяют состояние объектов на значения NULL. Компилятор не распознает пользовательские методы проверки значений NULL, и вам потребуется добавить заметки самостоятельно. При добавлении атрибута статический анализ компилятора знает, когда тестируемая переменная прошла проверку на наличие значения NULL.

Другим применением этих атрибутов является шаблон `Try*`. Постусловия для аргументов `ref` и `out` сообщаются через возвращаемое значение. Рассмотрим этот метод, показанный ранее (в контексте, не допускающем значение NULL):

```
bool TryGetMessage(string key, out string message)
{
    if (_messageMap.ContainsKey(key))
        message = _messageMap[key];
    else
        message = null;
    return message != null;
}
```

Предыдущий метод соответствует стандартному подходу .NET: возвращаемое значение указывает, было ли `message` задано найденное значение или задано значение по умолчанию, если сообщение не было найдено. Если метод возвращает `true`, значение `message` не равно NULL. В противном случае метод задает `message` значение NULL.

В контексте, допускающем значение NULL, этот принцип можно передать с помощью атрибута `NotNullWhen`. При обновлении сигнатуры для аннотации параметров типов, допускающих значение NULL, задайте `message` значение `string?` и добавьте атрибут:

```
bool TryGetMessage(string key, [NotNullWhen(true)] out string? message)
{
    if (_messageMap.ContainsKey(key))
        message = _messageMap[key];
    else
        message = null;
    return message is not null;
}
```

В предыдущем примере, когда `TryGetMessage` возвращает `true`, известно, что значение `message` не будет равно NULL. Точно так же можно добавить заметки к аналогичным методам в базе кода: аргументы могут иметь значение `null` и известно, что они не будут равны NULL при возвращении методом значения `true`.

Кроме того, может потребоваться еще один, последний атрибут. Иногда состояние NULL возвращаемого значения зависит от состояния NULL одного или нескольких аргументов. Эти методы возвращают значение, отличающееся от NULL, если определенные аргументы не равны `null`. Чтобы правильно добавить заметки к этим методам, используйте атрибут `NotNullIfNotNull`. Рассмотрим следующий метод.

```
string GetTopLevelDomainFromFullUrl(string url)
```

Если аргумент `url` не равен NULL, выходные данные не равны `null`. После включения ссылок, допускающих значение NULL, необходимо добавить дополнительные заметки, если API может принимать аргумент NULL. Можно добавить заметку к типу возвращаемого значения, как показано в следующем коде:

```
string? GetTopLevelDomainFromFullUrl(string? url)
```

Это работает, но вызывающим объектам придется часто проводить дополнительные проверки `null`. Контракт заключается в том, что возвращаемое значение будет равно `null` только тогда, когда аргумент `url` имеет значение `null`. Чтобы выразить этот контракт, можно добавить заметку к этому методу, как показано в следующем коде.

```
[return: NotNullIfNotNull("url")]
string? GetTopLevelDomainFromFullUrl(string? url)
```

Возвращаемое значение и аргумент снабжены заметкой `?`, указывающей, что каждый из них может быть равен `null`. Атрибут дополнительно уточняет, что возвращаемое значение не будет равно NULL, если аргумент `url` не принимает значение `null`.

Для указания условных постусловий используются следующие атрибуты.

- MaybeNullWhen. Аргумент, не допускающий значение NULL, может иметь значение NULL, если метод возвращает указанное значение `bool`.
- NotNullWhen. Аргумент, допускающий значение NULL, не будет иметь значение NULL, если метод возвращает указанное значение `bool`.
- NotNullIfNotNull. Возвращаемое значение не равно NULL, если аргумент для указанного параметра не равен NULL.

Вспомогательные методы: `MemberNotNull` и `MemberNotNullWhen`

Эти атрибуты определяют ваше намерение при рефакторинге общего кода из конструкторов в вспомогательные методы. Компилятор C# анализирует конструкторы и инициализаторы полей, чтобы убедиться, что все ссылочные поля, не допускающие значения NULL, были инициализированы перед возвратом каждого конструктора. При этом компилятор C# не следит за назначениями полей с помощью всех вспомогательных методов. Компилятор выдает предупреждение `CS8618`, когда поля инициализируются не непосредственно в конструкторе, а в вспомогательном методе. Вам нужно добавить MemberNotNullAttribute в объявление метода и указать поля, которые инициализируются значением, отличным от NULL, в методе. Рассмотрим следующий пример.

```
public class Container
{
    private string _uniqueIdentifier; // must be initialized.
    private string? _optionalMessage;

    public Container()
    {
        Helper();
    }

    public Container(string message)
    {
        Helper();
        _optionalMessage = message;
    }

    [MemberNotNull(nameof(_uniqueIdentifier))]
    private void Helper()
    {
        _uniqueIdentifier = DateTime.Now.Ticks.ToString();
    }
}
```

В конструкторе `MemberNotNull` атрибута можно указать несколько имен полей в качестве аргументов.

MemberNotNullWhenAttribute имеет аргумент `bool`. `MemberNotNullWhen` используется в ситуациях, когда вспомогательный метод возвращает тип `bool`, указывающий, инициализированы ли поля вспомогательного метода.

Останавливает анализ типов, допускающих значение NULL, при выдаче исключения вызванным методом

Некоторые методы, как правило, это вспомогательные методы исключений или другие служебные

методы, всегда завершают работу, вызывая исключение. Или вспомогательный метод может вызвать исключение на основе значения логического аргумента.

В первом случае можно добавить атрибут DoesNotReturnAttribute в объявление метода. При анализе *состояния NULL* компилятора не проверяется код в методе, который следует за вызовом метода, для которого добавлена заметка `DoesNotReturn`. Рассмотрим этот метод:

```
[DoesNotReturn]
private void FailFast()
{
    throw new InvalidOperationException();
}

public void SetState(object containedField)
{
    if (containedField is null)
    {
        FailFast();
    }

    // containedField can't be null:
    _field = containedField;
}
```

Компилятор не выдает никаких предупреждений после вызова `FailFast`.

Во втором случае добавьте атрибут System.Diagnostics.CodeAnalysis.DoesNotReturnIfAttribute к логическому параметру метода. Предыдущий пример можно изменить следующим образом.

```
private void FailFastIf([DoesNotReturnIf(true)] bool isNull)
{
    if (isNull)
    {
        throw new InvalidOperationException();
    }
}

public void SetFieldState(object? containedField)
{
    FailFastIf(containedField == null);
    // No warning: containedField can't be null here:
    _field = containedField;
}
```

Если значение аргумента совпадает со значением конструктора `DoesNotReturnIf`, компилятор не выполняет анализ *состояния NULL* после этого метода.

Сводка

> **IMPORTANT**
>
> В официальной документации отслеживается последняя версия C#. Сейчас мы пишем документацию для C# 9.0. В зависимости от используемой версии C# различные функции могут быть недоступны. Версия C# *по умолчанию* для проекта основана на целевой платформе. Дополнительные сведения см. в статье Управление версиями языка C# по умолчанию.

Добавление ссылочных типов, допускающих значения NULL, предоставляет исходный словарь для описания ожиданий API для переменных, которые могут иметь значение `null`. Атрибуты позволяют

более подробно описывать состояние NULL для переменных в качестве предусловий и постусловий. Эти атрибуты более четко описывают ожидания и обеспечивают более эффективную работу специалистов, использующих ваши API.

При обновлении библиотек для контекста, допускающего значение NULL, добавьте эти атрибуты, чтобы пользователи могли правильно использовать ваши API. С помощью этих атрибутов вы можете полностью описывать состояние NULL аргументов и возвращаемых значений.

- AllowNull. Поле, параметр или свойство, не допускающие значения NULL, могут иметь значение NULL.
- DisallowNull. Поле, параметр или свойство, допускающие значения NULL, не должны иметь значение NULL.
- MaybeNull. Поле, параметр, свойство или возвращаемое значение, не допускающие значения NULL, могут иметь значение NULL.
- NotNull. Поле, параметр, свойство или возвращаемое значение, допускающие значения NULL, не будут иметь значение NULL.
- MaybeNullWhen. Аргумент, не допускающий значение NULL, может иметь значение NULL, если метод возвращает указанное значение `bool`.
- NotNullWhen. Аргумент, допускающий значение NULL, не будет иметь значение NULL, если метод возвращает указанное значение `bool`.
- NotNullIfNotNull. Параметр, свойство или возвращаемое значение не равно NULL, если аргумент для указанного параметра не равен NULL.
- DoesNotReturn. Метод или свойство никогда не возвращает значение. Другими словами, он всегда создает исключение.
- DoesNotReturnIf. Этот метод или свойство никогда не возвращает значение, если связанный параметр `bool` имеет указанное значение.

Зарезервированные атрибуты: разное

18.12.2021 • 8 minutes to read

Эти атрибуты можно применять к элементам в коде. Они добавляют к ним семантическое значение. Компилятор использует эти семантические значения для изменения выходных данных и сообщения о возможных ошибках разработчиков, использующих код.

Атрибут `Conditional`

Атрибут `Conditional` определяет зависимость выполнения метода от идентификатора предварительной обработки. Атрибут `Conditional` является псевдонимом для ConditionalAttribute и может применяться к методу или классу атрибута.

В следующем примере атрибут `Conditional` применяется к методу для включения и отключения отображения диагностической информации для конкретной программы.

```
#define TRACE_ON
using System;
using System.Diagnostics;

namespace AttributeExamples
{
    public class Trace
    {
        [Conditional("TRACE_ON")]
        public static void Msg(string msg)
        {
            Console.WriteLine(msg);
        }
    }

    public class TraceExample
    {
        public static void Main()
        {
            Trace.Msg("Now in Main...");
            Console.WriteLine("Done.");
        }
    }
}
```

Если идентификатор `TRACE_ON` не определен, выходные данные трассировки не отображаются. Поэкспериментируйте в интерактивном окне.

Атрибут `Conditional` часто используется с идентификатором `DEBUG` для включения функций трассировки и ведения журнала для отладочных сборок (но не сборок выпуска), как показано в следующем примере.

```
[Conditional("DEBUG")]
static void DebugMethod()
{
}
```

Когда вызывается метод, помеченный как условный, наличие или отсутствие заданного символа

предварительной обработки определяет, включает ли компилятор вызов в метод или опускает его. Если этот символ определен, вызов включается; в противном случае вызов пропускается. Условный метод должен быть методом в объявлении класса или структуры и должен иметь тип возвращаемого значения `void`. Использование атрибута `Conditional` — это более понятная, элегантная и менее подверженная ошибкам альтернатива вложению методов в блоки `#if…#endif`.

Если метод содержит несколько атрибутов `Conditional`, компилятор включает вызовы в метод, если определен один или несколько условных символов (символы логически связаны с помощью оператора ИЛИ). В следующем примере наличие `A` или `B` приведет к вызову метода.

```
[Conditional("A"), Conditional("B")]
static void DoIfAorB()
{
    // ...
}
```

Использование `Conditional` с классами атрибутов

Атрибут `Conditional` также может применяться к определению класса атрибута. В следующем примере настраиваемый атрибут `Documentation` добавит сведения в метаданные, только если задано значение `DEBUG`.

```
[Conditional("DEBUG")]
public class DocumentationAttribute : System.Attribute
{
    string text;

    public DocumentationAttribute(string text)
    {
        this.text = text;
    }
}

class SampleClass
{
    // This attribute will only be included if DEBUG is defined.
    [Documentation("This method displays an integer.")]
    static void DoWork(int i)
    {
        System.Console.WriteLine(i.ToString());
    }
}
```

Атрибут `Obsolete`

Атрибут `Obsolete` помечает элемент кода как больше не рекомендуемый для использования. Использование сущности, помеченной как устаревшая, приводит к возникновению предупреждения или ошибки. Атрибут `Obsolete` является атрибутом однократного использования и может применяться к любой сущности, допускающей использование атрибутов. `Obsolete` является псевдонимом для ObsoleteAttribute.

В следующем примере атрибут `Obsolete` применяется к классу `A` и к методу `B.OldMethod`. Поскольку для второго аргумента конструктора атрибутов, примененного к `B.OldMethod`, задано значение `true`, этот метод будет вызывать ошибку компилятора, тогда как при использовании класса `A` будет просто создаваться предупреждение. При этом вызов `B.NewMethod` не создает предупреждений или ошибок. Например, при использовании с предыдущими определениями следующий код создает два предупреждения и одну ошибку:

```
using System;

namespace AttributeExamples
{
    [Obsolete("use class B")]
    public class A
    {
        public void Method() { }
    }

    public class B
    {
        [Obsolete("use NewMethod", true)]
        public void OldMethod() { }

        public void NewMethod() { }
    }

    public static class ObsoleteProgram
    {
        public static void Main()
        {
            // Generates 2 warnings:
            A a = new A();

            // Generate no errors or warnings:
            B b = new B();
            b.NewMethod();

            // Generates an error, compilation fails.
            // b.OldMethod();
        }
    }
}
```

Строка, предоставленная в качестве первого аргумента конструктору атрибута, будет отображаться как часть предупреждения или ошибки. Создается два предупреждения для класса `A`: одно для объявления ссылки на класс, а второе — для конструктора класса. Атрибут `Obsolete` может использоваться без аргументов, однако рекомендуется включать пояснение о том, что следует использовать вместо него.

В C# 10 можно использовать интерполяцию константной строки и оператор `nameof`, чтобы гарантировать совпадение имен:

```
public class B
{
    [Obsolete($"use {nameof(NewMethod)} instead", true)]
    public void OldMethod() { }

    public void NewMethod() { }
}
```

Атрибут `AttributeUsage`

Атрибут `AttributeUsage` определяет, как можно использовать пользовательский класс атрибутов. AttributeUsageAttribute — это атрибут, примененный к определениям настраиваемого атрибута. С помощью атрибута `AttributeUsage` можно контролировать:

- Какой атрибут элементов программы можно применить. Если вы не ограничите использование, атрибут можно применить к любому из следующих элементов программы:
  - Сборка

    - Модуль
    - Поле
    - Событие
    - Метод
    - Параметр
    - Свойство
    - Возвращает
    - Тип
- Можно ли применить атрибут к одному элементу программы несколько раз.
- Могут ли атрибуты наследоваться производными классами.

При явном применении параметры по умолчанию выглядят следующим образом:

```
[AttributeUsage(AttributeTargets.All,
                  AllowMultiple = false,
                  Inherited = true)]
class NewAttribute : Attribute { }
```

В этом примере класс `NewAttribute` можно применить к любому поддерживаемому элементу программы. Но он применяется к каждому объекту только один раз. Атрибут наследуется производными классами при применении к базовому классу.

Аргументы AllowMultiple и Inherited являются необязательными, так что следующий код имеет тот же результат:

```
[AttributeUsage(AttributeTargets.All)]
class NewAttribute : Attribute { }
```

Первый аргумент AttributeUsageAttribute должен состоять из одного или нескольких элементов перечисления AttributeTargets. Несколько типов целевого объекта можно связать с помощью оператора OR, как показано в следующем примере:

```
[AttributeUsage(AttributeTargets.Property | AttributeTargets.Field)]
class NewPropertyOrFieldAttribute : Attribute { }
```

Начиная с C# 7.3 атрибуты могут применяться либо к свойству, либо к резервному полю для автоматически реализуемого свойства. Атрибут применяется к свойству, если не указан описатель `field` для атрибута. Оба случая показаны в следующем примере:

```
class MyClass
{
    // Attribute attached to property:
    [NewPropertyOrField]
    public string Name { get; set; } = string.Empty;

    // Attribute attached to backing field:
    [field: NewPropertyOrField]
    public string Description { get; set; } = string.Empty;
}
```

Если аргументу AllowMultiple присвоено значение `true`, то результирующий атрибут можно применить несколько раз к одной сущности, как показано в следующем примере:

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
class MultiUse : Attribute { }

[MultiUse]
[MultiUse]
class Class1 { }

[MultiUse, MultiUse]
class Class2 { }
```

В этом случае `MultiUseAttribute` можно применять несколько раз, так как `AllowMultiple` имеет значение `true`. Для применения нескольких атрибутов допускаются оба показанных формата.

Если Inherited — `false`, атрибут не наследуется классами, производными от класса атрибутов. Пример:

```
[AttributeUsage(AttributeTargets.Class, Inherited = false)]
class NonInheritedAttribute : Attribute { }

[NonInherited]
class BClass { }

class DClass : BClass { }
```

В этом случае `NonInheritedAttribute` не применяется к `DClass` путем наследования.

Эти ключевые слова также позволяют указать, где нужно применить атрибут. Например, вы можете использовать описатель `field:`, чтобы добавить атрибут в резервное поле автоматически реализуемого свойства. Вы также можете использовать описатели `field:`, `property:` или `param:`, чтобы применить атрибут к любому из элементов, созданных из позиционной записи. Пример см. в разделе Позиционный синтаксис для определения свойств.

Атрибут `AsyncMethodBuilder`

Начиная с C# 7, атрибут System.Runtime.CompilerServices.AsyncMethodBuilderAttribute добавляется к типу, который может быть асинхронным типом возвращаемого значения. Атрибут задает тип, который создает реализацию асинхронного метода, когда указанный тип возвращается из асинхронного метода. Атрибут `AsyncMethodBuilder` можно применить к типу, который:

- Имеет доступный метод `GetAwaiter`.
- Объект, возвращаемый методом `GetAwaiter`, реализует интерфейс System.Runtime.CompilerServices.ICriticalNotifyCompletion.

Конструктор для атрибута `AsyncMethodBuilder` указывает тип связанного построителя. В построителе должны быть реализованы следующие доступные члены:

- статический метод `Create()`, возвращающий тип построителя;
- доступное для чтения свойство `Task`, возвращающее асинхронный тип возвращаемого значения;
- метод `void SetException(Exception)`, который задает исключение при сбое задачи;
- метод `void SetResult()` или `void SetResult(T result)`, который помечает задачу как завершенную и при необходимости задает результат задачи;
- метод `Start` со следующей сигнатурой API:

```
void Start<TStateMachine>(ref TStateMachine stateMachine)
          where TStateMachine : System.Runtime.CompilerServices.IAsyncStateMachine
```

- метод `AwaitOnCompleted` со следующей сигнатурой:

```
public void AwaitOnCompleted<TAwaiter, TStateMachine>(ref TAwaiter awaiter, ref TStateMachine
stateMachine)
    where TAwaiter : System.Runtime.CompilerServices.INotifyCompletion
    where TStateMachine : System.Runtime.CompilerServices.IAsyncStateMachine
```

- метод `AwaitUnsafeOnCompleted` со следующей сигнатурой:

```
      public void AwaitUnsafeOnCompleted<TAwaiter, TStateMachine>(ref TAwaiter awaiter, ref
TStateMachine stateMachine)
          where TAwaiter : System.Runtime.CompilerServices.ICriticalNotifyCompletion
          where TStateMachine : System.Runtime.CompilerServices.IAsyncStateMachine
```

Дополнительные сведения о построителях асинхронных методов см. в статьях, посвященных следующим сборщикам в .NET:

- System.Runtime.CompilerServices.AsyncTaskMethodBuilder
- System.Runtime.CompilerServices.AsyncTaskMethodBuilder<TResult>
- System.Runtime.CompilerServices.AsyncValueTaskMethodBuilder
- System.Runtime.CompilerServices.AsyncValueTaskMethodBuilder<TResult>

В C# 10 и более поздних версиях атрибут `AsyncMethodBuilder` может применяться к асинхронному методу для переопределения построителя для этого типа.

Атрибуты `InterpolatedStringHandler` и `InterpolatedStringHandlerArguments`

Начиная с версии C# 10 эти атрибуты используются для уточнения того, что тип указывает на *обработчика интерполированных строк*. Библиотека .NET 6 уже включает в себя System.Runtime.CompilerServices.DefaultInterpolatedStringHandler для сценариев, в которых интерполированная строка используется в качестве аргумента для параметра `string`. У вас могут быть другие экземпляры, в которых требуется управлять способом обработки интерполированных строк. Вам потребуется применить System.Runtime.CompilerServices.InterpolatedStringHandlerAttribute к типу, реализующему обработчик, а также применить System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute для параметров конструктора этого типа.

Дополнительные сведения о создании обработчика интерполированных строк см. в спецификации функций C# 10 для улучшения интерполяции строк.

Атрибут `ModuleInitializer`

Начиная с C# 9, атрибут `ModuleInitializer` помечает метод, который среда выполнения вызывает при загрузке сборки. `ModuleInitializer` является псевдонимом для ModuleInitializerAttribute.

Атрибут `ModuleInitializer` может применяться только к методу, который:

- является статическим;
- не имеет параметров;

- Возвращает `void`.
- доступен из содержащего модуля, а именно `internal` или `public`;
- не является универсальным методом;
- не содержится в универсальном классе;
- не является локальной функцией.

Атрибут `ModuleInitializer` можно применить к нескольким методам. В этом случае порядок, в котором среда выполнения вызывает их, является детерминированным, но не задан.

В следующем примере показано использование нескольких методов инициализатора модуля. Методы `Init1` и `Init2` выполняются до `Main`, и каждый добавляет строку в свойство `Text`. Поэтому при выполнении `Main` свойство `Text` уже имеет строки из обоих методов инициализатора.

```
using System;

internal class ModuleInitializerExampleMain
{
    public static void Main()
    {
        Console.WriteLine(ModuleInitializerExampleModule.Text);
        //output: Hello from Init1! Hello from Init2!
    }
}
```

```
using System.Runtime.CompilerServices;

internal class ModuleInitializerExampleModule
{
    public static string? Text { get; set; }

    [ModuleInitializer]
    public static void Init1()
    {
        Text += "Hello from Init1! ";
    }

    [ModuleInitializer]
    public static void Init2()
    {
        Text += "Hello from Init2! ";
    }
}
```

Иногда генераторам исходного кода требуется создать код инициализации. Инициализаторы модулей предоставляют стандартное расположение для этого кода.

Атрибут `SkipLocalsInit`

Начиная с C# 9, атрибут `SkipLocalsInit` не позволяет компилятору устанавливать флаг `.locals init` при выводе в метаданные. Атрибут `SkipLocalsInit` является одноразовым и может применяться к методу, свойству, классу, структуре, интерфейсу или модулю, но не к сборке. `SkipLocalsInit` является псевдонимом для SkipLocalsInitAttribute.

Флаг `.locals init` заставляет среду CLR инициализировать все локальные переменные, объявленные в методе, со значениями по умолчанию. Так как компилятор также гарантирует, что вы никогда не используете переменную, прежде чем назначить ей какое-либо значение, `.locals init` обычно не требуется. Однако дополнительная нулевая инициализация может привести к снижению

производительности в некоторых сценариях, например при использовании stackalloc для выделения массива в стеке. В таких случаях можно добавить атрибут `SkipLocalsInit`. При применении напрямую к методу атрибут воздействует на него и все его вложенные функции, включая лямбда-выражения и локальные функции. При применении к типу или модулю он влияет на все вложенные внутрь методы. Этот атрибут не воздействует на абстрактные методы, но воздействует на код, созданный для реализации.

Для этого атрибута требуется параметр компилятора AllowUnsafeBlocks. Требование указывает на то, что в некоторых случаях код может иметь доступ к неназначенной памяти (например, осуществлять чтение из неинициализированной памяти, выделенной в стеке).

В следующем примере показано воздействие атрибута `SkipLocalsInit` на метод, использующий `stackalloc`. Метод отображает все, что было в памяти на момент выделения массива целых чисел.

```
[SkipLocalsInit]
static void ReadUninitializedMemory()
{
    Span<int> numbers = stackalloc int[120];
    for (int i = 0; i < 120; i++)
    {
        Console.WriteLine(numbers[i]);
    }
}
// output depends on initial contents of memory, for example:
//0
//0
//0
//168
//0
//-1271631451
//32767
//38
//0
//0
//0
//38
// Remaining rows omitted for brevity.
```

Чтобы выполнить этот код самостоятельно, задайте параметр компилятора `AllowUnsafeBlocks` в файле *CSPROJ*.

```
<PropertyGroup>
  ...
  <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
</PropertyGroup>
```

См. также

- Attribute
- System.Reflection
- Атрибуты
- Отражение

Ненадежный код, типы указателей и указатели функций

18.12.2021 • 12 minutes to read

Большая часть создаваемого кода C# — «проверяемо безопасный код». Термин *«проверяемо безопасный код»* означает, что с помощью средств .NET можно проверить, является ли данный код безопасным. Как правило, в безопасном коде не используется прямой доступ к памяти с помощью указателей. В нем также не используется выделение блоков памяти в «сыром» виде. Вместо этого в нем создаются управляемые объекты.

C# поддерживает контекст `unsafe`, в котором можно писать *непроверяемый* код. В контексте `unsafe` в коде можно использовать указатели, выделять и освобождать блоки памяти, а также обращаться к методам с помощью указателей функций. Небезопасный код в C# не обязательно является опасным, просто его безопасность нельзя проверить.

Небезопасный код имеет следующие свойства:

- Методы, типы и блоки кода можно определить как небезопасные.
- В некоторых случаях небезопасный код может увеличить скорость работы приложения, если не проверяются границы массивов.
- Небезопасный код необходим при вызове встроенных стандартных функций, требующих указателей.
- Использование небезопасного кода создает риски для стабильности и безопасности.
- Код, содержащий небезопасные блоки, должен компилироваться с параметром компилятора AllowUnsafeBlocks.

типы указателей

В небезопасном контексте тип может быть не только типом значения или ссылочным типом, но и типом указателя. Объявления типа указателя выполняется одним из следующих способов:

```
type* identifier;
void* identifier; //allowed but not recommended
```

Тип, указанный до `*` в типе указателя, называется **ссылочным типом**. Ссылочным типом может быть только неуправляемый тип.

Типы указателей не наследуются от объекта, а типы указателей не преобразуются в `object`. Кроме того, упаковка-преобразование и распаковка-преобразование не поддерживают указатели. Однако можно выполнять преобразования между различными типами указателей, а также между типами указателей и целочисленными типами.

При объявлении нескольких указателей в одном объявлении знак (`*`) указывается только с базовым типом. Он не используется в качестве префикса для каждого имени указателя. Пример:

```
int* p1, p2, p3;   // Ok
int *p1, *p2, *p3;   // Invalid in C#
```

Указатель не может указывать на ссылку или на структуру, содержащую ссылки, поскольку ссылка на объект может быть подвергнута сбору мусора, даже если на нее указывает указатель. Сборщик мусора

не отслеживает наличие указателей любых типов, указывающих на объекты.

Значением переменной-указателя типа `MyType*` является адрес переменной типа `MyType`. Ниже приведены примеры объявлений типов указателей.

- `int* p` : `p` — указатель на целое число.
- `int** p` : `p` — указатель на указатель на целое число.
- `int*[] p` : `p` — одномерный массив указателей на целые числа.
- `char* p` : `p` — указатель на тип char.
- `void* p` : `p` — указатель на неизвестный тип.

Оператор косвенного обращения указателя `*` можно использовать для доступа к содержимому, на которое указывает переменная-указатель. В качестве примера рассмотрим следующее объявление:

```
int* myVariable;
```

Выражение `*myVariable` обозначает переменную `int`, находящуюся по адресу, содержащемуся в `myVariable`.

Несколько примеров указателей можно найти в статьях, посвященных оператору `fixed`. В следующем примере используются ключевое слово `unsafe` и оператор `fixed`, а также демонстрируется способ инкрементирования внутреннего указателя. Этот код можно вставить в функцию Main консольного приложения для его запуска. Эти примеры должны быть скомпилированы с заданным параметром AllowUnsafeBlocks.

```
// Normal pointer to an object.
int[] a = new int[5] { 10, 20, 30, 40, 50 };
// Must be in unsafe code to use interior pointers.
unsafe
{
    // Must pin object on heap so that it doesn't move while using interior pointers.
    fixed (int* p = &a[0])
    {
        // p is pinned as well as object, so create another pointer to show incrementing it.
        int* p2 = p;
        Console.WriteLine(*p2);
        // Incrementing p2 bumps the pointer by four bytes due to its type ...
        p2 += 1;
        Console.WriteLine(*p2);
        p2 += 1;
        Console.WriteLine(*p2);
        Console.WriteLine("--------");
        Console.WriteLine(*p);
        // Dereferencing p and incrementing changes the value of a[0] ...
        *p += 1;
        Console.WriteLine(*p);
        *p += 1;
        Console.WriteLine(*p);
    }
}

Console.WriteLine("--------");
Console.WriteLine(a[0]);

/*
Output:
10
20
30
--------
10
11
12
--------
12
*/
```

Для указателя типа `void*` использовать оператор косвенного обращения нельзя. Однако можно использовать приведение для преобразования указателя типа void в любой другой тип и наоборот.

Указатель может иметь значение `null`. При применении оператора косвенного обращения к указателю со значением null результат зависит от конкретной реализации.

При передаче указателей между методами может возникнуть неопределенное поведение. Рекомендуется использовать метод, возвращающий указатель в локальную переменную с помощью параметра `in`, `out` или `ref` либо в виде результата функции. Если указатель был задан в фиксированном блоке, переменная, на которую он указывает, больше не может быть фиксированной.

В следующей таблице перечислены операторы, которые можно использовать для указателей в небезопасном контексте.

ОПЕРАТОР	ИСПОЛЬЗОВАТЬ
`*`	Косвенное обращение к указателю.
`->`	Доступ к члену структуры через указатель.

ОПЕРАТОР	ИСПОЛЬЗОВАТЬ
`[]`	Индексирование указателя.
`&`	Получение адреса переменной.
`++` и `--`	Увеличение и уменьшение указателей.
`+` и `-`	Арифметические действия с указателем.
`==`, `!=`, `<`, `>`, `<=` и `>=`	Сравнение указателей.
`stackalloc`	Выделение памяти в стеке.
Инструкция `fixed`	Временная фиксация переменной, чтобы можно было найти ее адрес.

Дополнительные сведения об операторах, связанных с указателем, см. в разделе Операторы, связанные с указателем.

Любой тип указателя можно неявно преобразовать в тип `void*`. Любому типу указателя может быть присвоено значение `null`. Любой тип указателя можно явно преобразовать в любой другой тип указателя с помощью выражения приведения. Можно также преобразовать любой целочисленный тип в тип указателя или преобразовать любой тип указателя в целочисленный тип. Для этих преобразований требуется явным образом использовать приведение.

Например, в следующем примере тип `int*` преобразуется в тип `byte*`. Обратите внимание, что указатель указывает на наименьший адресуемый байт переменной. При последовательном увеличении результата до размера `int` (4 байта) можно отобразить оставшиеся байты переменной.

```
int number = 1024;

unsafe
{
    // Convert to byte:
    byte* p = (byte*)&number;

    System.Console.Write("The 4 bytes of the integer:");

    // Display the 4 bytes of the int variable:
    for (int i = 0 ; i < sizeof(int) ; ++i)
    {
        System.Console.Write(" {0:X2}", *p);
        // Increment the pointer:
        p++;
    }
    System.Console.WriteLine();
    System.Console.WriteLine("The value of the integer: {0}", number);

    /* Output:
        The 4 bytes of the integer: 00 04 00 00
        The value of the integer: 1024
    */
}
```

Буферы фиксированного размера

В языке C# для создания буфера с массивом фиксированного размера в структуре данных можно использовать оператор fixed. Буферы фиксированного размера полезны при написании методов, взаимодействующих с источниками данных, созданными на других языках или платформах. Фиксированный массив может принимать любые атрибуты или модификаторы, допустимые для обычных членов структуры. Единственным ограничением является то, что массив должен иметь тип `bool`, `byte`, `char`, `short`, `int`, `long`, `sbyte`, `ushort`, `uint`, `ulong`, `float` или `double`.

```
private fixed char name[30];
```

В безопасном коде структура C#, содержащая массив, не содержит элементы массива. Вместо этого в ней присутствуют ссылки на элементы. Вы можете внедрить массив фиксированного размера в структуру, если он используется в блоке небезопасного кода.

Размер следующего объекта `struct` не зависит от количества элементов в массиве, поскольку `pathName` представляет собой ссылку:

```
public struct PathArray
{
    public char[] pathName;
    private int reserved;
}
```

`struct` может содержать внедренный массив в небезопасном коде. В приведенном ниже примере массив `fixedBuffer` имеет фиксированный размер. Для установки указателя на первый элемент используется оператор `fixed`. С помощью этого указателя осуществляется доступ к элементам массива. Оператор `fixed` закрепляет поле экземпляра `fixedBuffer` в определенном месте в памяти.

```
internal unsafe struct Buffer
{
    public fixed char fixedBuffer[128];
}

internal unsafe class Example
{
    public Buffer buffer = default;
}

private static void AccessEmbeddedArray()
{
    var example = new Example();

    unsafe
    {
        // Pin the buffer to a fixed location in memory.
        fixed (char* charPtr = example.buffer.fixedBuffer)
        {
            *charPtr = 'A';
        }
        // Access safely through the index:
        char c = example.buffer.fixedBuffer[0];
        Console.WriteLine(c);

        // Modify through the index:
        example.buffer.fixedBuffer[0] = 'B';
        Console.WriteLine(example.buffer.fixedBuffer[0]);
    }
}
```

Размер массива из 128 элементов `char` составляет 256 байт. В буферах типа char фиксированного

размера на один символ всегда приходится по 2 байта независимо от кодировки. Этот размер массива сохраняется даже в том случае, когда буферы char маршалируются в методы API или структуры с `CharSet = CharSet.Auto` или `CharSet = CharSet.Ansi`. Для получения дополнительной информации см. CharSet.

В предыдущем примере демонстрировался доступ к полям `fixed` без закрепления в памяти, доступный в C#, начиная с версии 7.3.

Еще одним распространенным массивом фиксированного размера является массив bool. Элементы в массиве `bool` всегда имеют размер в 1 байт. Массивы `bool` не подходят для создания битовых массивов или буферов.

Буферы фиксированного размера компилируются с помощью класса System.Runtime.CompilerServices.UnsafeValueTypeAttribute, что указывает среде CLR, что тип содержит неуправляемый массив, который может привести к переполнению. Кроме того, в среде CLR для памяти, выделенной с помощью функции stackalloc, автоматически включаются функции обнаружения переполнения буфера. В предыдущем примере показано существование буфера фиксированного размера в `unsafe struct`.

```
internal unsafe struct Buffer
{
    public fixed char fixedBuffer[128];
}
```

Созданный компилятором код C# для `Buffer` помечен с помощью атрибутов, как показано далее.

```
internal struct Buffer
{
    [StructLayout(LayoutKind.Sequential, Size = 256)]
    [CompilerGenerated]
    [UnsafeValueType]
    public struct <fixedBuffer>e__FixedBuffer
    {
        public char FixedElementField;
    }

    [FixedBuffer(typeof(char), 128)]
    public <fixedBuffer>e__FixedBuffer fixedBuffer;
}
```

Буферы фиксированного размера отличаются от обычных массивов указанными ниже особенностями.

- Могут использоваться только в `unsafe` контексте.
- Могут быть только полями экземпляров структур.
- Всегда являются векторами или одномерными массивами.
- Объявление должно включать длину, например `fixed char id[8]`. Вы не можете использовать `fixed char id[]`.

Практическое руководство. Использование указателей для копирования массива байтов

В следующем примере указатели используются для копирования байт из одного массива в другой.

В этом примере применяется ключевое слово unsafe, которое позволяет использовать указатели в методе `Copy`. Оператор fixed используется для объявления указателей исходного и конечного массивов. Оператор `fixed` *закрепляет* расположение исходного и конечного массивов в памяти, чтобы они не

перемещались при сборке мусора. Закрепление этих блоков памяти отменяется, когда завершается обработка блока `fixed`. Поскольку метод `Copy` в этом примере использует ключевое слово `unsafe`, он должен быть скомпилирован с параметром компилятора AllowUnsafeBlocks.

В этом примере доступ к элементам обоих массивов выполняется с помощью индексов, а не второго неуправляемого указателя. Объявление указателей `pSource` и `pTarget` закрепляет массивы. Эта возможность доступна начиная с C# 7.3.

```
static unsafe void Copy(byte[] source, int sourceOffset, byte[] target,
    int targetOffset, int count)
{
    // If either array is not instantiated, you cannot complete the copy.
    if ((source == null) || (target == null))
    {
        throw new System.ArgumentException();
    }

    // If either offset, or the number of bytes to copy, is negative, you
    // cannot complete the copy.
    if ((sourceOffset < 0) || (targetOffset < 0) || (count < 0))
    {
        throw new System.ArgumentException();
    }

    // If the number of bytes from the offset to the end of the array is
    // less than the number of bytes you want to copy, you cannot complete
    // the copy.
    if ((source.Length - sourceOffset < count) ||
        (target.Length - targetOffset < count))
    {
        throw new System.ArgumentException();
    }

    // The following fixed statement pins the location of the source and
    // target objects in memory so that they will not be moved by garbage
    // collection.
    fixed (byte* pSource = source, pTarget = target)
    {
        // Copy the specified number of bytes from source to target.
        for (int i = 0; i < count; i++)
        {
            pTarget[targetOffset + i] = pSource[sourceOffset + i];
        }
    }
}

static void UnsafeCopyArrays()
{
    // Create two arrays of the same length.
    int length = 100;
    byte[] byteArray1 = new byte[length];
    byte[] byteArray2 = new byte[length];

    // Fill byteArray1 with 0 - 99.
    for (int i = 0; i < length; ++i)
    {
        byteArray1[i] = (byte)i;
    }

    // Display the first 10 elements in byteArray1.
    System.Console.WriteLine("The first 10 elements of the original are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray1[i] + " ");
    }
    System.Console.WriteLine("\n");
```

```
    // Copy the contents of byteArray1 to byteArray2.
    Copy(byteArray1, 0, byteArray2, 0, length);

    // Display the first 10 elements in the copy, byteArray2.
    System.Console.WriteLine("The first 10 elements of the copy are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray2[i] + " ");
    }
    System.Console.WriteLine("\n");

    // Copy the contents of the last 10 elements of byteArray1 to the
    // beginning of byteArray2.
    // The offset specifies where the copying begins in the source array.
    int offset = length - 10;
    Copy(byteArray1, offset, byteArray2, 0, length - offset);

    // Display the first 10 elements in the copy, byteArray2.
    System.Console.WriteLine("The first 10 elements of the copy are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray2[i] + " ");
    }
    System.Console.WriteLine("\n");
    /* Output:
        The first 10 elements of the original are:
        0 1 2 3 4 5 6 7 8 9

        The first 10 elements of the copy are:
        0 1 2 3 4 5 6 7 8 9

        The first 10 elements of the copy are:
        90 91 92 93 94 95 96 97 98 99
    */
}
```

Указатели функций

В C# имеются типы `delegate`, позволяющие определить безопасные объекты указателя функции. При вызове делегата создается экземпляр типа, производного от System.Delegate, и вызывается его виртуальный метод `Invoke`. В этом виртуальном вызове используется инструкция IL `callvirt`. В критически важных с точки зрения производительности путях к коду использование инструкции IL `calli` является более эффективным.

Указатель функции можно определить, используя синтаксический элемент `delegate*`. Вместо того, чтобы создавать экземпляр объекта `delegate` и вызывать метод `Invoke`, компилятор вызывает такую функцию, используя инструкцию `calli`. В следующем коде объявляются два метода, которые используют `delegate` или `delegate*` для объединения двух объектов одного типа. В первом методе используется тип делегата System.Func<T1,T2,TResult>. Во втором методе используется объявление `delegate*` с теми же параметрами и типом возвращаемого значения:

```
public static T Combine<T>(Func<T, T, T> combinator, T left, T right) =>
    combinator(left, right);

public static T UnsafeCombine<T>(delegate*<T, T, T> combinator, T left, T right) =>
    combinator(left, right);
```

В следующем коде показано, как объявить статическую локальную функцию и вызвать метод `UnsafeCombine`, используя указатель на эту локальную функцию:

```
static int localMultiply(int x, int y) => x * y;
int product = UnsafeCombine(&localMultiply, 3, 4);
```

В приведенном выше коде иллюстрируется ряд правил работы с функциями, доступ к которым осуществляется по указателю:

- Указатели функций могут быть объявлены только в контексте `unsafe`.
- Методы, принимающие в качестве параметра значение типа `delegate*` (или возвращающие значение типа `delegate*`), могут вызываться только в контексте `unsafe`.
- Оператор `&` для получения адреса функции допускается только для функций `static`. (Это правило применяется как к функциям-членам, так и к локальным функциям).

Синтаксис имеет сходства с объявлением типов `delegate` и использованием указателей. Суффикс `*` в служебном слове `delegate` указывает на то, что данное объявление является *указателем функции*. Знак `&` при назначении группы методов указателю функции указывает, что операция использует адрес метода.

Для `delegate*` можно указать соглашение о вызовах, используя ключевые слова `managed` и `unmanaged`. Кроме того, соглашение о вызовах можно указать для указателей на функции `unmanaged`. Примеры для всех этих случаев можно увидеть в объявлениях ниже. В первом объявлении используется соглашение о вызовах `managed`, которое используется по умолчанию. В следующих трех используется соглашение о вызовах `unmanaged`. В каждом из них указано одно из соглашений о вызовах из стандарта ECMA 335: `Cdecl`, `Stdcall`, `Fastcall` или `Thiscall`. В последнем объявлении используется соглашение о вызовах `unmanaged`, что дает возможность среде CLR выбрать используемое по умолчанию соглашение о вызовах для данной платформы. Среда CLR выберет соглашение о вызовах во время выполнения кода.

```
public static T ManagedCombine<T>(delegate* managed<T, T, T> combinator, T left, T right) =>
    combinator(left, right);
public static T CDeclCombine<T>(delegate* unmanaged[Cdecl]<T, T, T> combinator, T left, T right) =>
    combinator(left, right);
public static T StdcallCombine<T>(delegate* unmanaged[Stdcall]<T, T, T> combinator, T left, T right) =>
    combinator(left, right);
public static T FastcallCombine<T>(delegate* unmanaged[Fastcall]<T, T, T> combinator, T left, T right) =>
    combinator(left, right);
public static T ThiscallCombine<T>(delegate* unmanaged[Thiscall]<T, T, T> combinator, T left, T right) =>
    combinator(left, right);
public static T UnmanagedCombine<T>(delegate* unmanaged<T, T, T> combinator, T left, T right) =>
    combinator(left, right);
```

Дополнительные сведения об указателях функций см. в предложении Указатель функции для C# 9.0.

Спецификация языка C#

Дополнительные сведения см. в разделе Небезопасный код спецификации языка C#.

Директивы препроцессора C#

18.12.2021 • 12 minutes to read

Хотя у компилятора нет отдельного препроцессора, директивы, описанные в этом разделе, обрабатываются так, как если бы он был. Они используются в условной компиляции. В отличие от директив C и C++ вы не можете использовать их для создания макросов. Директива препроцессора должна быть единственной инструкцией в строке.

Контекст, допускающий значение NULL

Директива препроцессора `#nullable` устанавливает контекст с *заметками о допустимости значений NULL* и *контекст с предупреждениями о допустимости значений NULL*. Эта директива определяет, действуют ли заметки, допускающие значение NULL, и могут ли быть заданы предупреждения о допустимости значений NULL. Каждый контекст либо *отключен*, либо *включен*.

Оба контекста можно указать на уровне проекта (за пределами исходного кода C#). Директива `#nullable` управляет контекстами заметок и предупреждений и имеет приоритет над параметрами уровня проекта. Директива задает контексты, которыми управляет, пока другая директива не переопределит ее, или до конца исходного файла.

Ниже приведены результаты использования директив:

- `#nullable disable`: задает контексты с заметками и предупреждениями о допустимости значения NULL в значение *отключено*.
- `#nullable enable`: задает контексты с заметками и предупреждениями о допустимости значения NULL в значение *включено*.
- `#nullable restore`: восстанавливает контексты с заметками и предупреждениями о допустимости значения NULL до параметров проекта.
- `#nullable disable annotations`: устанавливает контекст с заметками о допустимости значения NULL в режим *отключено*.
- `#nullable enable annotations`: устанавливает контекст с заметками о допустимости значения NULL в режим *включено*.
- `#nullable restore annotations`: восстанавливает контексты с заметками о допустимости значения NULL до параметров проекта.
- `#nullable disable warnings`: устанавливает контекст с предупреждениями о допустимости значения NULL в режим *отключено*.
- `#nullable enable warnings`: устанавливает контекст с предупреждениями о допустимости значения NULL в режим *включено*.
- `#nullable restore warnings`: восстанавливает контексты с предупреждениями о допустимости значения NULL до параметров проекта.

Условная компиляция

Для управления условной компиляцией используются четыре директивы препроцессора.

- `#if`: открывает условную компиляцию, где код компилируется, только если определен указанный символ.
- `#elif`: закрывает предыдущую условную компиляцию и открывает новую на основе того, определен ли указанный символ.

- `#else`: закрывает предыдущую условную компиляцию и открывает новую, если указанный символ не определен.
- `#endif`: закрывает предыдущую условную компиляцию.

Когда компилятор C# встречает директиву `#if`, за которой следует директива `#endif`, код между этими директивами он компилирует, только когда определен указанный символ. В отличие от C и C++ здесь невозможно назначить символу числовое значение. Оператор `#if` в C# является логическим. Он проверяет только одно условие — определен ли указанный символ. Пример:

```
#if DEBUG
    Console.WriteLine("Debug version");
#endif
```

Вы можете использовать операторы `==` (равенство) и `!=` (неравенство) для проверки значений `bool` `true` или `false`. Значение `true` означает, что символ определен. Инструкция `#if DEBUG` имеет то же значение, что и `#if (DEBUG == true)`. Вы можете использовать операторы `&&` (и), `||` (или) и `!` (не), чтобы узнать, определено ли несколько символов. Можно также группировать символы и операторы при помощи скобок.

`#if`, как и директивы `#else`, `#elif`, `#endif`, `#define` и `#undef`, позволяет включить или исключить код в зависимости от существования одного или нескольких символов. Условная компиляция может быть полезной при компиляции кода для отладочной сборки или для определенной конфигурации.

Условные директивы, начинающиеся с директивы `#if`, должны явным образом завершаться директивой `#endif`. `#define` позволяет определить символ, чтобы выражение, в качестве которого этот символ передается в директиву `#if`, при вычислении давало значение `true`. Символ также можно определить с помощью параметра компилятора DefineConstants. Для отмены определения символа служит директива `#undef`. Символ, создаваемый с помощью `#define`, будет определен в пределах того файл, в котором он определен. Символ, определенный с помощью DefineConstants или `#define`, не конфликтует с одноименной переменной. Соответственно, имя переменной не должно передаваться директиве препроцессора, а символ может использоваться только в директиве препроцессора.

Директива `#elif` позволяет создать составную условную директиву. Выражение `#elif` будет вычисляться в том случае, если ни одна из предшествующих директив `#if` или необязательных директив `#elif` после вычисления выражения не возвращает значение `true`. Если после вычисления выражения `#elif` возвращается значение `true`, компилятор вычисляет весь код между директивой `#elif` и следующей условной директивой. Пример:

```
#define VC7
//...
#if debug
    Console.WriteLine("Debug build");
#elif VC7
    Console.WriteLine("Visual Studio 7");
#endif
```

С помощью директивы `#else` можно создать составную условную директиву со следующим поведением: если ни одно из выражений в предшествующих директивах `#if` или (необязательно) `#elif` не принимает значение `true`, компилятор вычисляет код между директивой `#else` и последующей директивой `#endif`. Директива `#endif` обязательно указывается в качестве следующей директивы препроцессора после `#else`.

`#endif` указывает на конец условной директивы, начало которой было задано с помощью директивы `#if`.

Система сборки также учитывает символы препроцессора, представляющие целевые платформы в проектах в стиле SDK. Они полезны при создании приложений, предназначенных для нескольких версий .NET.

ТРЕБУЕМЫЕ ВЕРСИИ .NET FRAMEWORK	СИМВОЛЫ	ДОПОЛНИТЕЛЬНЫЕ СИМВОЛЫ, ДОСТУПНЫЕ В ПАКЕТЕ SDK ДЛЯ .NET 5 И БОЛЕЕ ПОЗДНИХ ВЕРСИЙ
.NET Framework	`NETFRAMEWORK`, `NET48`, `NET472`, `NET471`, `NET47`, `NET462`, `NET461`, `NET46`, `NET452`, `NET451`, `NET45`, `NET40`, `NET35`, `NET20`	`NET48_OR_GREATER`, `NET472_OR_GREATER`, `NET471_OR_GREATER`, `NET47_OR_GREATER`, `NET462_OR_GREATER`, `NET461_OR_GREATER`, `NET46_OR_GREATER`, `NET452_OR_GREATER`, `NET451_OR_GREATER`, `NET45_OR_GREATER`, `NET40_OR_GREATER`, `NET35_OR_GREATER`, `NET20_OR_GREATER`
.NET Standard	`NETSTANDARD`, `NETSTANDARD2_1`, `NETSTANDARD2_0`, `NETSTANDARD1_6`, `NETSTANDARD1_5`, `NETSTANDARD1_4`, `NETSTANDARD1_3`, `NETSTANDARD1_2`, `NETSTANDARD1_1`, `NETSTANDARD1_0`	`NETSTANDARD2_1_OR_GREATER`, `NETSTANDARD2_0_OR_GREATER`, `NETSTANDARD1_6_OR_GREATER`, `NETSTANDARD1_5_OR_GREATER`, `NETSTANDARD1_4_OR_GREATER`, `NETSTANDARD1_3_OR_GREATER`, `NETSTANDARD1_2_OR_GREATER`, `NETSTANDARD1_1_OR_GREATER`, `NETSTANDARD1_0_OR_GREATER`
.NET 5+ (и .NET Core)	`NET`, `NET6_0`, `NET6_0_ANDROID`, `NET6_0_IOS`, `NET6_0_MACOS`, `NET6_0_MACCATALYST`, `NET6_0_TVOS`, `NET6_0_WINDOWS`, `NET5_0`, `NETCOREAPP`, `NETCOREAPP3_1`, `NETCOREAPP3_0`, `NETCOREAPP2_2`, `NETCOREAPP2_1`, `NETCOREAPP2_0`, `NETCOREAPP1_1`, `NETCOREAPP1_0`	`NET6_0_OR_GREATER`, `NET6_0_ANDROID_OR_GREATER`, `NET6_0_IOS_OR_GREATER`, `NET6_0_MACOS_OR_GREATER`, `NET6_0_MACCATALYST_OR_GREATER`, `NET6_0_TVOS_OR_GREATER`, `NET6_0_WINDOWS_OR_GREATER`, `NET5_0_OR_GREATER`, `NETCOREAPP_OR_GREATER`, `NETCOREAPP3_1_OR_GREATER`, `NETCOREAPP3_0_OR_GREATER`, `NETCOREAPP2_2_OR_GREATER`, `NETCOREAPP2_1_OR_GREATER`, `NETCOREAPP2_0_OR_GREATER`, `NETCOREAPP1_1_OR_GREATER`, `NETCOREAPP1_0_OR_GREATER`

NOTE

- Символы без привязки к версии определены независимо от версии, которую вы хотите использовать в качестве целевой.
- Символы для определенных версий определены только для тех версий, которые вы хотите использовать в качестве целевых.
- Символы `<framework>_OR_GREATER` определены для версии, которую вы хотите использовать в качестве целевой, и всех более ранних версий. Например, если вы выбрали .NET Framework 2.0, определяются следующие символы: `NET_2_0`, `NET_2_0_OR_GREATER`, `NET_1_1_OR_GREATER` и `NET_1_0_OR_GREATER`.

NOTE

Для традиционных проектов, в которых не используется пакет SDK, необходимо вручную настроить символы условной компиляции для различных целевых платформ в Visual Studio с помощью страниц свойств проекта.

Другие предопределенные символы включают константы `DEBUG` и `TRACE`. Вы можете переопределить значения для проектов с помощью `#define`. Например, символ DEBUG автоматически устанавливается в зависимости от свойств конфигурации сборки (в режиме отладки или выпуска).

В следующем примере показано, как определить символ `MYTEST` в файле и затем протестировать значения символов `MYTEST` и `DEBUG`. Выходные данные этого примера зависят от режима конфигурации, в котором создан проект (**Отладка** или **Выпуск**).

```
#define MYTEST
using System;
public class MyClass
{
    static void Main()
    {
#if (DEBUG && !MYTEST)
        Console.WriteLine("DEBUG is defined");
#elif (!DEBUG && MYTEST)
        Console.WriteLine("MYTEST is defined");
#elif (DEBUG && MYTEST)
        Console.WriteLine("DEBUG and MYTEST are defined");
#else
        Console.WriteLine("DEBUG and MYTEST are not defined");
#endif
    }
}
```

В следующем примере показано, как тестировать разные целевые платформы для использования более новых интерфейсов API, когда это возможно:

```
public class MyClass
{
    static void Main()
    {
#if NET40
        WebClient _client = new WebClient();
#else
        HttpClient _client = new HttpClient();
#endif
    }
    //...
}
```

Определение символов

Используйте следующие две директивы препроцессора, чтобы определить или отменить определение символов для условной компиляции.

- `#define`: определение символа.
- `#undef`: подчеркивание символа.

`#define` позволяет определить символ. При использовании символа в качестве выражения, которое передается директиве `#if`, выражение будет иметь значение `true`, как показано в следующем примере:

```
#define VERBOSE

#if VERBOSE
   Console.WriteLine("Verbose output version");
#endif
```

> **NOTE**
> Директиву `#define` нельзя использовать для объявления значений констант, как это обычно делается в C и C++. Для определения констант в C# следует использовать статические элементы класса или структуры. При наличии нескольких констант имеет смысл создать для них отдельный класс "Constants".

Символы можно использовать для указания условий компиляции. Для проверки символов можно использовать директивы `#if` или `#elif`. Для условной компиляции также можно использовать ConditionalAttribute. Вы можете определить символ, но не можете присвоить символу значение. Директива `#define` должна находиться в файле перед использованием любых инструкций, которые также не являются директивами препроцессора. Символ также можно определить с помощью параметра компилятора DefineConstants. Для отмены определения символа служит директива `#undef`.

Определение областей

Вы можете определить области кода, которые можно свернуть в структуру, используя следующие две директивы препроцессора.

- `#region`: начало области.
- `#endregion`: конец области.

Директива `#region` позволяет указать блок кода, который можно разворачивать и сворачивать с помощью функции структурирования в редакторе кода. В больших файлах кода удобно сворачивать или скрывать одну область или несколько, чтобы не отвлекаться от той части файла, над которой в настоящее время идет работа. В следующем примере показано, как определить область:

```
#region MyClass definition
public class MyClass
{
    static void Main()
    {
    }
}
#endregion
```

В конце блока `#region` должна присутствовать директива `#endregion`. Блок `#region` не может накладываться на блок `#if`. Однако блок `#region` можно вложить в блок `#if`, а блок `#if` — в блок

`#region`.

Сведения об ошибках и предупреждениях

Вы указываете компилятору создавать определенные пользователем ошибки и предупреждения компилятора, а также управлять сведениями о строках с помощью следующих директив.

- `#error`: создание ошибки компилятора с указанным сообщением.
- `#warning`: создание предупреждения компилятора с конкретным сообщением.
- `#line`: изменение номера строки, выводимого с сообщениями компилятора.

`#error` позволяет создать определяемую пользователем ошибку CS1029 из определенного места в коде. Пример:

```
#error Deprecated code in this method.
```

> **NOTE**
> Компилятор обрабатывает `#error version` особым образом и сообщает об ошибке компилятора CS8304 с сообщением, содержащим используемые версии компилятора и языка.

`#warning` позволяет создать предупреждение компилятора CS1030 первого уровня из определенного места в коде. Пример:

```
#warning Deprecated code in this method.
```

Директива `#line` позволяет изменять номер строки компилятора и при необходимости имя файла, в который будут выводиться ошибки и предупреждения.

В следующем примере показано, как включить в отчет два предупреждения, связанные с номерами строк. Директива `#line 200` принудительно устанавливает номер следующей строки 200 (по умолчанию используется номер 6). До выполнения следующей директивы `#line` в отчете будет указываться имя файла Special. Директива `#line default` по умолчанию восстанавливает нумерацию строк в исходное состояние с учетом строк, номера которых были изменены с помощью предшествующей директивы.

```
class MainClass
{
    static void Main()
    {
#line 200 "Special"
        int i;
        int j;
#line default
        char c;
        float f;
#line hidden // numbering not affected
        string s;
        double d;
    }
}
```

В результате компиляции формируются следующие результаты:

```
Special(200,13): warning CS0168: The variable 'i' is declared but never used
Special(201,13): warning CS0168: The variable 'j' is declared but never used
MainClass.cs(9,14): warning CS0168: The variable 'c' is declared but never used
MainClass.cs(10,15): warning CS0168: The variable 'f' is declared but never used
MainClass.cs(12,16): warning CS0168: The variable 's' is declared but never used
MainClass.cs(13,16): warning CS0168: The variable 'd' is declared but never used
```

Директива `#line` может использоваться на автоматизированном промежуточном этапе процесса построения. Например, если строки были удалены из первоначального файла с исходным кодом, но вам по-прежнему требуется создавать выходные файлы компилятора на основе изначальной нумерации строк в файле, можно удалить строки и затем смоделировать их первичную нумерацию с помощью директивы `#line`.

Директива `#line hidden` скрывает последующие строки для отладчика. В этом случае при пошаговой проверке кода разработчиком все строки между `#line hidden` и следующей директивой `#line` (кроме случаев, когда это также директива `#line hidden`) будут пропущены. Этот параметр также можно использовать для того, чтобы дать ASP.NET возможность различать определяемый пользователем и создаваемый компьютером код. В основном эта функция используется в ASP.NET, но также может быть полезна и в других генераторах исходного кода.

Директива `#line hidden` не влияет на имена файлов и номера строк в отчетах об ошибках. Это значит, что при обнаружении ошибки в скрытом блоке компилятор укажет в отчете текущие имя файла и номер строки, где найдена ошибка.

Директива `#line filename` задает имя файла, которое будет отображаться в выходных данных компилятора. По умолчанию используется фактическое имя файла с исходным кодом. Имя файла должно заключаться в двойные кавычки (" "). Перед ним должен указываться номер строки.

Начиная с C# 10 можно использовать новую форму директивы `#line`:

```
#line (1, 1) - (5, 60) 10 "partial-class.g.cs"
/*34567*/int b = 0;
```

Компоненты этой формы:

- `(1, 1)`: начальная строка и столбец для первого символа строки, следующей за директивой. В этом примере следующая строка будет отображаться как строка 1, столбец 1.
- `(5, 60)`: конечная строка и столбец для помеченной области.
- `10`: смещение столбца, чтобы директива `#line` вступила в силу. В этом примере в качестве столбца 1 будет отображаться десятый столбец. Здесь начинается объявление `int b = 0;`. Это поле является необязательным. Если этот параметр опущен, директива вступает в силу в первом столбце.
- `"partial-class.g.cs"`: имя выходного файла.

В предыдущем примере будет создано следующее предупреждение:

```
partial-class.g.cs(1,5,1,6): warning CS0219: The variable 'b' is assigned but its value is never used
```

После повторного сопоставления переменная `b` находится в первой строке, в шестом символе.

Предметно-ориентированные языки (DSL) обычно используют этот формат, чтобы обеспечить более эффективное сопоставление исходного файла с созданными выходными данными C#. Дополнительные примеры этого формата см. в разделе примеров в спецификации функции.

Директивы pragma

Директива `#pragma` предоставляет компилятору специальные инструкции для компиляции файла, в котором она появляется. Компилятор должен поддерживать эти инструкции. Другими словами, директиву `#pragma` невозможно использовать для создания настраиваемых инструкций предварительной обработки.

- `#pragma warning`: включение или отключение предупреждений.
- `#pragma checksum`: создание контрольной суммы.

```
#pragma pragma-name pragma-arguments
```

`pragma-name` — имя распознанной прагмы, а `pragma-arguments` — аргументы, относящиеся к прагме.

#pragma warning

`#pragma warning` может включать или отключать определенные предупреждения.

```
#pragma warning disable warning-list
#pragma warning restore warning-list
```

`warning-list` — список номеров предупреждений с разделителем-запятой. Префикс CS является необязательным. Если номера предупреждений не указаны, `disable` отключает все предупреждения, а `restore` включает все предупреждения.

> **NOTE**
> Чтобы найти номера предупреждений в Visual Studio, выполните сборку проекта, а затем поиск номеров предупреждений в окне **Вывод**.

Параметр `disable` вступает в силу, начиная со следующей строки исходного файла. Предупреждение восстанавливается в строке после `restore`. Если в файле нет `restore`, предупреждения восстанавливаются до их состояния по умолчанию в первой строке всех последующих файлов в той же компиляции.

```
// pragma_warning.cs
using System;

#pragma warning disable 414, CS3021
[CLSCompliant(false)]
public class C
{
    int i = 1;
    static void Main()
    {
    }
}
#pragma warning restore CS3021
[CLSCompliant(false)]  // CS3021
public class D
{
    int i = 1;
    public static void F()
    {
    }
}
```

#pragma checksum

Создает контрольные суммы для исходных файлов, чтобы помочь с отладкой страниц ASP.NET.

```
#pragma checksum "filename" "{guid}" "checksum bytes"
```

`"filename"` — это имя файла, для которого требуется наблюдение за изменениями или обновлениями, `"{guid}"` — глобальный уникальный идентификатор (GUID) для хэш-алгоритма, а `"checksum_bytes"` — строка шестнадцатеричных цифр, представляющих байты контрольной суммы. Должно быть четным числом шестнадцатеричных цифр. Нечетное число цифр приведет к выводу предупреждения во время компиляции, и директива будет пропущена.

Отладчик Visual Studio использует контрольную сумму, чтобы подтвердить нахождение правильного источника. Компилятор вычисляет контрольную сумму для исходного файла, а затем передает результат в файл базы данных (PDB) программы. Отладчик затем использует PDB-файл для сравнения с контрольной суммой, вычисленной им для исходного файла.

Это решение не работает для проектов ASP.NET, так как рассчитанная контрольная сумма относится к созданному исходному файлу, а не файлу ASPX. Чтобы решить эту проблему, `#pragma checksum` предоставляет поддержку контрольных сумм для страниц ASP.NET.

При создании проекта ASP.NET в Visual C# созданный исходный файл содержит контрольную сумму для ASPX-файла, из которого создается источник. Затем компилятор записывает эти данные в PDB-файл.

Если компилятор не обнаруживает директиву `#pragma checksum` в файле, он вычисляет контрольную сумму и записывает значение в PDB-файл.

```
class TestClass
{
    static int Main()
    {
        #pragma checksum "file.cs" "{406EA660-64CF-4C82-B6F0-42D48172A799}" "ab007f1d23d9" // New checksum
    }
}
```

Параметры компилятора C#

18.12.2021 • 2 minutes to read

В этом разделе описаны параметры, интерпретируемые компилятором C#. Параметры сгруппированы в отдельные статьи в зависимости от того, чем они управляют. Это могут быть, например, функции языка, создание кода и выходные данные. Для навигации по параметрам можно воспользоваться оглавлением.

Установка параметров

В проектах .NET параметры компилятора можно задать двумя способами:

- ***В файле** *CSPROJ*

  Можно добавить свойства MSBuild для любого параметра компилятора в файле **CSPROJ* в формате XML. Имя свойства совпадает с именем параметра компилятора. Значение свойства задает значение параметра компилятора. Например, в следующем фрагменте файла проекта задается свойство `LangVersion`.

  ```
  <PropertyGroup>
    <LangVersion>preview</LangVersion>
  </PropertyGroup>
  ```

  Дополнительные сведения о настройке параметров в файлах проекта см. в статье Справочник по MSBuild для проектов пакета SDK для .NET.

- ***Использование страниц свойств** Visual Studio*

  В Visual Studio доступны страницы свойств, на которых можно изменять свойства сборки. Дополнительные сведения см. в статье Управление свойствами проектов и решений (Windows) или Управление свойствами проектов и решений (Mac).

Проекты .NET Framework

> **IMPORTANT**
> Этот раздел относится только к проектам .NET Framework.

Кроме описанных выше способов, для проектов .NET Framework можно задать параметры компилятора с помощью двух дополнительных методов:

- **Аргументы командной строки для проектов .NET Framework**. Для сборки проектов .NET Framework используется файл *csc.exe*, а не команда `dotnet build`. Аргументы командной строки можно указать в *csc.exe* для проектов .NET Framework.
- **Скомпилированные страницы ASP.NET**. Проекты .NET Framework используют раздел файла *web.config* для компиляции страниц. Для новой системы сборки и проектов ASP.NET Core параметры берутся из файла проекта.

Некоторые параметры компилятора больше не используются в формате для *csc.exe* и проектов .NET Framework. Они используются в формате для новой системы MSBuild. В этом разделе применяется новый синтаксис. Обе версии приведены в верхней части каждой страницы. Для *csc.exe* все аргументы указываются после параметра и двоеточия. Например, параметр `-doc` будет выглядеть так:

```
-doc:DocFile.xml
```

Чтобы вызвать компилятор C#, следует ввести имя соответствующего исполняемого файла (*csc.exe*) в командной строке.

Кроме того, для проектов .NET Framework *csc.exe* можно запустить из командной строки. Каждый параметр компилятора можно использовать в двух формах записи: -**параметр** или /**параметр**. В проектах .NET Framework параметры для компиляции кода программной части указываются в файле *web.config*. Дополнительные сведения см. в разделе <compiler> Element.

Если используется окно **Командная строка разработчика для** Visual Studio, все необходимые переменные среды устанавливаются автоматически. Дополнительные сведения о получении доступа к этому инструменту см. в статье Командная строка разработчика и PowerShell для разработчиков.

Обычно исполняемый файл *csc.exe* расположен в папке Microsoft.NET\Framework\ *<Version>* в каталоге *Windows*. Расположение файла может зависеть от конкретной конфигурации компьютера. Если на компьютере установлено несколько версий .NET Framework, будет несколько версий этого файла. Дополнительные сведения о подобных вариантах установки см. в разделе Практическое руководство. Определение установленных версий платформы .NET Framework.

Параметры компилятора C# для правил языковых функций

18.12.2021 • 6 minutes to read

Следующие параметры определяют, как компилятор интерпретирует языковые функции. Новый синтаксис MSBuild выделен **полужирным шрифтом**. Для старого синтаксиса *csc.exe* используется формат `code style`.

- **CheckForOverflowUnderflow** / `-checked`: создание проверок на переполнение.
- **AllowUnsafeBlocks** / `-unsafe`: разрешение использовать небезопасный код.
- **DefineConstants** / `-define`: определение символов условной компиляции.
- **LangVersion** / `-langversion`: указание версии языка, например `default` (последняя основная версия) или `latest` (последняя версия, включая дополнительные версии).
- **Nullable** / `-nullable`: включение контекста, допускающего значение NULL, или предупреждений, допускающих значение NULL.

CheckForOverflowUnderflow

Параметр **CheckForOverflowUnderflow** указывает, будет ли использование целочисленного арифметического оператора, в результате выполнения которого получено значение, выходящее за установленный для определенного типа данных диапазон значений, приводить к возникновению исключения во время выполнения.

```
<CheckForOverflowUnderflow>true</CheckForOverflowUnderflow>
```

Действие параметра **CheckForOverflowUnderflow** не распространяется на целочисленный арифметический оператор, находящийся в области действия ключевых слов `checked` или `unchecked`. Если в результате выполнения целочисленного арифметического оператора, находящегося за пределами области действия ключевых слов `checked` или `unchecked`, получено значение, выходящее за установленный для определенного типа данных диапазон значений, и для параметра **CheckForOverflowUnderflow** установлено значение `true`, во время выполнения возникнет исключение из-за этого оператора. Если параметр **CheckForOverflowUnderflow** имеет значение `false`, использование такого оператора не приводит к возникновению исключения во время выполнения. Значение этого параметра по умолчанию — `false`; проверка переполнения отключена.

AllowUnsafeBlocks

Параметр **AllowUnsafeBlocks** разрешает компилировать код, в котором используется ключевое слово unsafe. Значение по умолчанию для этого параметра — `false`, то есть использование небезопасного кода запрещено.

```
<AllowUnsafeBlocks>true</AllowUnsafeBlocks>
```

Дополнительные сведения см. в разделе Небезопасный код и указатели.

DefineConstants

Параметр **DefineConstants** определяет символы в файлах исходного кода вашей программы.

```
<DefineConstants>name;name2</DefineConstants>
```

Этот параметр задает имена одного или нескольких символов, которые необходимо определить. Параметр **DefineConstants** действует так же, как директива препроцессора #define, но применяется ко всем файлам проекта. Символ остается определенным в файле исходного кода до тех пор, пока определение не будет отменено с помощью директивы #undef в файле исходного кода. При использовании параметра `-define` директива `#undef` в одном файле не действует для других файлов исходного кода в проекте. Вы можете использовать символы, созданные этим параметром, с директивами #if, #else, #elif и #endif для условной компиляции исходных файлов. Компилятор C# сам по себе не определяет символы и макросы, которые можно использовать в исходном коде. Все такие определения задаются пользователем.

> **NOTE**
> Директива C# `#define` не поддерживает присваивание значения символу, как, например, в языке C++. Например, с помощью директивы `#define` нельзя создать макрос или определить константу. Чтобы определить константу, используйте переменную `enum`. Для создания макросов в стиле C++ необходимо использовать альтернативные способы, например универсальные шаблоны. Поскольку макросы по своей природе подвержены ошибкам, в C# они запрещены, однако вместо них предлагаются более безопасные альтернативы.

LangVersion

Инструктирует компилятор принимать только синтаксис, включенный в заданную спецификацию языка C#.

```
<LangVersion>9.0</LangVersion>
```

Допустимы следующие значения.

ЗНАЧЕНИЕ	ЗНАЧЕНИЕ
`preview`	Компилятор допускает использование любого допустимого синтаксиса языка из последней предварительной версии.
`latest`	Компилятор принимает синтаксис из последней выпущенной версии компилятора (включая дополнительный номер версии).
`latestMajor` (`default`)	Компилятор принимает синтаксис из последней основной версии компилятора.
`10.0`	Компилятор принимает только синтаксис, включенный в спецификацию C# 10 или более ранних версий.
`9.0`	Компилятор принимает только синтаксис, включенный в спецификацию C# 9 или более ранних версий.
`8.0`	Компилятор принимает только синтаксис, включенный в спецификацию C# 8.0 или более ранней версии.

ЗНАЧЕНИЕ	ЗНАЧЕНИЕ
`7.3`	Компилятор принимает только синтаксис, включенный в спецификацию C# 7.3 или более ранней версии.
`7.2`	Компилятор принимает только синтаксис, включенный в спецификацию C# 7.2 или более ранней версии.
`7.1`	Компилятор принимает только синтаксис, включенный в спецификацию C# 7.1 или более ранней версии.
`7`	Компилятор принимает только синтаксис, включенный в спецификацию C# 7.0 или более ранней версии.
`6`	Компилятор принимает только синтаксис, включенный в спецификацию C# 6.0 или более ранней версии.
`5`	Компилятор принимает только синтаксис, включенный в спецификацию C# 5.0 или более ранней версии.
`4`	Компилятор принимает только синтаксис, включенный в спецификацию C# 4.0 или более ранней версии.
`3`	Компилятор принимает только синтаксис, включенный в спецификацию C# 3.0 или более ранней версии.
`ISO-2` (или `2`)	Компилятор принимает только синтаксис, включенный в спецификацию ISO/IEC 23270:2006 C# (2.0).
`ISO-1` (или `1`)	Компилятор принимает только синтаксис, включенный в спецификацию ISO/IEC 23270:2003 C# (1.0/1.2).

Версия языка по умолчанию зависит от целевой платформы приложения, а также от установленной версии пакета SDK или Visual Studio. Такие правила определены в статье Управление версиями языка C#.

Параметр компилятора **LangVersion** не влияет на метаданные, на которые ссылается ваше приложение C#.

Так как каждая версия компилятора C# содержит расширения для спецификации языка, параметр **LangVersion** не позволяет использовать возможности, аналогичные возможностям более ранней версии компилятора.

Кроме того, в отличие от обновлений версии C#, которые обычно совпадают с основными выпусками .NET Framework, новые функции и возможности синтаксиса могут быть не привязаны к конкретной версии этой платформы. Для работы с новыми версиями требуется обновление компилятора, которое также выпускается с новой редакцией C#. Тем не менее для каждой функции определен собственный набор минимальных требований к API .NET или общеязыковой среде выполнения, в результате чего их можно выполнять на более ранних версиях платформы, добавив необходимые пакеты NuGet или другие библиотеки.

Независимо от того, какой параметр **LangVersion** вы задали, используйте для создания файлов с расширением .exe или .dll. текущую версию среды CLR. Единственным исключением являются дружественные сборки и **ModuleAssemblyName**, которые работают при установке параметра -

langversion:ISO-1.

Другие способы указания версии языка C# см. в статье Управление версиями языка C#.

Дополнительные сведения об установке этого параметра компилятора программным путем см. в разделе LanguageVersion.

Спецификация языка C#

VERSION	ССЫЛКА	ОПИСАНИЕ
C# 7.0 и более поздние версии		В настоящее время недоступно.
C# 6.0	Ссылка	Спецификация языка C# версии 6 (неофициальный проект): .NET Foundation
C# 5.0	Загрузить PDF-файл	Стандарт ECMA-334, 5-й выпуск
C# 3.0	Загрузить DOC-файл	Спецификация языка C#, версия 3.0: Microsoft Corporation
C# 2.0	Загрузить PDF-файл	Стандарт ECMA-334, 4-й выпуск
C# 1.2	Загрузить DOC-файл	Спецификация языка C#, версия 1.2: Microsoft Corporation
C# 1.0	Загрузить DOC-файл	Спецификация языка C#, версия 1.0: Microsoft Corporation

Минимальная версия пакета SDK, необходимая для поддержки всех возможностей языка

В следующей таблице перечислены минимальные версии пакета SDK с компилятором C#, поддерживающим соответствующую версию языка:

ВЕРСИЯ C#	МИНИМАЛЬНАЯ ВЕРСИЯ ПАКЕТА SDK
C# 10	Microsoft Visual Studio/Build Tools 2022 или пакет SDK для .NET 6
C# 9.0	Microsoft Visual Studio/Build Tools 2019, версия 16.8 или пакет SDK для .NET 5
C# 8.0	Microsoft Visual Studio/Build Tools 2019, версия 16.3 или пакет SDK .NET Core 3.0
C# 7.3	Microsoft Visual Studio/Build Tools 2017, версия 15.7
C# 7.2	Microsoft Visual Studio/Build Tools 2017, версия 15.5
C# 7.1	Microsoft Visual Studio/Build Tools 2017, версия 15.3
C# 7.0	Microsoft Visual Studio/Build Tools 2017
C# 6	Microsoft Visual Studio/Build Tools 2015

ВЕРСИЯ C#	МИНИМАЛЬНАЯ ВЕРСИЯ ПАКЕТА SDK
C# 5	Microsoft Visual Studio/Build Tools 2012 или встроенный компилятор .NET Framework 4.5
C# 4	Microsoft Visual Studio/Build Tools 2010 или встроенный компилятор .NET Framework 4.0
C# 3	Microsoft Visual Studio/Build Tools 2008 или встроенный компилятор .NET Framework 3.5
C# 2	Microsoft Visual Studio/Build Tools 2005 или встроенный компилятор .NET Framework 2.0
C# 1.0/1.2	Microsoft Visual Studio/Build Tools .NET 2002 или встроенный компилятор .NET Framework 1.0

Допускает значения NULL

Параметр **Nullable** позволяет указать контекст, допускающий значение NULL. `disable` является значением по умолчанию для данного параметра.

```
<Nullable>enable</Nullable>
```

Аргумент должен иметь одно из следующих значений: `enable`, `disable`, `warnings` или `annotations`. Аргумент `enable` разрешает контекст, допускающий значение NULL. При указании `disable` контекст, допускающий значение NULL, будет отключен. Аргумент `warnings` включает контекст предупреждения, допускающий значение NULL. При указании аргумента `annotations` будет включен контекст заметок, допускающий значение NULL.

Анализ потока используется для определения допустимости значений NULL в переменных в исполняемом коде. Выводимая допустимость переменной значения NULL не зависит от объявленной в переменной допустимости значения NULL. Вызовы методов анализируются, даже если они условно опущены. Например, Debug.Assert в режиме выпуска.

Вызов методов, снабженных следующими атрибутами, также повлияет на анализ потока:

- Простые предварительные условия: AllowNullAttribute и DisallowNullAttribute
- Простые постусловия: MaybeNullAttribute и NotNullAttribute
- Условные постусловия: MaybeNullWhenAttribute и NotNullWhenAttribute
- DoesNotReturnIfAttribute (например, `DoesNotReturnIf(false)` для Debug.Assert) и DoesNotReturnAttribute
- NotNullIfNotNullAttribute
- Постусловия элемента: MemberNotNullAttribute(String) и MemberNotNullAttribute(String[])

IMPORTANT

Глобальный контекст, допускающий значения NULL, не применяется для созданных файлов кода. Независимо от этого параметра, контекст, допускающий значение NULL, *отключен* для любого исходного файла, помеченного как созданный. Существует четыре способа пометки файла как созданного:

1. В файле. editorconfig укажите `generated_code = true` в разделе, который применяется к этому файлу.
2. Вставьте `<auto-generated>` или `<auto-generated/>` в комментарий в верхней части файла. Он может находиться в любой строке комментария, однако блок комментариев должен быть первым элементом в файле.
3. Имя файла следует начинать с *TemporaryGeneratedFile_*
4. В конце имени файла следует указать *.designer.cs*, *.generated.cs*, *.g.cs* или *.g.i.cs*.

Генераторы могут явно использовать директиву препроцессора `#nullable`.

Параметры компилятора C#, которые управляют выводом компилятора

18.12.2021 • 7 minutes to read

Следующие параметры управляют созданием выходных данных компилятора.

MSBUILD	*CSC.EXE*	ОПИСАНИЕ
DocumentationFile	`-doc:`	Создание XML-файла документации из комментариев `///`.
OutputAssembly	`-out:`	Указание выходного файла сборки.
PlatformTarget	`-platform:`	Указание разрядности ЦП целевой платформы.
ProduceReferenceAssembly	`-refout:`	Создание базовой сборки.
TargetType	`-target:`	Указание типа выходной сборки.

DocumentationFile

Параметр **DocumentationFile** позволяет поместить комментарии из документации в XML-файл. Дополнительные сведения о документировании кода см. в статье Рекомендуемые теги для комментариев документации. Значение указывает путь к выходному XML-файлу. XML-файл содержит комментарии в файлах исходного кода компиляции.

```
<DocumentationFile>path/to/file.xml</DocumentationFile>
```

Файл исходного кода, содержащий метод Main или инструкции верхнего уровня, выводится в XML первым. Часто вы предпочтете использовать созданный XML-файл с IntelliSense. Имя *XML*-файла должно совпадать с именем сборки. *XML*-файл должен находиться в том же каталоге, что и сборка. Если в проект Visual Studio добавляется ссылка на сборку, *XML*-файл также будет найден. Дополнительные сведения о создании комментариев к коду см. в статье Вставка XML-комментариев для создания документации. Если при компиляции не используется параметр `<TargetType:Module>`, `file` будет содержать теги `<assembly>` и `</assembly>`, которые указывают имя файла, содержащего манифест сборки для выходного файла. Примеры см. в статье Практическое руководство. Использование XML-документации.

> **NOTE**
> Параметр **DocumentationFile** применяется ко всем файлам в проекте. Чтобы отключить предупреждения, связанные с комментариями документации для определенного файла или раздела кода, используйте директиву #pragma warning.

Этот параметр можно использовать в любом проекте в стиле пакета SDK для .NET. Дополнительные сведения см. в разделе о свойстве DocumentationFile.

OutputAssembly

Параметр **OutputAssembly** позволяет задать имя выходного файла. Выходной путь указывает на папку, куда помещаются выходные данные компилятора.

```
<OutputAssembly>folder</OutputAssembly>
```

Укажите полное имя и расширение файла, который требуется создать. Если не указать имя выходного файла, MSBuild использует имя проекта для указания имени выходной сборки. К проектам со старым стилем применяются следующие правила:

- EXE-файлу будет присвоено имя файла исходного кода, который содержит метод `Main` или инструкции верхнего уровня.
- DLL-файлы и NETMODULE-файлы берут имя из первого файла исходного кода.

Все модули, созданные в процессе компиляции, будут связаны со сборкой, полученной в результате этого процесса. С помощью ildasm.exe можно просмотреть связанные файлы в манифесте сборки.

Параметр компилятора **OutputAssembly** обязателен, если требуется установить EXE-файл в качестве целевого для дружественной сборки.

PlatformTarget

Указывает, в какой версии среды CLR может запускаться сборка.

```
<PlatformTarget>anycpu</PlatformTarget>
```

- **anycpu** (по умолчанию) позволяет компилировать сборку для запуска на любой платформе. Если возможно, приложение выполняется как 64-разрядный процесс, а если доступен только 32-разрядный режим, переключается на него.
- **anycpu32bitpreferred** (по умолчанию) позволяет компилировать сборку для запуска на любой платформе. Приложение выполняется в 32-разрядном режиме в системах, поддерживающих и 64-разрядные, и 32-разрядные приложения. Можно задать этот параметр только для проектов, предназначенных для .NET Framework 4.5 и выше.
- **ARM** компилирует сборку для выполнения на компьютере с процессором Advanced RISC Machine (ARM).
- **ARM64** компилирует сборку для выполнения 64-разрядной средой CLR на компьютере с процессором Advanced RISC Machine (ARM) с поддержкой набора инструкций A64.
- **x64** компилирует сборку для запуска в 64-разрядной среде CLR на компьютере, поддерживающем набор инструкций AMD64 или EM64T.
- **x86** компилирует сборку для запуска в 32-разрядной среде CLR с архитектурой x86.
- **Itanium** компилирует сборку для запуска в 64-разрядной среде CLR на компьютере с процессором Itanium.

В 64-разрядной ОС Windows:

- Сборки, скомпилированные с параметром **x86**, будут выполняться в 32-разрядной среде CLR в подсистеме WOW64.
- Библиотеки DLL, скомпилированные с использованием параметра **anycpu**, выполняются в той же среде CLR, в которую загружается процесс.
- Исполняемые файлы, скомпилированные с использованием параметра **anycpu**, выполняются в 64-разрядной среде CLR.

- Исполняемые файлы, скомпилированные с использованием параметра **anycpu32bitpreferred**, выполняются в 32-разрядной среде CLR.

Параметр **anycpu32bitpreferred** является допустимым только для исполняемых файлов (.exe). Он используется только с .NET Framework 4.5 и выше. Дополнительные сведения о разработке приложений для запуска в 64-разрядной операционной системе Windows см. в разделе 64-разрядные приложения.

Параметр **PlatformTarget** можно задать на странице свойств **сборки** для проекта в Visual Studio.

В .NET Core, .NET 5 и более поздних выпусках параметр **anycpu** имеет некоторые особенности. Если вы указали параметр **anycpu**, опубликуйте приложение и выполняйте его только с `dotnet.exe` x86 или только с `dotnet.exe` x64. Если вы используете автономные приложения, на этапе `dotnet publish` упаковывается исполняемый файл для настройки RID.

ProduceReferenceAssembly

Параметр **ProduceReferenceAssembly** указывает путь к файлу, в который нужно выводить базовую сборку. В API Emit он преобразуется в `metadataPeStream`. `filepath` указывает путь для базовой сборки. Обычно он совпадает с путем к файлу основной сборки. Согласно рекомендуемому соглашению (используемому в MSBuild), базовую сборку следует помещать во вложенную папку ref/ относительно основной сборки.

```
<ProduceReferenceAssembly>filepath</ProduceReferenceAssembly>
```

Базовые сборки являются особым типом сборки, которая содержит только минимальный объем метаданных, необходимый для представления общедоступного API библиотеки. Такие сборки включают в себя объявления для всех элементов, которые важны при указании ссылки на сборку в средствах сборки. Базовые сборки исключают все реализации элементов, а также объявления закрытых элементов, не имеющих наблюдаемого влияния на их контракт API. Дополнительные сведения см. в статье Базовые сборки в руководстве по .NET.

Параметры **ProduceReferenceAssembly** и **ProduceOnlyReferenceAssembly** являются взаимоисключающими.

TargetType

Параметр компилятора **TargetType** можно задать в одной из следующих форм:

- **library**: для создания библиотеки кода. **library** — это значение по умолчанию.
- **exe**: для создания EXE-файла.
- **module**: для создания модуля.
- **winexe**: для создания программы Windows.
- **winmdobj**: для создания промежуточного файла *.winmdobj*.
- **appcontainerexe**: для создания EXE-файла для приложений Магазина Windows 8.x.

> **NOTE**
> Если вы создаете приложения для .NET Framework и не указываете значение **module**, этот параметр приводит к помещению манифеста сборки .NET Framework в выходной файл. Дополнительные сведения см. в разделах Сборки в .NET и Общие атрибуты.

```
<TargetType>library</TargetType>
```

При каждой компиляции компилятор создает только один манифест сборки. В манифест сборки добавляются сведения обо всех файлах в компиляции. При создании нескольких выходных файлов в командной строке может быть создан только один манифест сборки, который добавляется в первый выходной файл, указанный в командной строке.

При создании сборки можно указать, что код полностью или частично CLS-совместим с атрибутом CLSCompliantAttribute.

библиотека

Параметр **library** указывает компилятору создавать библиотеку динамической компоновки (DLL), а не исполняемый файл (EXE). Библиотека DLL создается с расширением *.dll*. Выходной файл получает имя первого входного файла, если только с помощью параметра **OutputAssembly** не указано иное. При сборке *DLL*-файла метод `Main` не требуется.

exe

Параметр **exe** указывает компилятору создать исполняемое (EXE) консольное приложение. Исполняемый файл создается с расширением EXE. Используйте параметр **winexe** для создания исполняемого файла программы Windows. Если не указано иное с помощью параметра **OutputAssembly**, имя выходного файла совпадает с именем входного файла, который содержит точку входа (метод Main или инструкции верхнего уровня). В файлах исходного кода, который компилируется в EXE-файл, должна содержаться только одна точка входа. Если код содержит несколько классов с методом `Main`, указать, какой класс содержит метод `Main`, можно с помощью параметра компилятора **StartupObject**.

module

Этот параметр указывает компилятору не создавать манифест сборки. По умолчанию выходной файл, созданный при компиляции с этим параметром, имеет расширение *.netmodule*. Файл без манифеста сборки не может быть загружен в среду выполнения .NET. Но его можно включить в манифест сборки с помощью параметра **AddModules**. Если в рамках одной процедуры компиляции создается несколько модулей, типы internal из одного модуля будут доступны для других модулей, компилируемых вместе с ним. Если код одного модуля ссылается на типы `internal` в другом модуле, оба модуля нужно включить в манифест сборки с помощью параметра **AddModules**. Создание модуля в среде разработки Visual Studio не поддерживается.

winexe

Параметр **winexe** указывает компилятору создать исполняемый файл (EXE) программы Windows. Исполняемый файл создается с расширением EXE. Программа Windows предоставляет пользовательский интерфейс либо из библиотеки .NET, либо с помощью API-интерфейсов Windows. Воспользуйтесь параметром **exe**, чтобы создать консольное приложение. Если не указано иное с помощью параметра **OutputAssembly**, имя выходного файла совпадает с именем входного файла, который содержит метод `Main`. В файлах исходного кода, который компилируется в EXE-файл, должен содержаться один и только один метод `Main`. Если код содержит несколько классов с методом `Main`, указать, какой класс содержит метод `Main`, можно с помощью параметра **StartupObject**.

winmdobj

Если используется параметр **winmdobj**, компилятор создает промежуточный *WINMDOBJ*-файл, который можно преобразовать в двоичный *WINMD*-файл среды выполнения Windows. Затем *WINMD*-файл можно использовать в программах на языках JavaScript и C++ в дополнение к программам, использующим управляемые языки.

Параметр **winmdobj** сигнализирует компилятору, что необходим промежуточный модуль. Затем *WINMDOBJ*-файл можно передать с помощью инструмента экспорта WinMDExp для создания файла метаданных Windows (*WINCMD*-файл). *WINMD*-файл содержит код из исходной библиотеки и метаданные WinMD, используемые JavaScript, C++ и средой выполнения Windows. Выходные данные файла, скомпилированного с помощью параметра компилятора **winmdobj**, используются только в

качестве входных данных инструментом экспорта WimMDExp (на *WINMDOBJ*-файл нет прямой ссылки). Выходной файл получает имя первого входного файла, если только не используется параметр OutputAssembly. Метод `Main` не требуется.

appcontainerexe

Если используется параметр компилятора **appcontainerexe**, компилятор создает исполняемый файл Windows (*EXE*-файл), который должен запускаться в контейнере приложения. Этот параметр аналогичен -target:winexe, но предназначен для приложений Магазина Windows 8.x.

Этот параметр устанавливает бит в переносимом исполняемом файле (PE), чтобы обеспечить запуск приложения в контейнере. Если он установлен, при попытке запустить исполняемый файл вне контейнера приложения методом CreateProcess будет возникать ошибка. Выходной файл получает имя входного файла, содержащего метод `Main`, если только с помощью параметра OutputAssembly не указано иное.

Параметры компилятора C#, которые определяют входные данные

18.12.2021 • 4 minutes to read

Следующие параметры управляют входными данными компилятора. Новый синтаксис MSBuild выделен **полужирным шрифтом**. Для старого синтаксиса *csc.exe* используется формат `code style`.

- **References** / `-reference` или `-references`: создание ссылки на метаданные из указанного файла или файлов сборки.
- **AddModules** / `-addmodule`: добавление модуля (созданного с помощью `target:module` для этой сборки).
- **EmbedInteropTypes** / `-link`: встраивание метаданных из указанных файлов сборки взаимодействия.

Ссылки

Параметр **Reference** указывает компилятору импортировать сведения типа public из указанного файла в текущий проект. Это позволяет ссылаться на метаданные из указанных файлов сборки.

```
<Reference Include="filename" />
```

`filename` — это имя файла, который содержит манифест сборки. Чтобы импортировать данные из нескольких файлов, включите отдельный элемент **Reference** для каждого файла. Псевдоним можно определить как дочерний элемент элемента **Reference**:

```
<Reference Include="filename.dll">
  <Aliases>LS</Aliases>
</Reference>
```

В предыдущем примере `LS` — это допустимый идентификатор C# для представления корневого пространства имен, которое будет содержать все пространства имен в сборке *filename.dll*. Импортируемые файлы должны содержать манифест. Для указания каталога, в котором находятся одна или несколько ссылок на сборки, используйте AdditionalLibPaths. В разделе с описанием AdditionalLibPaths также рассматриваются каталоги, в которых компилятор ищет сборки. Чтобы компилятор мог распознавать тип в сборке (не в модуле), ему следует указать принудительно разрешать типы. Это можно сделать, определив экземпляр типа. Возможны и другие способы разрешения компилятором имен типов в сборке. Например, если тип наследуется от типа в сборке, его имя будет распознаваться компилятором. Иногда бывает необходимо сослаться на две различные версии одного компонента из одной сборки. Для этого используйте элемент **Aliases** в элементе **References** для каждого файла, чтобы различать два этих файла. Этот псевдоним используется в качестве квалификатора имени компонента и разрешается в компонент в одном из файлов.

> **NOTE**
>
> В Visual Studio используйте команду **Добавить ссылку**. Для получения дополнительной информации см. Практическое руководство. Добавление и удаление ссылок с помощью диспетчера ссылок.

AddModules

Установка этого параметра приводит к добавлению модуля, созданного с помощью параметра `<TargetType>module</TargetType>` для текущей компиляции:

```
<AddModule Include=file1 />
<AddModule Include=file2 />
```

Где `file`, `file2` — это выходные файлы, содержащие метаданные. В этот файл не может входить манифест сборки. Чтобы импортировать несколько файлов, разделите их имена запятыми или точками с запятой. Все модули, добавленные с помощью **AddModules**, во время выполнения должны находиться в том же каталоге, что и выходной файл. То есть во время компиляции можно указать модуль в любом каталоге, но во время выполнения он должен находиться в каталоге приложения. Если во время выполнения модуль отсутствует в каталоге приложения, возникнет исключение TypeLoadException. `file` не может содержать сборку. Например, если выходной файл был создан с помощью параметра **TargetType** **module**, для импорта его метаданных можно использовать **AddModules**.

Если выходной файл был создан с помощью параметра **TargetType**, отличающегося от **module**, для импорта его метаданных нельзя использовать **AddModules**, но можно использовать параметр **References**.

EmbedInteropTypes

Дает компилятору указание сделать всю информацию о типах COM из указанных сборок доступной компилируемому проекту.

```
<References>
  <EmbedInteropTypes>file1;file2;file3</EmbedInteropTypes>
</References>
```

Где `file1;file2;file3` — это список с именами файлов сборок, разделенными точкой с запятой. Если имя файла содержит пробел, заключите его в кавычки. Параметр **EmbedInteropTypes** позволяет развернуть приложение, содержащее сведения о внедренном типе. После этого приложение может использовать типы из сборки среды выполнения, реализующей информацию о внедренных типах, без ссылки на эту сборку. Если опубликовано несколько версий сборки среды выполнения, приложение, содержащее сведения о внедренных типах, может работать с различными версиями без перекомпиляции. Пример см. в разделе Пошаговое руководство. внедрению типов из управляемых сборок.

Параметр **EmbedInteropTypes** особенно полезен при работе с COM-взаимодействием. COM-типы внедряются для того, чтобы приложению не требовалась основная сборка взаимодействия (PIA) на целевом компьютере. Параметр **EmbedInteropTypes** указывает компилятору внедрить сведения о COM-типах из указанной сборки взаимодействия в результирующий скомпилированный код. COM-тип определяется значением CLSID (GUID). Это позволяет запускать приложение на целевом компьютере, где установлены те же COM-типы с такими же значениями CLSID. В качестве примера можно привести приложения, автоматизирующие Microsoft Office. Поскольку в приложениях типа Office значение CLSID обычно не зависит от версии, ваше приложение сможет использовать COM-типы по ссылке до тех пора, пока на целевом компьютере установлена платформа .NET Framework 4 или более поздней версии, а приложение работает с методами, свойствами или событиями, включенными в эти COM-типы. Параметр **EmbedInteropTypes** внедряет только интерфейсы, структуры и делегаты. Внедрение COM-классов не поддерживается.

NOTE

Если в коде создается экземпляр внедренного COM-типа, его следует создавать, используя соответствующий интерфейс. При попытке создать экземпляр внедренного COM-типа с помощью компонентного класса возникнет ошибка.

Как и параметр **References** компилятора, параметр **EmbedInteropTypes** компилятора использует файл ответов Csc.rsp, который ссылается на часто используемые сборки .NET. Если вы не хотите, чтобы компилятор использовал файл Csc.rsp, примените параметр **NoConfig** компилятора.

```
// The following code causes an error if ISampleInterface is an embedded interop type.
ISampleInterface<SampleType> sample;
```

Типы с универсальным параметром, тип которого внедрен из сборки взаимодействия, нельзя использовать, если он относится к внешней сборке. Это ограничение не относится к интерфейсам. Например, рассмотрим интерфейс Range, который определен в сборке Microsoft.Office.Interop.Excel. Если библиотека содержит внедренные типы взаимодействия из сборки Microsoft.Office.Interop.Excel и предоставляет метод, возвращающий универсальный тип с параметром, типом которого является интерфейс Range, этот метод должен возвращать универсальный интерфейс, как показано в следующем примере кода.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Office.Interop.Excel;

public class Utility
{
    // The following code causes an error when called by a client assembly.
    public List<Range> GetRange1()
    {
        return null;
    }

    // The following code is valid for calls from a client assembly.
    public IList<Range> GetRange2()
    {
        return null;
    }
}
```

В следующем примере клиентский код может вызывать метод, возвращающий универсальный интерфейс IList без ошибок.

```
public class Client
{
    public void Main()
    {
        Utility util = new Utility();

        // The following code causes an error.
        List<Range> rangeList1 = util.GetRange1();

        // The following code is valid.
        List<Range> rangeList2 = (List<Range>)util.GetRange2();
    }
}
```

Параметры компилятора C# для информирования об ошибках и предупреждениях

18.12.2021 • 3 minutes to read

Следующие параметры управляют тем, как компилятор сообщает об ошибках и предупреждениях. Новый синтаксис MSBuild выделен **полужирным шрифтом**. Для старого синтаксиса *csc.exe* используется формат `code style`.

- **WarningLevel** / `-warn`: определение порога предупреждений.
- **TreatWarningsAsErrors** / `-warnaserror`: обработка всех предупреждений как ошибок.
- **WarningsAsErrors** / `-warnaserror`: обработка одного или нескольких предупреждений как ошибок.
- **WarningsNotAsErrors** / `-warnnotaserror`: обработка одного или нескольких предупреждений не как ошибок.
- **DisabledWarnings** / `-nowarn`: определение списка отключенных предупреждений.
- **CodeAnalysisRuleSet** / `-ruleset`: определение файла набора правил, который отключает определенную диагностику.
- **ErrorLog** / `-errorlog`: определение файла для регистрации всей диагностики компилятора и анализатора.
- **ReportAnalyzer** / `-reportanalyzer`: регистрация дополнительных сведений анализатора, включая сведения о времени выполнения.

WarningLevel

Параметр **WarningLevel** определяет порог предупреждений для отображения компилятором.

```
<WarningLevel>3</WarningLevel>
```

Значение элемента — это порог предупреждений, который должен отображаться для компиляции: при меньших значениях отображаются только самые серьезные предупреждения. Чем выше значение, тем больше предупреждений будет отображаться. Значение должно быть больше нуля или равно ему.

УРОВЕНЬ ПРЕДУПРЕЖДЕНИЙ	ЗНАЧЕНИЕ
0	Отключает вывод всех предупреждающих сообщений.
1	Отображает серьезные предупреждающие сообщения.
2	Отображает предупреждения уровня 1, а также некоторые менее серьезные предупреждения, в том числе связанные со скрытием членов класса.
3	Отображает предупреждения уровня 2, а также некоторые менее серьезные предупреждения, в том числе связанные с выражениями, которые всегда возвращают значение `true` или `false`.

УРОВЕНЬ ПРЕДУПРЕЖДЕНИЙ	ЗНАЧЕНИЕ
4 (по умолчанию)	Отображает все предупреждения уровня 3, а также информационные предупреждения.
5	Отображает предупреждения уровня 4, а также дополнительные предупреждения от компилятора, поставляемого в составе C# 9.0.
Больше 5	Любое значение больше 5 будет рассматриваться как 5. Чтобы гарантировать постоянное отображение всех предупреждений, если в компиляторе указаны новые пороги предупреждений, задайте произвольное большое значение (например, `9999`).

Чтобы получить сведения об ошибке или предупреждении, выполните поиск по соответствующему коду в указателе справочной системы. Дополнительные сведения о других способах получить информацию об ошибках и предупреждениях см. в разделе Ошибки компилятора C#. Используйте **TreatWarningsAsErrors**, чтобы обрабатывать все предупреждения как ошибки. Используйте **DisabledWarnings**, чтобы отключить определенные предупреждения.

TreatWarningsAsErrors

Параметр **TreatWarningsAsErrors** включает обработку всех предупреждений как ошибок. Вы также можете использовать **TreatWarningsAsErrors**, чтобы обрабатывать как ошибки только некоторые предупреждения. Включив **TreatWarningsAsErrors**, вы можете использовать **TreatWarningsAsErrors**, чтобы перечислить предупреждения, которые не должны обрабатываться как ошибки.

```
<TreatWarningsAsErrors>true</TreatWarningsAsErrors>
```

Все предупреждающие сообщения вместо этого будут выводиться как ошибки. Процесс сборки будет остановлен (выходные файлы не будут создаваться). По умолчанию **TreatWarningsAsErrors** не применяется, и это означает, что наличие предупреждений не препятствует созданию выходного файла. Если требуется обрабатывать как ошибки только конкретные предупреждения, укажите их номера через запятую. Набор всех предупреждений о допустимости значений NULL можно указать с помощью сокращения **Nullable**. Используйте **WarningLevel**, чтобы определить порог предупреждений, которые будет отображать компилятор. Используйте **DisabledWarnings**, чтобы отключить определенные предупреждения.

WarningsAsErrors и WarningsNotAsErrors

Параметры **WarningsAsErrors** и **WarningsNotAsErrors** переопределяют параметр **TreatWarningsAsErrors** для списка предупреждений.

Включение предупреждений 0219 и 0168 как ошибок:

```
<WarningsAsErrors>0219,0168</WarningsAsErrors>
```

Отключение этих же предупреждений как ошибок:

```
<WarningsNotAsErrors>0219,0168</WarningsNotAsErrors>
```

Вы можете использовать **WarningsAsErrors**, чтобы настроить набор предупреждений как ошибок.

Используйте **WarningsNotAsErrors**, чтобы настроить набор предупреждений, которые не должны обрабатываться как ошибки, если вы включили обработку всех предупреждений как ошибок.

DisabledWarnings

Параметр **DisabledWarnings** позволяет отключить отображение одного или нескольких предупреждений компилятором. Разделяйте предупреждения запятыми.

```
<DisabledWarnings>number1, number2</DisabledWarnings>
```

`number1`, `number2` — номера предупреждений, которые требуется отключить в компиляторе. Вы можете указать только числовую часть идентификатора предупреждения. Например, чтобы отключить *CS0028*, можно указать `<DisabledWarnings>28</DisabledWarnings>`. Компилятор просто пропустит номера предупреждений, переданные в **DisabledWarnings**, которые были действительными в предыдущих выпусках, но были удалены. Например, предупреждение *CS0679* было действительным в компиляторе Visual Studio .NET 2002, но было удалено в последующих версиях.

Следующие предупреждения нельзя отключить с помощью параметра **DisabledWarnings**:

- Предупреждение компилятора (уровень 1) CS2002
- Предупреждение компилятора (уровень 1) CS2023
- Предупреждение компилятора (уровень 1) CS2029

CodeAnalysisRuleSet

Укажите файл набора правил, который настраивает определенную диагностику.

```
<CodeAnalysisRuleSet>MyConfiguration.ruleset</CodeAnalysisRuleSet>
```

`MyConfiguration.ruleset` — это путь к файлу набора правил. Дополнительные сведения об использовании наборов правил см. в статье Использование набора правил для группирования правил анализа кода.

ErrorLog

Укажите файл для записи данных диагностики компилятора и анализатора в журнал.

```
<ErrorLog>compiler-diagnostics.sarif</ErrorLog>
```

Параметр **ErrorLog** указывает компилятору вывести журнал в формате обмена результатами статического анализа (SARIF). Журналы SARIF обычно считываются средствами, которые анализируют результаты диагностики компилятора и анализатора.

ReportAnalyzer

Включение в отчет дополнительных сведений об анализаторе, включая время выполнения.

```
<ReportAnalyzer>true</ReportAnalyzer>
```

Параметр **ReportAnalyzer** указывает компилятору создавать дополнительные сведения журнала MSBuild, в которых подробно описываются характеристики производительности анализаторов в сборке. Обычно он используется создателями анализаторов в ходе проверки анализатора.

Параметры компилятора C#, которые управляют созданием кода

18.12.2021 • 3 minutes to read

Следующие параметры управляют созданием кода в компиляторе. Новый синтаксис MSBuild выделен **полужирным шрифтом**. Для старого синтаксиса *csc.exe* используется формат `code style`.

- **DebugType** / `-debug`: выдача (или невыдача) отладочных сведений.
- **Optimize** / `-optimize`: включение оптимизаций.
- **Deterministic** / `-deterministic`: создание эквивалентных (побайтовое сравнение) выходных данных для одного источника входных данных.
- **ProduceOnlyReferenceAssembly** / `-refonly`: создание базовой сборки вместо полной сборки в качестве основного выходного файла.

DebugType

При использовании параметра **DebugType** компилятор создает отладочные сведения и помещает их в выходном файле или файлах. Отладочные сведения добавляются по умолчанию для конфигурации сборки *отладки*. По умолчанию этот параметр отключен для конфигурации сборки *выпуска*.

```
<DebugType>pdbonly</DebugType>
```

Для всех версий компилятора, начиная с C# 6.0, разницы между *pdbonly* и *full* нет. Выберите *pdbonly*. Чтобы изменить расположение файла с расширением *.pdb*, см. раздел **PdbFile**.

Допустимы следующие значения.

ЗНАЧЕНИЕ	ЗНАЧЕНИЕ
`full`	Выводить информацию об отладке в *PDB*-файл, используя стандартный формат для текущей платформы: **Windows**: PDB-файл Windows. **Linux/macOS**: переносимый PDB-файл.
`pdbonly`	Эквивалентно `full`. Дополнительные сведения см. в примечании ниже.
`portable`	Выводить информацию об отладке в PDB-файл, используя кроссплатформенный формат переносимого PDB-файла.
`embedded`	Выводить информацию об отладке в файл *DLL или EXE* (*PDB*-файл не создается), используя формат переносимого PDB-файла.

IMPORTANT

Следующая информация касается только компиляторов версии ниже C# 6.0. Значение этого элемента может быть либо `full`, либо `pdbonly`. Аргумент *full*, действующий при отсутствии *pdbonly*, позволяет присоединить отладчик к выполняющейся программе. Определение *pdbonly* позволяет выполнять отладку исходного кода при запуске программы в отладчике, но при этом ассемблер отображается только при подключении выполняющейся программы к отладчику. Используйте этот параметр для создания отладочных сборок. При использовании параметра *full* нужно учитывать некоторое влияние на скорость и размер оптимизированного кода JIT и незначительное влияние на качество кода с *full*. Для создания кода выпуска рекомендуется использовать *pdbonly* или не использовать PDB. Разница между *pdbonly* и *full* заключается в том, что компилятор с *full* создает DebuggableAttribute, чтобы сообщить JIT-компилятору о доступности отладочных сведений. Поэтому если при использовании *full* код содержит DebuggableAttribute со значением false, вы получите сообщение об ошибке. Дополнительные сведения о настройке производительности отладки для приложения см. в разделе Упрощение отладки образов.

Оптимизация

Параметр **Optimize** включает или отключает оптимизацию кода компилятором, чтобы сделать код более быстрым, коротким и эффективным. Параметр *Optimize* включен по умолчанию для конфигурации сборки *выпуска*. По умолчанию этот параметр отключен для конфигурации сборки *отладки*.

```
<Optimize>true</Optimize>
```

Параметр **Optimize** можно задать на странице свойств **сборки** для проекта в Visual Studio.

Кроме того, параметр **Optimize** включает оптимизацию кода во время выполнения в общеязыковой среде выполнения. По умолчанию оптимизация отключена. Чтобы включить оптимизацию, укажите **Optimize+** . При создании модуля для сборки используйте те же настройки **Optimize**, что и для сборки. Параметры **Optimize** и **Debug** можно комбинировать.

Детерминированное

Указывает компилятору на необходимость создания сборки, чьи побайтовые выходные данные идентичны в разных компиляциях, если входные данные идентичны.

```
<Deterministic>true</Deterministic>
```

По умолчанию выходные данные компилятора из заданного набора входных данных являются уникальными, так как компилятор добавляет метку времени и идентификатор MVID, который создается из случайных чисел. Вы можете использовать параметр `<Deterministic>` для создания *детерминированной сборки*, двоичное содержимое которой идентично в разных компиляциях при условии, что входные данные не изменяются. В такой сборке поля timestamp и MVID будут заменены значениями, полученными из хэша всех входных данных компиляции. Компилятор рассматривает следующие входные данные, которые влияют на детерминированность:

- Последовательность параметров командной строки.
- Содержимое RSP-файла ответов в компиляторе.
- Точная версия компилятора и его связанные сборки.
- Текущий путь к каталогу.
- Двоичное содержимое всех файлов, явным образом переданных компилятору прямо или косвенно, в том числе:
  - Исходные файлы

    - Связанные сборки
    - Связанные модули
    - Ресурсы
    - Файл ключа строгого имени
    - Файлы ответов @
    - Анализаторы
    - Наборы правил
    - Другие файлы, которые могут использоваться анализаторами
- Текущий язык и региональные параметры (для языка сообщений о диагностике и исключениях).
- Кодировка по умолчанию (или текущая кодовая страница), если кодировка не указана.
- Наличие, отсутствие и содержимое файлов на пути поиска компилятора (задается, например, с помощью `-lib` или `-recurse`).
- Платформа общеязыковой среды выполнения (CLR), на которой выполняется компилятор.
- Значение `%LIBPATH%`, которое может повлиять на загрузку зависимостей анализатора.

Детерминированную компиляцию можно использовать, чтобы определить, компилируются ли двоичные данные из надежного источника. Детерминированные выходные данные могут быть полезны, если источник является общедоступным. Также с их помощью можно определить, будут ли шаги сборки, зависящие от изменений в двоичном файле, использоваться в процессе сборки.

ProduceOnlyReferenceAssembly

Параметр **ProduceOnlyReferenceAssembly** указывает на то, что в качестве основных выходных данных должна быть выведена базовая сборка, а не сборка реализации. Параметр **ProduceOnlyReferenceAssembly** автоматически отключает вывод PDB, так как базовые сборки не могут выполняться.

```
<ProduceOnlyReferenceAssembly>true</ProduceOnlyReferenceAssembly>
```

Базовые сборки являются особым типом сборки, которая содержит только минимальный объем метаданных, необходимый для представления общедоступного API библиотеки. Такие сборки включают в себя объявления для всех элементов, которые важны при указании ссылки на сборку в средствах сборки, но исключают все реализации элементов, а также объявления закрытых элементов, не имеющих наблюдаемого влияния на их контракт API. Дополнительные сведения см. в статье Базовые сборки.

Параметры **ProduceOnlyReferenceAssembly** и **ProduceReferenceAssembly** являются взаимоисключающими.

Параметры компилятора C# для обеспечения безопасности

18.12.2021 • 4 minutes to read

Следующие параметры управляют параметрами безопасности компилятора. Новый синтаксис MSBuild выделен **полужирным шрифтом**. Для старого синтаксиса *csc.exe* используется формат `code style`.

- **PublicSign** / `-publicsign`: публичное подписывание сборки.
- **DelaySign** / `-delaysign`: отложить подпись для сборки, используя только открытую часть ключа строгого имени.
- **KeyFile** / `-keyfile`: указание файла ключа строгого имени.
- **KeyContainer** / `-keycontainer`: указание контейнера ключа строгого имени.
- **HighEntropyVA** / `-highentropyva`: включить технологию ASLR с высокой энтропией.

PublicSign

Этот параметр указывает компилятору на необходимость применения открытого ключа, но фактически не подписывает сборку. Кроме того, параметр **PublicSign** задает в сборке бит, тем самым сообщая среде выполнения, что файл подписан.

```
<PublicSign>true</PublicSign>
```

С параметром **PublicSign** необходимо использовать параметр KeyFile или KeyContainer. Параметры **KeyFile** и **KeyContainer** определяют открытый ключ. Параметры **PublicSign** и **DelaySign** — взаимоисключающие. При таком подписывании в сборку добавляется открытый ключ. Кроме того, в сборке устанавливается флаг "подписано". Такой подход иногда называют "фиктивным подписыванием" или "подписыванием OSS". Но фактически сборка не подписывается закрытым ключом. Разработчики используют публичное подписывание для проектов с открытым кодом. Им нужно создавать сборки, которые совместимы с выпущенными "полностью подписанными" сборками, но у них нет доступа к закрытому ключу, который использовался для подписывания сборок. Так как потребителям редко нужно проверять, полностью ли подписана сборка, создаваемые сообществами сборки можно использовать практические во всех случаях, в которых может использоваться полностью подписанная сборка.

DelaySign

Этот параметр указывает компилятору зарезервировать пространство в выходном файле, чтобы впоследствии добавить в него цифровую подпись.

```
<DelaySign>true</DelaySign>
```

Если требуется полностью подписанная сборка, используйте параметр **DelaySign**- . Если нужно лишь поместить в сборку открытый ключ, используйте параметр **DelaySign**. Параметр **DelaySign** не действует, если не использовать его с KeyFile или KeyContainer. Параметры KeyContainer и PublicSign являются взаимоисключающими. При запросе полностью подписанной сборки компилятор хэширует файл, содержащий манифест (метаданные сборки), и подписывает хэш закрытым ключом. Эта операция предназначена для создания цифровой подписи, которая хранится в файле, содержащем манифест. Если применяется отложенное подписывание сборки, компилятор не вычисляет и не сохраняет подпись, но он

резервирует пространство в файле, чтобы подпись можно было добавить позже.

Параметр **DelaySign** позволяет поместить сборку в глобальный кэш при тестировании. После тестирования можно полностью подписать сборку, поместив в нее закрытый ключ с помощью компоновщика сборок. Дополнительные сведения см. в разделах Создание и использование сборок со строгими именами и Отложенная подпись сборки.

KeyFile

Задает имя файла, содержащего криптографический ключ.

```
<KeyFile>filename</KeyFile>
```

`file` — это имя файла, содержащего ключ строгого имени. При использовании этого параметра компилятор вставляет открытый ключ из указанного файла в манифест сборки и затем подписывает окончательную сборку закрытым ключом. Чтобы создать файл ключа, в командной строке введите `sn -k file`. При компиляции с параметром **-target:module** имя файла ключа сохраняется в модуле и включается в сборку, создаваемую при компиляции с параметром **AddModules**. Также можно передать сведения о шифровании компилятору с помощью параметра **Keycontainer**. Если требуется использовать частично подписанную сборку, укажите параметр **DelaySign**. Если для одной процедуры компиляции одновременно заданы параметры **KeyFile** и **KeyContainer**, сначала будет предпринята попытка использовать контейнер ключей. В случае успеха сборка подписывается данными контейнера ключей. Если компилятору не удалось обнаружить контейнер ключей, будет предпринята попытка использовать файл, заданный параметром **KeyFile**. В случае успеха сборка подписывается данными из файла ключей, и эти данные о ключах будут помещены в контейнер ключей. Таким образом при следующей компиляции контейнер ключей будет действителен. Файл ключей может содержать только открытый ключ. Дополнительные сведения см. в разделах Создание и использование сборок со строгими именами и Отложенная подпись сборки.

KeyContainer

Задает имя контейнера криптографического ключа.

```
<KeyContainer>container</KeyContainer>
```

`container` — это имя контейнера ключа строгого имени. Если используется параметр **KeyContainer**, компилятор создает совместно используемый компонент. Компилятор вставляет открытый ключ из указанного контейнера в манифест сборки, после чего подписывает финальную сборку закрытым ключом. Чтобы создать файл ключа, в командной строке введите `sn -k file`. Команда `sn -i` устанавливает пару ключей в контейнер. Этот параметр не поддерживается, если компилятор работает в CoreCLR. Чтобы подписать сборку при компиляции в CoreCLR, используйте параметр **KeyFile**. При компиляции с параметром **TargetType** имя файла ключа сохраняется в модуле и включается в сборку при компиляции этого модуля с параметром **AddModules**. Этот параметр можно также указать в исходном коде любого модуля MSIL в качестве настраиваемого атрибута (System.Reflection.AssemblyKeyNameAttribute). Также можно передать сведения о шифровании компилятору с помощью параметра **KeyFile**. Если нужно добавить в манифест сборки открытый ключ, но при этом отложить подпись сборки до завершения ее тестирования, используйте параметр **DelaySign**. Дополнительные сведения см. в разделах Создание и использование сборок со строгими именами и Отложенная подпись сборки.

HighEntropyVA

Параметр компилятора **HighEntropyVA** сообщает ядру Windows, поддерживает ли указанный исполняемый файл технологию Address Space Layout Randomization (ASLR) с высокой энтропией.

```
<HighEntropyVA>true</HighEntropyVA>
```

Этот параметр указывает, что 64-битный исполняемый файл или исполняемый файл, отмеченный параметром компилятора PlatformTarget, поддерживает виртуальный диапазон адресов с высокой энтропией. Этот параметр отключен по умолчанию. Чтобы включить его, используйте параметр **HighEntropyVA**.

Параметр **HighEntropyVA** позволяет совместимым версиям ядра Windows использовать более высокие степени энтропии во время рандомизации размещения диапазона адресов в рамках ASLR. Использование более высоких степеней энтропии означает, что можно выделить больше адресов таким областям памяти, как стеки и кучи. Из-за этого сложнее подобрать расположение определенной области памяти. Если указан параметр компилятора **HighEntropyVA**, целевой исполняемый файл и все модули, от которых он зависит, должны иметь возможность обработать значения указателя, размер которых превышает 4 гигабайта (ГБ), если они выполняются как 64-разрядный процесс.

Параметры компилятора C#, указывающие ресурсы

18.12.2021 • 4 minutes to read

Следующие параметры определяют способ создания и импорта ресурсов Win32 компилятором C#. Новый синтаксис MSBuild выделен **полужирным шрифтом**. Для старого синтаксиса *csc.exe* используется формат `code style`.

- **Win32Resource** / `-win32res`: указание файла ресурсов Win32 (RES-файл).
- **Win32Icon** / `-win32icon`: создание ссылки на метаданные из указанного файла или файлов сборки.
- **Win32Manifest** / `-win32manifest`: указание файла манифеста Win32 (XML-файл).
- **NoWin32Manifest** / `-nowin32manifest`: указание не включать манифест Win32 по умолчанию.
- **Resources** / `-resource`: встраивание указанного ресурса (короткая форма: /res).
- **LinkResources** / `-linkresources`: связывание указанного ресурса с этой сборкой.

Win32Resource

Параметр **Win32Resource** вставляет ресурс Win32 в выходной файл.

```
<Win32Resource>filename</Win32Resource>
```

`filename` — это файл ресурсов, который требуется добавить в выходной файл. Ресурс Win32 может содержать сведения о версии или точечный рисунок (значок) для упрощения идентификации приложения в проводнике. Если этот параметр не задан, компилятор будет создавать сведения о версии на основе версии сборки.

Win32Icon

Параметр **Win32Icon** вставляет в выходной файл ICO-файл, который придает выходному файлу необходимый вид в проводнике.

```
<Win32Icon>filename</Win32Icon>
```

`filename` — это файл с расширением *.ico*, который требуется добавить в выходной файл. Файл с расширением *.ico* можно создать с помощью компилятора ресурсов. Компилятор ресурсов вызывается при компиляции программы Visual C++. При этом файл с расширением *.ico* создается из файла *.rc*.

Win32Manifest

Параметр **Win32Manifest** позволяет указать пользовательский файл манифеста приложения win32manifest для внедрения в переносимый исполняемый файл проекта (PE-файл).

```
<Win32Manifest>filename</Win32Manifest>
```

`filename` — это имя и расположение пользовательского файла манифеста. По умолчанию компилятор C# внедряет манифест приложения, определяющий запрошенный уровень выполнения asInvoker. Он

создает манифест в той же папке, в которой создается исполняемый файл. Если необходимо предоставить пользовательский манифест, например, чтобы задать уровень выполнения highestAvailable or requireAdministrator, используйте этот параметр, чтобы указать имя файла.

> **NOTE**
> Этот параметр и параметр **Win32Resources** являются взаимоисключающими. При попытке использовать оба параметра в одной командной строке возникнет ошибка построения.

Для приложения без манифеста, определяющего запрошенный уровень выполнения, требуется виртуализация файлов и реестра с помощью функции контроля учетных записей в Windows. Дополнительные сведения см. в разделе Контроль учетных записей.

Приложение требует виртуализации в любом из следующих случаев:

- Вы можете использовать параметр **NoWin32Manifest** и не предоставлять манифест на более позднем этапе сборки или в файле ресурсов Windows (*.res*) с помощью параметра **Win32Resource**.
- Вы можете предоставлять пользовательский манифест, который не определяет запрошенный уровень выполнения.

Visual Studio создает файл по умолчанию с расширением *.manifest* и сохраняет его в каталогах отладки и выпуска вместе с исполняемым файлом. Пользовательский манифест можно добавить, создав его в любом текстовом редакторе и добавив полученный файл в проект. Вы также можете щелкнуть значок **проекта** в **Обозревателе решений** и нажать кнопку **Добавить новый элемент**, а затем **Файл манифеста приложения**. Добавленный новый или существующий файл манифеста появится в раскрывающемся списке **Манифест**. Дополнительные сведения см. в разделе Страница "Приложение" в конструкторе проектов (C#).

Манифест приложения можно предоставить во время пользовательского этапа после сборки или в составе файла ресурсов Win32 с помощью параметра **NoWin32Manifest**. Этот же параметр можно использовать, если вы хотите, чтобы ваше приложение требовало виртуализации файлов или реестров в Windows Vista.

NoWin32Manifest

С помощью параметра **NoWin32Manifest** можно указать компилятору не внедрять манифест приложения в исполняемый файл.

```
<NoWin32Manifest />
```

При использовании этого параметра приложение будет подлежать виртуализации в Windows Vista, если манифест приложения не будет предоставлен в файле ресурсов Win32 или на более поздних этапах сборки.

В Visual Studio этот параметр можно задать на странице **Свойство приложения**, выбрав в раскрывающемся списке **Манифест** пункт **Создать приложение без манифеста**. Дополнительные сведения см. в разделе Страница "Приложение" в конструкторе проектов (C#).

Ресурсы

Внедряет указанный ресурс в выходной файл.

```
<Resources Include=filename>
  <LogicalName>identifier</LogicalName>
  <Access>accessibility-modifier</Access>
</Resources>
```

`filename` — это файл ресурсов .NET, который требуется внедрить в выходной файл. `identifier` (необязательно) — это логическое имя ресурса, используемое для его загрузки. По умолчанию используется имя файла. `accessibility-modifier` (необязательно) — это доступность ресурса: public (открытый) или private (закрытый). Значение по умолчанию: public. По умолчанию ресурсы в сборке открыты, если они создавались с помощью компилятора C#. Чтобы сделать ресурс закрытым, укажите параметр `private` в качестве модификатора доступа. Уровни доступности, отличные от `public` или `private`, не допускаются. Если `filename` является файлом ресурсов .NET, созданным, например, с помощью Resgen.exe или в среде разработки, то к нему можно обращаться с помощью элементов в пространстве имен System.Resources. Для получения дополнительной информации см. System.Resources.ResourceManager. Чтобы получить доступ ко всем остальным ресурсам во время выполнения, используйте методы `GetManifestResource` в классе Assembly. Порядок расположения ресурсов в выходном файле будет определяться порядком, указанным в файле проекта.

LinkResources

Создает в выходном файле ссылку на ресурс .NET. Файл ресурсов не добавляется в выходной файл. Параметр **LinkResources** отличается от параметра **Resource**, который внедряет файл ресурсов в выходной файл.

```
<LinkResources Include=filename>
  <LogicalName>identifier</LogicalName>
  <Access>accessibility-modifier</Access>
</LinkResources>
```

`filename` — это файл ресурсов .NET, ссылку на который необходимо создать из сборки. `identifier` (необязательно) — это логическое имя ресурса, используемое для его загрузки. По умолчанию используется имя файла. `accessibility-modifier` (необязательно) — это доступность ресурса: public (открытый) или private (закрытый). Значение по умолчанию: public. По умолчанию связанные ресурсы в сборке открыты, если они создавались с помощью компилятора C#. Чтобы сделать ресурс закрытым, укажите параметр `private` в качестве модификатора доступа. Модификаторы, отличные от `public` или `private`, не допускаются. Если `filename` является файлом ресурсов .NET, созданным, например, с помощью Resgen.exe или в среде разработки, то к нему можно обращаться с помощью элементов в пространстве имен System.Resources. Для получения дополнительной информации см. System.Resources.ResourceManager. Чтобы получить доступ ко всем остальным ресурсам во время выполнения, используйте методы `GetManifestResource` в классе Assembly. Файл, указанный в параметре `filename`, может иметь любой формат. Например, может потребоваться сделать имеющуюся на компьютере библиотеку DLL частью сборки, поэтому ее можно разместить в глобальном кэше сборок и обеспечить к ней доступ из управляемого кода сборки. Это действие можно также выполнить в компоновщике сборок. Дополнительные сведения см. в разделах Al.exe (компоновщик сборок) и Работа со сборками и глобальным кэшем сборок.

Прочие параметры компилятора C#

18.12.2021 • 2 minutes to read

Следующие параметры управляют прочей функциональностью компилятора. Новый синтаксис MSBuild выделен **полужирным шрифтом**. Для старого синтаксиса *csc.exe* используется формат `code style`.

- **ResponseFiles** / `-@`: считывание файла ответов с дополнительными параметрами.
- **NoLogo** / `-nologo`: запрещает отображение сообщения компилятора об авторских правах.
- **NoConfig** / `-noconfig`: запрещает автоматическое включение файла *CSC.RSP*.

ResponseFiles

С помощью параметра **ResponseFiles** можно указать файл, содержащий параметры компилятора и список файлов исходного кода, которые требуется компилировать.

```
<ResponseFiles>response_file</ResponseFiles>
```

`response_file` указывает файл, содержащий список параметров компилятора и файлов исходного кода, которые требуется компилировать. Параметры компилятора и файлы исходного кода будут обрабатываться компилятором таким образом, как если бы они были указаны в командной строке. Чтобы задать несколько файлов ответов для компиляции, используйте соответствующее число параметров файла ответов. В одной строке файла ответов может содержаться несколько параметров компилятора и файлов исходного кода. Спецификация отдельного параметра компилятора должна размещаться на одной строке и не может разбиваться на несколько строк. В файл ответов можно добавлять комментарии, которые должны начинаться с символа #. Указание параметров компилятора в файле ответов аналогично выполнению соответствующих команд из командной строки. Компилятор обрабатывает параметры команд в том порядке, в котором они считываются. Аргументы командной строки могут переопределять параметры, заданные ранее в файле ответов. Аналогичным образом, параметры в файле ответов будут переопределять параметры, ранее заданные в командной строке или в других файлах ответов. В C# представлен файл csc.rsp, который находится в одном каталоге с файлом csc.exe. Дополнительные сведения о формате файла ответов см. в разделе о параметре NoConfig. Этот параметр компилятора нельзя задать в среде разработки Visual Studio или изменить программными средствами. Ниже приведено несколько строк из образца файла ответов:

```
# build the first output file
-target:exe -out:MyExe.exe source1.cs source2.cs
```

NoLogo

Параметр **NoLogo** отключает отображение приветствия при запуске компилятора и информационных сообщений во время компиляции.

```
<NoLogo>true</NoLogo>
```

NoConfig

Параметр **NoConfig** указывает компилятору не использовать файл *csc.rsp* при компиляции.

```
<NoConfig>true</NoConfig>
```

Файл *csc.rsp* содержит ссылки на все сборки, поставляемые вместе с .NET Framework. Фактические ссылки, которые включает среда разработки Visual Studio .NET, зависят от типа проекта. Вы можете изменить файл *csc.rsp* и указать дополнительные параметры компилятора, которые нужно включать при каждой компиляции. Если не требуется, чтобы компилятор искал и использовал параметры в файле *csc.rsp*, укажите параметр **NoConfig**. Этот параметр компилятора недоступен в Visual Studio и не может быть изменен программным способом.

Расширенные параметры компилятора C#

18.12.2021 • 8 minutes to read

Следующие параметры поддерживают сложные сценарии. Новый синтаксис MSBuild выделен **полужирным шрифтом**. Для старого синтаксиса `csc.exe` используется `code style`.

- **MainEntryPoint**, **StartupObject** / `-main`: указание типа, который содержит точку входа.
- **PdbFile** / `-pdb`: указание имени файла с данными отладки.
- **PathMap** / `-pathmap`: указание сопоставления для вывода компилятором имен исходных путей.
- **ApplicationConfiguration** / `-appconfig`: указание файла конфигурации приложения, который содержит параметры привязки сборки.
- **AdditionalLibPaths** / `-lib`: указание дополнительных каталогов для поиска ссылок.
- **GenerateFullPaths** / `-fullpath`: компилятор будет создавать абсолютные пути.
- **PreferredUILang** / `-preferreduilang`: указание имени предпочтительного языка для вывода данных.
- **BaseAddress** / `-baseaddress`: указание базового адреса библиотеки для сборки.
- **ChecksumAlgorithm** / `-checksumalgorithm`: указание алгоритма для расчета контрольной суммы файла источника, хранящегося в PDB.
- **CodePage** / `-codepage`: указание кодовой страницы, используемой при открытии исходных файлов.
- **Utf8Output** / `-utf8output`: сообщения компилятора будут выводиться в кодировке UTF-8.
- **FileAlignment** / `-filealign`: указание выравнивания для разделов выходного файла.
- **ErrorEndLocation** / `-errorendlocation`: выходные строка и столбец конечного расположения каждой ошибки.
- **NoStandardLib** / `-nostdlib`: не указывать ссылку на стандартную библиотеку *mscorlib.dll*.
- **SubsystemVersion** / `-subsystemversion`: указание версии подсистемы для этой сборки.
- **ModuleAssemblyName** / `-moduleassemblyname`: имя сборки, частью которой будет этот модуль.

MainEntryPoint или StartupObject

Этот параметр определяет класс, который содержит точку входа в программу, если метод `Main` содержит сразу несколько классов.

```
<StartupObject>MyNamespace.Program</StartupObject>
```

или

```
<MainEntryPoint>MyNamespace.Program</MainEntryPoint>
```

Где `Program` — это тип, содержащий метод `Main`. Указанное имя класса должно быть полным; оно должно включать полное пространство имен, содержащее ключевое слово class, за которым следует имя класса. Например, если метод `Main` находится в классе `Program` в пространстве имен `MyApplication.Core`, необходимо указать параметр компилятора `-main:MyApplication.Core.Program`. Если ваша компиляция включает более одного типа с методом `Main`, вы можете указать, какой тип содержит метод `Main`.

> **NOTE**
> Этот параметр нельзя использовать для проекта, который включает инструкции верхнего уровня, даже если этот проект содержит один или несколько методов `Main`.

PdbFile

Параметр компилятора **PdbFile** задает имя и расположение файла отладочных символов. Значение `filename` указывает на имя и расположение файла отладочных символов.

```
<PdbFile>filename</PdbFile>
```

Если указан **DebugType**, компилятор создаст *PDB*-файл в том же каталоге, в котором он создаст выходной файл (EXE или DLL). *PDB*-файл имеет такое же базовое имя файла, что и выходной файл. С помощью параметра **PdbFile** можно задать имя и расположение PDB-файла, отличающееся от используемых по умолчанию. Этот параметр компилятора нельзя задать в среде разработки Visual Studio или изменить программными средствами.

PathMap

Параметр компилятора **PathMap** определяет способ сопоставления компилятором физических путей и выходных имен исходных путей. Этот параметр сопоставляет каждый физический путь на компьютере, где выполняется компилятор, с соответствующим путем, который должен быть записан в выходные файлы. В следующем примере `path1` — это полный путь к исходным файлам в текущей среде, а `sourcePath1` — исходный путь, подставляемый вместо `path1` во всех выходных файлах. Чтобы указать несколько сопоставленных исходных путей, разделите их точкой с запятой.

```
<PathMap>path1=sourcePath1;path2=sourcePath2</PathMap>
```

Компилятор записывает исходный путь в выходные данные по следующим причинам:

1. Исходный путь подставляется вместо аргумента, когда CallerFilePathAttribute применяется как необязательный параметр.
2. Исходный путь внедряется как PDB-файл.
3. Путь к PDB-файлу внедряется в PE-файл (переносимый исполняемый файл).

ApplicationConfiguration

Параметр компилятора **ApplicationConfiguration** позволяет приложению C# задать расположение файла конфигурации приложения сборки (app.config) в среде CLR во время привязки сборки.

```
<ApplicationConfiguration>file</ApplicationConfiguration>
```

Где `file` — это файл конфигурации приложения, содержащий параметры привязки сборки. Один из случаев использования параметра **ApplicationConfiguration** — сложные сценарии, когда в сборке одновременно используются ссылки на версию .NET Framework и на версию .NET Framework для Silverlight определенной базовой сборки. Например, для конструктора XAML, написанного в Windows Presentation Foundation (WPF), может потребоваться ссылаться на оба рабочих стола WPF, для пользовательского интерфейса конструктора и для подмножества WPF, поставляемого с Silverlight. Одна и та же сборка конструктора имеет доступ к обеим сборкам. По умолчанию отдельные ссылки вызывают ошибку

компиляции, так как привязка сборки видит две эквивалентные сборки. Параметр компилятора **ApplicationConfiguration** позволяет указать расположение файла app.config, который отключает поведение по умолчанию с помощью тега `<supportPortability>`, как показано в следующем примере.

```
<supportPortability PKT="7cec85d7bea7798e" enable="false"/>
```

Компилятор передает расположение файла в логику с привязкой сборки среды CLR.

> **NOTE**
> Чтобы использовать файл app.config, уже заданный в проекте, добавьте тег свойства `<UseAppConfigForCompiler>` в *CSPROJ*-файл и задайте для него значение `true`. Чтобы указать другой файл app.config, добавьте тег свойства `<AppConfigForCompiler>` и задайте в качестве его значения расположение требуемого файла.

В следующем примере показан файл app.config, позволяющий приложению иметь ссылки на реализации .NET Framework и .NET Framework для реализации Silverlight любой сборки .NET Framework, существующей в обеих реализациях. Параметр компилятора **ApplicationConfiguration** позволяет указать расположение этого файла app.config.

```
<configuration>
  <runtime>
    <assemblyBinding>
      <supportPortability PKT="7cec85d7bea7798e" enable="false"/>
      <supportPortability PKT="31bf3856ad364e35" enable="false"/>
    </assemblyBinding>
  </runtime>
</configuration>
```

AdditionalLibPaths

Параметр **AdditionalLibPaths** позволяет задать расположение сборок, указанных с помощью параметра References.

```
<AdditionalLibPaths>dir1[,dir2]</AdditionalLibPaths>
```

Где `dir1` — это каталог, в котором компилятор должен искать базовую сборку, если она отсутствует в текущем рабочем каталоге (из которого был вызван компилятор) и системном каталоге среды CLR. `dir2` — это один или несколько дополнительных каталогов для поиска связанных сборок. Имена каталогов разделяются запятыми без пробелов. Компилятор выполняет поиск ссылок на сборки, для которых не указано полное имя, в следующем порядке:

1. Текущая рабочая папка.
2. Системный каталог среды CLR.
3. Каталоги, указанные с помощью параметра **AdditionalLibPaths**.
4. Каталоги, указанные переменной среды LIB.

Для указания ссылки на сборку используется параметр **Reference**. **AdditionalLibPaths** является аддитивным, то есть каждое следующее указание этого параметра присоединяется к предыдущим значениям. Так как путь к зависимой сборке не указывается в манифесте сборки, приложение найдет используемую сборку в глобальном кэше сборок. Компилятор, ссылающийся на сборку, не подразумевает, что среда CLR может найти и загрузить сборку во время выполнения. Сведения о том, как

среда выполнения выполняет поиск связанных сборок, см. в разделе Обнаружение сборок в среде выполнения.

GenerateFullPaths

Параметр **GenerateFullPaths** указывает компилятору на необходимость задавать полный путь к файлу при выводе списка ошибок и предупреждений компиляции.

```
<GenerateFullPaths>true</GenerateFullPaths>
```

По умолчанию для ошибок и предупреждений, возникающих в ходе компиляции, указывается имя файла, в котором они были обнаружены. Параметр **GenerateFullPaths** указывает компилятору на необходимость задать полный путь к файлу. Этот параметр компилятора недоступен в Visual Studio и не может быть изменен программным способом.

PreferredUILang

С помощью параметра компилятора **PreferredUILang** можно указать язык, на котором компилятор C# отображает выходные данные, например сообщения об ошибках.

```
<PreferredUILang>language</PreferredUILang>
```

Где `language` — это название языка, который будет использоваться для вывода компилятора. Параметр компилятора **PreferredUILang** можно использовать, чтобы указать язык, который компилятор C# должен использовать для сообщений об ошибках и других данных вывода командной строки. Если необходимый языковой пакет не установлен, вместо него используются языковые настройки операционной системы.

BaseAddress

Параметр **BaseAddress** позволяет указать предпочтительный базовый адрес для загрузки библиотеки DLL. Дополнительные сведения о случаях использования этого параметра см. в блоге Ларри Остермана (Larry Osterman).

```
<BaseAddress>address</BaseAddress>
```

Где `address` — это базовый адрес для библиотеки DLL. Этот адрес можно задать в десятичном, шестнадцатеричном или восьмеричном формате. Базовый адрес по умолчанию для библиотеки DLL задается в среде выполнения .NET. Младшее слово этого адреса будет округляться, например значение `0x11110001` округляется до `0x11110000`. Чтобы завершить процесс подписи для библиотеки DLL, используйте файл SN. EXE с параметром -R.

ChecksumAlgorithm

Этот параметр управляет алгоритмом контрольной суммы, который используется для кодирования исходных файлов в формат PDB.

```
<ChecksumAlgorithm>algorithm</ChecksumAlgorithm>
```

Параметр `algorithm` должен иметь значение `SHA1` (по умолчанию) или `SHA256`.

CodePage

Этот параметр позволяет указать кодовую страницу, используемую во время компиляции, если требуемая страница не является текущей кодовой страницей системы по умолчанию.

```
<CodePage>id</CodePage>
```

Где `id` — это идентификатор кодовой страницы, используемой для всех файлов исходного кода при компиляции. Во-первых, компилятор будет пытаться интерпретировать все исходные файлы в кодировке UTF-8. Если кодировка файлов исходного кода отличается от UTF-8 и использует символы, отличные от 7-разрядных символов ASCII, используйте параметр **CodePage**, чтобы указать нужную кодовую страницу. Параметр **CodePage** применяется ко всем файлам исходного кода, включенным в компиляцию. Сведения о том, как найти кодовые страницы, которые поддерживаются в системе, см. на странице, посвященной функции GetCPInfo.

Utf8Output

Параметр **Utf8Output** отображает выходные данные компилятора в кодировке UTF-8.

```
<Utf8Output>true</Utf8Output>
```

В некоторых конфигурациях для различных языков выходные данные компилятора некорректно отображаются в консоли. Используйте **Utf8Output** и перенаправьте выходные данные компилятора в файл.

FileAlignment

Параметр **FileAlignment** позволяет указать размер разделов в выходном файле. Допустимые значения: 512, 1024, 2048, 4096 и 8192. Эти значения указаны в байтах.

```
<FileAlignment>number</FileAlignment>
```

Параметр **FileAlignment** задается на странице **Дополнительно** в свойствах **сборки** проекта в Visual Studio. Каждый раздел выравнивается по границе, кратной значению **FileAlignment**. Фиксированный размер по умолчанию не предусмотрен. Если значение **FileAlignment** не указано, среда CLR выбирает значение по умолчанию во время компиляции. Указанный размер раздела влияет на размер выходного файла. Изменение размера раздела может применяться для программ, выполняющихся на небольших устройствах. Используйте DUMPBIN для просмотра информации о разделах выходного файла.

ErrorEndLocation

Указывает компилятору выводить строку и столбец конечного расположения каждой ошибки.

```
<ErrorEndLocation>true</ErrorEndLocation>
```

По умолчанию компилятор записывает начальное расположение в исходном коде для всех ошибок и предупреждений. Если этот параметр имеет значение True, компилятор записывает как начальное, так и конечное расположения для каждой ошибки и предупреждения.

NoStandardLib

Параметр **NoStandardLib** запрещает импорт библиотеки mscorlib.dll, которая определяет все пространство имен System.

```
<NoStandardLib>true</NoStandardLib>
```

Используйте этот параметр, если вы хотите определить или создать собственное пространство имен System и объекты. Если вы не укажете параметр **NoStandardLib**, библиотека mscorlib.dll будет импортирована в вашу программу (как и при указании `<NoStandardLib>false</NoStandardLib>`).

SubsystemVersion

Задает минимальную версию подсистемы, в которой может выполняться исполняемый файл. Чаще всего этот параметр предоставляет исполняемому файлу возможность использовать функции безопасности, недоступные в прежних версиях Windows.

> **NOTE**
> Чтобы указать саму подсистему, используйте параметр компилятора **TargetType**.

```
<SubsystemVersion>major.minor</SubsystemVersion>
```

`major.minor` задает минимальную требуемую версию подсистемы (через точку записываются основная и дополнительная версии). Например, можно указать, что приложение не может работать в операционной системе старше Windows 7. Установите для этого параметра значение 6.01, как описано в таблице ниже в этой статье. Укажите значения для параметров `major` и `minor` в виде целых чисел. Нули в начале версии `minor` не изменяют версию, нули в конце — изменяют. Например, 6.1 и 6.01 — одна версия, а 6.10 — другая. Рекомендуется указывать дополнительный номер версии двумя цифрами, чтобы избежать путаницы.

В следующей таблице перечислены распространенные версии подсистем Windows.

ВЕРСИЯ WINDOWS	ВЕРСИЯ ПОДСИСТЕМЫ
Windows Server 2003	5.02
Windows Vista	6.00
Windows 7	6.01
Windows Server 2008	6.01
Windows 8	6.02

Значение по умолчанию параметра компилятора **SubsystemVersion** зависит от условий в следующем списке:

- Значение по умолчанию — 6.02, если задан любой параметр компилятора из следующего списка.
  - /target:appcontainerexe
  - /target:winmdobj
  - -platform:arm
- Значение по умолчанию — 6.00, если используется средство MSBuild, приложение предназначено для

.NET Framework 4.5 и не установлены параметры компилятора, определенные ранее в этом списке.

- Если ни одно из предыдущих условий не верно, значение по умолчанию — 4.00.

ModuleAssemblyName

Указывает имя сборки, к отдельным типам которой может обращаться *.netmodule*.

```
<ModuleAssemblyName>assembly_name</ModuleAssemblyName>
```

Параметр **ModuleAssemblyName** нужно использовать при сборке *.netmodule* и выполнении следующих условий:

- Для *.netmodule* требуется доступ к неоткрытым типам в существующей сборке.
- Известно имя сборки, в которой будет создан .netmodule.
- Существующая сборка предоставила дружественной сборке доступ к сборке, в которую будет встроен *.netmodule*.

Дополнительные сведения о сборке .netmodule см. в разделе о параметре **TargetType** и его значении **module**. Дополнительные сведения см. в разделе Дружественные сборки.

Комментарии XML-документации

18.12.2021 • 9 minutes to read

В исходных файлах C# могут находиться структурированные комментарии, используемые для создания документации по API для типов, определенных в этих файлах. Компилятор C# создает файл *XML*, содержащий структурированные данные, представляющие комментарии и сигнатуры API. Другие средства могут обрабатывать эти выходные данные в формате XML и создавать удобочитаемую документацию, например, в виде веб-страниц или PDF-файлов.

Благодаря этому процессу добавление документации по API в код предоставляет множество преимуществ:

- Компилятор C# объединяет структуру кода C# и текст комментариев в XML-документ.
- Компилятор C# проверяет, соответствуют ли комментарии сигнатурам API для соответствующих тегов.
- Средства, обрабатывающие XML-файлы документации, могут определять XML-элементы и атрибуты, характерные для этих средств.

Такие средства как Visual Studio предоставляют IntelliSense для многих распространенных XML-элементов, используемых в комментариях к документации.

В этой статье рассматриваются следующие темы:

- Комментарии к документации и создание XML-файла
- Теги, проверяемые компилятором C# и Visual Studio
- Формат созданного XML-файла

Создание выходных данных XML-документации

Вы создаете документацию для кода, указывая ее в специальных полях комментариев, обозначаемых тройными косыми чертами. Поля комментария включают XML-элементы, которые описывают блок кода, следующий за комментариями. Пример:

```
/// <summary>
///  This class performs an important function.
/// </summary>
public class MyClass {}
```

Вы задаете параметр **DocumentationFile** или **DocumentationFile**, а компилятор находит все поля комментариев с XML-тегами в исходном коде и создает XML-файл документации на основе этих комментариев. Если этот параметр включен, компилятор создаст предупреждение CS1591 для любого открытого видимого члена, объявленного в проекте без комментариев XML-документации.

Формат комментариев XML

Для создания комментариев XML-документации необходимо использовать разделители, указывающие начало и конец комментария. С тегами в XML-документации можно использовать следующие разделители:

- Однострочный разделитель `///`: в примерах документации и шаблонах проектов C# используется эта форма. Если после разделителя указан пробел, он не включается в выходные данные XML.

> **NOTE**
> Visual Studio автоматически вставляет теги `<summary>` и `</summary>` и устанавливает курсор между ними после ввода разделителя `///` в редакторе кода. Вы можете включить или отключить эту функцию в диалоговом окне "Параметры".

- Многострочные разделители `/** */`: разделители `/** */` имеют следующие правила форматирования:
  - Строка, содержащая разделитель `/**`, не обрабатывается как комментарий, если оставшаяся ее часть представляет собой пробелы. Если первый знак после разделителя `/**` является пробелом, то этот пробел игнорируется, а оставшаяся часть строки обрабатывается. В противном случае весь текст, расположенный в строке после разделителя `/**`, обрабатывается как часть комментария.
  - Строка, содержащая разделитель `*/`, игнорируется, если перед разделителем `*/` стоят только пробелы. В противном случае текст, расположенный в строке до разделителя `*/`, обрабатывается как часть комментария.
  - В начале каждой из строк, расположенных после строки, которая начинается с разделителя `/**`, компилятор выполняет поиск общего шаблона. Шаблон может включать необязательный пробел и знак звездочки (`*`), после которого следуют необязательные пробелы. Если компилятор находит общий шаблон в начале каждой строки, которая не начинается с разделителя `/**` или не заканчивается разделителем `*/`, он игнорирует этот шаблон для каждой строки.
  - В следующем комментарии обрабатывается только строка, которая начинается с `<summary>`. Формат с тремя тегами позволяет создать точно такие же комментарии.

    ```
    /** <summary>text</summary> */

    /**
    <summary>text</summary>
    */

    /**
    * <summary>text</summary>
    */
    ```

  - Компилятор обнаруживает в начале второй и третьей строки общий шаблон " * ". Шаблон не включается в выходные данные.

    ```
    /**
    * <summary>
    * text </summary>*/
    ```

  - Компилятор не обнаруживает общий шаблон в следующем комментарии, так как второй знак в третьей строке не является звездочкой. Весь текст во второй и третьей строках обрабатывается как часть комментария.

    ```
    /**
    * <summary>
       text </summary>
    */
    ```

- Компилятор не обнаруживает шаблон в следующем комментарии по двум причинам. Во-первых, количество пробелов перед звездочкой не является одинаковым в разных строках. Во-вторых, пятая строка начинается с символа табуляции, который не является пробелом. Весь текст из строк со второй по пятую обрабатывается как часть комментария.

```
/**
  * <summary>
  * text
*  text2
    *  </summary>
*/
```

Для ссылки на XML-элементы (например, если функция обрабатывает определенные XML-элементы, которые требуется включить в комментарии XML-документации) можно использовать стандартный механизм заключения в скобки (`<` и `>`). Для ссылки на универсальные идентификаторы в элементах ссылок кода (`cref`) можно использовать escape-символы (например, `cref="List<T>"`) или фигурные скобки (`cref="List{T}"`). В особом случае компилятор анализирует фигурные скобки, как угловые, чтобы при ссылке на универсальные идентификаторы сделать комментарий документации менее громоздким.

> **NOTE**
> Комментарии XML-документации не являются метаданными. Они не включаются в скомпилированную сборку, и поэтому не доступны посредством отражения.

Средства, принимающие входные данные XML-документации

Следующие средства создают выходные данные на основе XML-комментариев:

- DocFX: *DocFX* — это генератор документации API для .NET, который сейчас поддерживает C#, Visual Basic и F#. Он также позволяет изменять создаваемую справочную документацию. DocFX создает статический веб-сайт HTML на основе исходного кода и файлов Markdown. Кроме того, DocFX предоставляет гибкие возможности по настройке макета и стиля веб-сайта с помощью шаблонов. Можно также создавать пользовательские шаблоны.
- Sandcastle: *средства Sandcastle* создают файлы справки для библиотек управляемых классов, содержащие как концептуальные страницы справки, так и страницы справки для API. Средства Sandcastle работают из командной строки и не имеют графического интерфейса, функций управления проектами или автоматизированного процесса сборки. Средство Sandcastle Help File Builder предоставляет автономный пользовательский интерфейс и инструменты командной строки для создания файла справки в автоматическом режиме. Также доступен пакет интеграции Visual Studio, позволяющий создавать проекты справки и управлять ими только из Visual Studio.
- Doxygen: *Doxygen* создает интерактивный обозреватель документации (в формате HTML) или автономное справочное руководство (в формате LaTeX) на основе набора исходных файлов с комментариями документации. Также поддерживается создание документации в форматах RTF (MS Word), PostScript, PDF с гиперссылками, HTML со сжатием, DocBook и страницы руководства Unix. Doxygen можно настроить для извлечения структуры кода из исходных файлов без комментариев документации.

Строки идентификаторов

Каждый тип или член хранится в элементе в выходном XML-файле. Каждый из этих элементов имеет уникальную строку идентификатора, определяющую тип или член. Строка идентификатора должна учитывать операторы, параметры, возвращаемые значения, параметры универсального типа, а также параметры `ref` , `in` и `out` . Чтобы закодировать все эти потенциальные элементы, компилятор следует

четко определенным правилам при создании строк идентификаторов. С помощью строки идентификатора программы, обрабатывающие XML-файл, могут определить, к какому элементу метаданных или отражения .NET применяется эта документация.

Компилятор соблюдает следующие правила при формировании строк идентификаторов.

- В строке нет пробелов.
- Первая часть строки определяет тип элемента, использующего один символ, за которым следует двоеточие. Используются следующие типы элементов.

ЗНАК	ТИП ЧЛЕНА	ПРИМЕЧАНИЯ
Нет	namespace	Вы не можете присвоить комментарий документации пространству имен, но можете создать для него cref-ссылку, если такие ссылки поддерживаются.
T	type	В качестве типа может использоваться класс, интерфейс, структура, перечисление или делегат.
C	поле	
C	свойство;	Включает индексаторы или другие индексированные свойства.
M	method	Включает специальные методы, такие как конструкторы и операторы.
E	event	
!	текст ошибки	Остальная часть строки предоставляет сведения об ошибке. Компилятор C# создает сведения об ошибках для ссылок, которые не могут быть разрешены.

- Вторая часть строки содержит полное имя элемента, начиная от корня пространства имен. Имя элемента, включающие типы и пространства имен разделяются точками. Если в имени самого элемента есть точки, они заменяются символами решетки ("#"). Предполагается, что в именах элементов не может содержаться символ решетки. Например, полное имя конструктора для объекта String имеет вид System.String.#ctor.
- Для свойств и методов указывается список параметров, заключенный в круглые скобки. Если параметры не используются, скобки отсутствуют. Параметры разделяются запятыми. Кодирование каждого параметра прямо соответствует тому, как он кодируется в сигнатуре .NET (определения всех элементов в следующем списке, записанных заглавными буквами, см. в разделе Microsoft.VisualStudio.CorDebugInterop.CorElementType):
  - Базовые типы. Обычные типы (`ELEMENT_TYPE_CLASS` или `ELEMENT_TYPE_VALUETYPE`) представляются в виде полного имени типа.

- Встроенные типы (например, `ELEMENT_TYPE_I4`, `ELEMENT_TYPE_OBJECT`, `ELEMENT_TYPE_STRING`, `ELEMENT_TYPE_TYPEDBYREF` и `ELEMENT_TYPE_VOID`) представляются в виде полного имени соответствующего полного типа. Например, `System.Int32` или `System.TypedReference`.
- `ELEMENT_TYPE_PTR` представляется в виде "*" после имени измененного типа.
- `ELEMENT_TYPE_BYREF` представляется в виде "@" после имени измененного типа.
- `ELEMENT_TYPE_CMOD_OPT` представляется в виде "!" и полного имени класса модификатора после имени измененного типа.
- `ELEMENT_TYPE_SZARRAY` представляется в виде "[]" после типа элемента массива.
- `ELEMENT_TYPE_ARRAY` представляется как [*нижняя_граница*:`size`,*нижняя_граница*:`size`], где число запятых на единицу меньше размерности массива, а нижние границы и размер каждого измерения, если они известны, представлены в десятичном формате. Если нижняя граница или размер не указаны, они опускаются. Если нижняя граница и размер для определенной размерности опущены, опускается и символ ":". Например, двухмерный массив со значением 1 для нижних границ без указания размеров обозначается так: "[1:,1:]".

- Только для операторов преобразования (`op_Implicit` и `op_Explicit`) возвращаемое значение метода кодируется как символ `~`, за которым следует тип возвращаемого значения. Например: `<member name="M:System.Decimal.op_Explicit(System.Decimal arg)~System.Int32">` — это тег для оператора приведения `public static explicit operator int (decimal value);`, объявленного в классе `System.Decimal`.

- Для универсальных типов в конце имени указывается символ обратного апострофа и число, которое обозначает количество параметров универсального типа. Например: `<member name="T:SampleClass``2">` — это тег для типа, который определен как `public class SampleClass<T, U>`. Для методов, принимающих в качестве параметров универсальные типы, параметры универсального типа указываются в виде чисел, которым предшествуют символы обратного апострофа (например, `0,`1). Каждое число представляет нотацию массива с отсчетом индексации с нуля для параметров универсального типа.

  - `ELEMENT_TYPE_PINNED` представляется в виде "^" после имени измененного типа. Компилятор C# никогда не создает такую кодировку.
  - `ELEMENT_TYPE_CMOD_REQ` представляется в виде "|" и полного имени класса модификатора после имени измененного типа. Компилятор C# никогда не создает такую кодировку.
  - `ELEMENT_TYPE_GENERICARRAY` представляется в виде "[?]" после типа элемента массива. Компилятор C# никогда не создает такую кодировку.
  - `ELEMENT_TYPE_FNPTR` представляется как "=FUNC:`type`(*подпись*)", где `type` обозначает возвращаемый тип, а *подпись* представляет аргументы метода. Если аргументы не используются, скобки опускаются. Компилятор C# никогда не создает такую кодировку.
  - Следующие компоненты сигнатуры не представляются, поскольку они никогда не используются для различения перегруженных методов:
    - соглашение о вызовах;
    - тип возвращаемого значения;
    - `ELEMENT_TYPE_SENTINEL`

Примеры ниже демонстрируют, как создаются строки идентификаторов для класса и его элементов.

```
namespace MyNamespace
{
    /// <summary>
    /// Enter description here for class X.
    /// ID string generated is "T:MyNamespace.X".
    /// </summary>
    public unsafe class MyClass
    {
```

```
{
        /// <summary>
        /// Enter description here for the first constructor.
        /// ID string generated is "M:MyNamespace.MyClass.#ctor".
        /// </summary>
        public MyClass() { }

        /// <summary>
        /// Enter description here for the second constructor.
        /// ID string generated is "M:MyNamespace.MyClass.#ctor(System.Int32)".
        /// </summary>
        /// <param name="i">Describe parameter.</param>
        public MyClass(int i) { }

        /// <summary>
        /// Enter description here for field message.
        /// ID string generated is "F:MyNamespace.MyClass.message".
        /// </summary>
        public string message;

        /// <summary>
        /// Enter description for constant PI.
        /// ID string generated is "F:MyNamespace.MyClass.PI".
        /// </summary>
        public const double PI = 3.14;

        /// <summary>
        /// Enter description for method func.
        /// ID string generated is "M:MyNamespace.MyClass.func".
        /// </summary>
        /// <returns>Describe return value.</returns>
        public int func() { return 1; }

        /// <summary>
        /// Enter description for method someMethod.
        /// ID string generated is
"M:MyNamespace.MyClass.someMethod(System.String,System.Int32@,System.Void*)".
        /// </summary>
        /// <param name="str">Describe parameter.</param>
        /// <param name="num">Describe parameter.</param>
        /// <param name="ptr">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public int someMethod(string str, ref int nm, void* ptr) { return 1; }

        /// <summary>
        /// Enter description for method anotherMethod.
        /// ID string generated is
"M:MyNamespace.MyClass.anotherMethod(System.Int16[],System.Int32[0:,0:])".
        /// </summary>
        /// <param name="array1">Describe parameter.</param>
        /// <param name="array">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public int anotherMethod(short[] array1, int[,] array) { return 0; }

        /// <summary>
        /// Enter description for operator.
        /// ID string generated is
"M:MyNamespace.MyClass.op_Addition(MyNamespace.MyClass,MyNamespace.MyClass)".
        /// </summary>
        /// <param name="first">Describe parameter.</param>
        /// <param name="second">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public static MyClass operator +(MyClass first, MyClass second) { return first; }

        /// <summary>
        /// Enter description for property.
        /// ID string generated is "P:MyNamespace.MyClass.prop".
        /// </summary>
        public int prop { get { return 1; } set { } }
```

```
        /// <summary>
        /// Enter description for event.
        /// ID string generated is "E:MyNamespace.MyClass.OnHappened".
        /// </summary>
        public event Del OnHappened;

        /// <summary>
        /// Enter description for index.
        /// ID string generated is "P:MyNamespace.MyClass.Item(System.String)".
        /// </summary>
        /// <param name="str">Describe parameter.</param>
        /// <returns></returns>
        public int this[string s] { get { return 1; } }

        /// <summary>
        /// Enter description for class Nested.
        /// ID string generated is "T:MyNamespace.MyClass.Nested".
        /// </summary>
        public class Nested { }

        /// <summary>
        /// Enter description for delegate.
        /// ID string generated is "T:MyNamespace.MyClass.Del".
        /// </summary>
        /// <param name="i">Describe parameter.</param>
        public delegate void Del(int i);

        /// <summary>
        /// Enter description for operator.
        /// ID string generated is "M:MyNamespace.MyClass.op_Explicit(MyNamespace.X)~System.Int32".
        /// </summary>
        /// <param name="myParameter">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public static explicit operator int(MyClass myParameter) { return 1; }
    }
}
```

Спецификация языка C#

Дополнительные сведения см. в приложении Спецификация языка C# в комментариях документации.

Рекомендуемые XML-теги для комментариев к документации по C#

18.12.2021 • 9 minutes to read

В комментариях к документации по C# используются XML-элементы, что позволяет определить структуру выходной документации. Одним из последствий использования этой функции является то, что в комментарии к документации можно добавить любой допустимый XML-код. Компилятор C# копирует эти элементы в выходной XML-файл. Несмотря на то что в комментариях (включая любой допустимый HTML-элемент) можно использовать любой допустимый XML-код, документирование кода рекомендуется по многим причинам.

Далее приводится ряд рекомендаций, распространенные сценарии использования и вопросы, которые нужно иметь в виду при работе с тегами XML-документации в коде C#. Несмотря на то что теги можно поместить в комментарии к документации, в этой статье описываются рекомендуемые теги для наиболее распространенных конструкций языка. Во всех случаях следует соблюдать следующие рекомендации:

- Для обеспечения согласованности все открытые типы и их члены должны быть задокументированы.
- Закрытые члены можно также задокументировать с помощью XML-комментариев. Однако это приводит к раскрытию внутренних (возможно, конфиденциальных) процессов библиотеки.
- Как минимум, типы и их члены должен иметь тег `<summary>`, так как его содержимое требуется для IntelliSense.
- Текст документации должны быть написан с использованием законченных предложений, в конце которых ставится точка.
- Разделяемые классы полностью поддерживаются, и данные о документации объединяются в одну запись для этого типа.

Документация XML начинается с `///`. При создании проекта шаблоны добавляют несколько строк `///`. Обработка этих комментариев имеет некоторые ограничения.

- Документация должна представлять собой XML с правильным форматом. Если XML сформирован неправильно, компилятор выдает предупреждение. Файл документации будет содержать комментарий, в котором сообщается, что обнаружена ошибка.
- Некоторые рекомендуемые теги имеют особые значения.
  - Тег `<param>` используется для описания параметров. При использовании этого тега компилятор проверяет, что параметр существует и все параметры описаны в документации. При сбое проверки компилятор выдает предупреждение.
  - Атрибут `cref` может быть присоединен к любому тегу для ссылки на элемент кода. Компилятор проверяет наличие этого элемента кода. При сбое проверки компилятор выдает предупреждение. Компилятор учитывает любые операторы `using` при поиске типа, описанного в атрибуте `cref`.
  - Тег `<summary>` используется технологией IntelliSense в Visual Studio для отображения дополнительных сведений о типе или элементе.

> **NOTE**
> XML-файл не предоставляет полную информацию о типе и членах (например, он не содержит никаких сведений о типе). Чтобы получить полную информацию о типе или элементе, используйте файл документации вместе с отражением для текущего типа или элемента.

- Разработчики могут создавать собственные наборы тегов. Компилятор скопирует их в выходной файл.

Некоторые из рекомендуемых тегов можно использовать для любого элемента языка. Другие имеют более специализированное использование. Наконец, некоторые теги используются для форматирования текста в документации. В этой статье описываются рекомендуемые теги, упорядоченные по их использованию.

Компилятор проверяет синтаксис элементов, за которыми следует один * в следующем списке. Visual Studio предоставляет IntelliSense для тегов, проверенных компилятором, и всех тегов, за которыми следует ** в следующем списке. Помимо указанных здесь тегов, компилятор и Visual Studio проверяют теги `<b>`, `<i>`, `<u>`, `<br/>` и `<a>`. Компилятор также проверяет `<tt>`, который является устаревшим тегом HTML.

- Общие теги, используемые для нескольких элементов. Эти теги являются минимальным набором для любого API.
  - `<summary>`: значение этого элемента отображается в IntelliSense в Visual Studio.
  - `<remarks>` **
- Теги, используемые для членов. Эти теги используются при документировании методов и свойств.
  - `<returns>`: значение этого элемента отображается в IntelliSense в Visual Studio.
  - `<param>` *: значение этого элемента отображается в IntelliSense в Visual Studio.
  - `<paramref>`
  - `<exception>` *
  - `<value>`: значение этого элемента отображается в IntelliSense в Visual Studio.
- Формат вывода документации. Эти теги предоставляют направления форматирования для средств, создающих документацию.
  - `<para>`
  - `<list>`
  - `<c>`
  - `<code>`
  - `<example>` **
- Повторное использование текста документации. Эти теги предоставляют средства, упрощающие использование комментариев XML.
  - `<inheritdoc>` **
  - `<include>` *
- Создание ссылок и гиперссылок. Эти теги создают ссылки на другую документацию.
  - `<see>` *
  - `<seealso>` *
  - `<cref>`
  - `<href>`
- Теги для универсальных типов и методов. Эти теги используются только для универсальных типов и методов
  - `<typeparam>` *: значение этого элемента отображается в IntelliSense в Visual Studio.
  - `<typeparamref>`

> **NOTE**
> Комментарии документации не применяются к пространству имен.

Чтобы ввести в текст комментария документации угловые скобки, используйте для символов `<` и `>` коды HTML `<` и `>` соответственно. Это показано в следующем примере.

```
/// <summary>
/// This property always returns a value &lt; 1.
/// </summary>
```

Общие теги

<summary>

```
<summary>description</summary>
```

Тег `<summary>` следует использовать для описания типа или элемента типа. Чтобы добавить дополнительную информацию в описание типа, используйте <remarks>. Чтобы включить средства документации, такие как DocFX и Sandcastle, для создания внутренних гиперссылок на страницы документации для элементов кода, используйте атрибут cref. Текст в теге `<summary>` является единственным источником сведений о типе для технологии IntelliSense и также отображается в окне обозревателя объектов.

<remarks>

```
<remarks>
description
</remarks>
```

Тег `<remarks>` позволяет добавить сведения о типе или члене типа, дополняющие информацию, указанную с помощью тега <summary>. Эти сведения отображаются в окне "Обозреватель объектов". Этот тег может содержать более длинные объяснения. Возможно, вы обнаружите, что использование разделов `CDATA` для разметки упрощает написание. Такие средства, как docfx, обрабатывают текст разметки в разделах `CDATA`.

Члены документа

<returns>

```
<returns>description</returns>
```

Тег `<returns>` следует использовать в комментариях к объявлению метода для описания возвращаемого значения.

<param>

```
<param name="name">description</param>
```

- `name`: имя параметра метода. Имя заключается в двойные кавычки (" "). Имена параметров должны соответствовать сигнатуре API. Если один или несколько параметров не охвачены, компилятор выдает предупреждение. Компилятор также выдает предупреждение, если значение `name` не соответствует формальному параметру в объявлении метода.

Тег `<param>` следует использовать в комментариях к объявлению метода для описания одного из параметров такого метода. Чтобы задокументировать несколько параметров, используйте несколько

тегов `<param>`. Текст тега `<param>` отображается в IntelliSense, обозревателе объектов и веб-отчете по комментариям к коду.

<paramref>

```
<paramref name="name"/>
```

- `name`: имя параметра, на который указывается ссылка. Имя заключается в двойные кавычки (" ").

Тег `<paramref>` позволяет указать, что слово в комментариях к коду, например в блоке `<summary>` или `<remarks>`, ссылается на параметр. В XML-файл такое слово может выделяться особым образом, например курсивом или полужирным шрифтом.

<exception>

```
<exception cref="member">description</exception>
```

- cref = "`member`": ссылка на исключение, которое доступно из текущей среды компиляции. Компилятор проверяет, существует ли исключение, и приводит `member` к каноническому имени элемента в выходных XML-данных. `member` необходимо заключать в двойные кавычки (" ").

Тег `<exception>` служит для указания возможных исключений. Этот тег может применяться к определениям методов, свойств, событий и индексаторов.

<value>

```
<value>property-description</value>
```

Тег `<value>` позволяет описывать значение, которое представляется свойством. При добавлении свойства с помощью мастера создания кода из среды разработки Visual Studio .NET для нового свойства будет добавлен тег <summary>. Следует вручную добавить тег `<value>` для описания значения, которое представляется свойством.

Форматирование выходных данных документации

<para>

```
<remarks>
    <para>
        This is an introductory paragraph.
    </para>
    <para>
        This paragraph contains more details.
    </para>
</remarks>
```

Тег `<para>` используется внутри другого тега, например <summary>, <remarks> или <returns>, и позволяет добавить к тексту структуру. Тег `<para>` создает абзац с двойным отступом. Используйте тег `<br/>`, если хотите добавить в него один абзац.

<list>

```
<list type="bullet|number|table">
    <listheader>
        <term>term</term>
        <description>description</description>
    </listheader>
    <item>
        <term>Assembly</term>
        <description>The library or executable built from a compilation.</description>
    </item>
</list>
```

Блок `<listheader>` используется для определения строки заголовка в таблице или списке определений. При определении таблицы необходимо ввести данные для `term` в заголовке. Каждый элемент в списке указывается в блоке `<item>`. При создании списка определений необходимо указать одновременно `term` и `description`. Тем не менее для таблицы, маркированного или нумерованного списка достаточно ввести только `description`. Число блоков `<item>` в списке или таблице не ограничено.

<c>

```
<c>text</c>
```

С помощью тега `<c>` можно указать, что текст в описании необходимо пометить как код. Чтобы определить несколько строк в качестве кода, используйте тег <code>.

<code>

```
<code>
    var index = 5;
    index++;
</code>
```

Тег `<code>` используется для указания нескольких строк кода. С помощью тега <c> можно указать, что однострочный текст в описании необходимо пометить как код.

<example>

```
<example>
This shows how to increment an integer.
<code>
    var index = 5;
    index++;
</code>
</example>
```

Тег `<example>` позволяет указать пример использования метода или другого элемента библиотеки. Пример обычно включает использование тега <code>.

Повторное использование текста документации

<inheritdoc>

```
<inheritdoc [cref=""] [path=""]/>
```

Наследование XML-комментариев от базовых классов, интерфейсов и аналогичных методов. Использование `inheritdoc` позволяет обойтись без копирования и вставки одинаковых XML-комментариев и автоматически поддерживать их синхронизацию. Обратите внимание, что при

добавлении тега `<inheritdoc>` к типу все члены также будут наследовать эти комментарии.

- `cref`: указывает элемент, из которого следует наследовать документацию. Унаследованные теги не переопределяют уже определенные теги для текущего элемента.
- `path`: запрос с выражением XPath, в результате которого выводится набор узлов. С помощью этого атрибута можно отфильтровать теги, которые следует включить в наследуемую документацию или исключить из нее.

Добавьте XML-комментарии в базовые классы или интерфейсы, и InheritDoc скопирует их во все реализации классов. Добавьте XML-комментарии в синхронные методы, и InheritDoc скопирует их в асинхронные версии аналогичных методов. Если нужно скопировать комментарии из определенного элемента, укажите нужный элемент в атрибуте `cref`.

<include>

```
<include file='filename' path='tagpath[@name="id"]' />
```

- `filename`: имя XML-файла, содержащего документацию. Имя файла может быть квалифицировано с помощью относительного пути к файлу исходного кода. `filename` необходимо заключать в одинарные кавычки (' ').
- `tagpath`: путь тегов в `filename`, который ведет к тегу `name`. Путь необходимо заключать в одинарные кавычки (' ').
- `name`: спецификатор имени в теге, предшествующий комментариям. `name` будет иметь идентификатор `id`.
- `id`: идентификатор тега, который предшествует комментариям. Идентификатор заключается в двойные кавычки (" ").

Тег `<include>` позволяет задать ссылку на комментарии в другом файле, которые описывают типы и элементы вашего исходного кода. Включение внешнего файла является альтернативой размещению комментариев документации непосредственно в файле исходного кода. Помещая комментарии документации в отдельный файл, вы можете реализовать управление их версиями отдельно от версий исходного кода. В этом случае файл исходного кода может быть извлечен для изменения одним пользователем, а файл документации — другим. Тег `<include>` использует XML-синтаксис XPath. Сведения об использовании тега `<include>` см. в документации по XPath.

Создание ссылок и гиперссылок

<see>

```
/// <see cref="member"/>
// or
/// <see cref="member">Link text</see>
// or
/// <see href="link">Link Text</see>
// or
/// <see langword="keyword"/>
```

- `cref="member"`: ссылка на член или поле, которые доступны для вызова из текущей среды компиляции. Компилятор проверяет, существует ли элемент кода, и передает `member` в имя элемента в выходных XML-данных. *member* необходимо заключать в двойные кавычки (" "). Можно указать другой текст ссылки для "cref", используя отдельный закрывающий тег.
- `href="link"`: гиперссылка на заданный URL-адрес. Например, `<see href="https://github.com">GitHub</see>` формирует гиперссылку с текстом GitHub, которая ведет на

сайт `https://github.com`.

- `langword="keyword"`: ключевое слово языка, например `true`.

Тег `<see>` позволяет задать ссылку из текста. С помощью тега <seealso> можно указать, что текст должен быть размещен в разделе "См. также". Используйте атрибут cref для создания внутренних гиперссылок на страницы документации для элементов кода. Вы включаете параметры типа, чтобы указать ссылку на универсальный тип или метод, например `cref="IDictionary{T, U}"`. Кроме того, `href` является допустимым атрибутом, который будет функционировать как гиперссылка.

<seealso>

```
/// <seealso cref="member"/>
// or
/// <seealso href="link">Link Text</seealso>
```

- `cref="member"`: ссылка на член или поле, которые доступны для вызова из текущей среды компиляции. Компилятор проверяет, существует ли элемент кода, и передает `member` в имя элемента в выходных XML-данных. `member` необходимо заключать в двойные кавычки (" ").
- `href="link"`: гиперссылка на заданный URL-адрес. Например, `<seealso href="https://github.com">GitHub</seealso>` формирует гиперссылку с текстом GitHub, которая ведет на сайт `https://github.com`.

С помощью тега `<seealso>` можно указать текст, который должен отображаться в разделе **См. также**. Тег <see> позволяет задать ссылку из текста. Тег `seealso` нельзя вложить в тег `summary`.

Атрибут cref

Атрибут `cref` в теге XML-документации означает "кодовая ссылка". Он указывает, что текст внутри тега представляет собой элемент кода, например тип, метод или свойство. Средства создания документации, такие как DocFX и Sandcastle, используют атрибуты `cref` для автоматического создания гиперссылок на страницу, где документирован тип или член.

Атрибут href

Атрибут `href` означает ссылку на веб-страницу. Его можно использовать для прямой ссылки на интерактивную документацию по API или библиотеке.

Универсальные типы и методы

<typeparam>

```
<typeparam name="TResult">The type returned from this method</typeparam>
```

- `TResult`: имя параметра типа. Имя заключается в двойные кавычки (" ").

Тег `<typeparam>` следует использовать в комментариях к объявлению универсального типа или метода для описания параметра типа. Добавьте такой тег для каждого параметра типа универсального типа или метода. Текст для тега `<typeparam>` будет отображаться в IntelliSense.

<typeparamref>

```
<typeparamref name="TKey"/>
```

- `TKey`: имя параметра типа. Имя заключается в двойные кавычки (" ").

Используйте этот тег, чтобы разрешить получателям файла документации форматировать слово

определенным образом, например курсивным шрифтом.

Пользовательские теги

Все описанные выше теги распознаются компилятором C#. Тем не менее пользователь может определять собственные теги. Инструменты, такие как Sandcastle, обеспечивают поддержку дополнительных тегов (например, <event> и <note>) и даже поддержку пространств имен документирования. Средства создания пользовательской или внутренней документации также можно использовать со стандартными тегами. Кроме того, поддерживается несколько выходных форматов — от HTML до PDF.

Примеры комментариев к XML-документации

18.12.2021 • 17 minutes to read

В этой статье содержится три примера для добавления комментариев к XML-документации к большинству элементов языка C#. В первом примере показано, как задокументировать класс с разными членами. Во втором показано, как можно повторно использовать объяснения для иерархии классов или интерфейсов. В третьем примере показаны теги, используемые для универсальных классов и членов. Во втором и третьем примерах используются концепции, рассмотренные в первом примере.

Документирование класса, структуры или интерфейса

В следующем примере показаны общие элементы языка и теги, которые, скорее всего, будут использоваться для описания этих элементов. Комментарии к документации служат для описания использования тегов, а не самого класса.

```
/// <summary>
/// Every class and member should have a one sentence
/// summary describing its purpose.
/// </summary>
/// <remarks>
/// You can expand on that one sentence summary to
/// provide more information for readers. In this case,
/// the <c>ExampleClass</c> provides different C#
/// elements to show how you would add documentation
///comments for most elements in a typical class.
/// <para>
/// The remarks can add multiple paragraphs, so you can
/// write detailed information for developers that use
/// your work. You should add everything needed for
/// readers to be successful. This class contains
/// examples for the following:
/// </para>
/// <list type="table">
/// <item>
/// <term>Summary</term>
/// <description>
/// This should provide a one sentence summary of the class or member.
/// </description>
/// </item>
/// <item>
/// <term>Remarks</term>
/// <description>
/// This is typically a more detailed description of the class or member
/// </description>
/// </item>
/// <item>
/// <term>para</term>
/// <description>
/// The para tag separates a section into multiple paragraphs
/// </description>
/// </item>
/// <item>
/// <term>list</term>
/// <description>
/// Provides a list of terms or elements
/// </description>
/// </item>
/// <item>
/// <term>returns, param</term>
```

```
/// <description>
/// Used to describe parameters and return values
/// </description>
/// </item>
/// <item>
/// <term>value</term>
/// <description>Used to describe properties</description>
/// </item>
/// <item>
/// <term>exception</term>
/// <description>
/// Used to describe exceptions that may be thrown
/// </description>
/// </item>
/// <item>
/// <term>c, cref, see, seealso</term>
/// <description>
/// These provide code style and links to other
/// documentation elements
/// </description>
/// </item>
/// <item>
/// <term>example, code</term>
/// <description>
/// These are used for code examples
/// </description>
/// </item>
/// </list>
/// <para>
/// The list above uses the "table" style. You could
/// also use the "bullet" or "number" style. Neither
/// would typically use the "term" element.
/// <br/>
/// Note: paragraphs are double spaced. Use the *br*
/// tag for single spaced lines.
/// </para>
/// </remarks>
public class ExampleClass
{
    /// <value>
    /// The <c>Label</c> property represents a label
    /// for this instance.
    /// </value>
    /// <remarks>
    /// The <see cref="Label"/> is a <see langword="string"/>
    /// that you use for a label.
    /// <para>
    /// Note that there isn't a way to provide a "cref" to
    /// each accessor, only to the property itself.
    /// </para>
    /// </remarks>
    public string Label
    {
        get;
        set;
    }

    /// <summary>
    /// Adds two integers and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two integers.
    /// </returns>
    /// <param name="left">
    /// The left operand of the addition.
    /// </param>
    /// <param name="right">
    /// The right operand of the addition.
    /// </param>
```

```
        /// <example>
        /// <code>
        /// int c = Math.Add(4, 5);
        /// if (c > 10)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// <exception cref="System.OverflowException">
        /// Thrown when one parameter is
        /// <see cref="Int32.MaxValue">MaxValue</see> and the other is
        /// greater than 0.
        /// Note that here you can also use
        /// <see href="https://docs.microsoft.com/dotnet/api/system.int32.maxvalue"/>
        ///  to point a web page instead.
        /// </exception>
        /// <see cref="ExampleClass"/> for a list of all
        /// the tags in these examples.
        /// <seealso cref="ExampleClass.Label"/>
        public static int Add(int left, int right)
        {
            if ((left == int.MaxValue && right > 0) || (right == int.MaxValue && left > 0))
                throw new System.OverflowException();

            return left + right;
        }
    }

    /// <summary>
    /// This is an example of a positional record.
    /// </summary>
    /// <remarks>
    /// There isn't a way to add XML comments for properties
    /// created for positional records, yet. The language
    /// design team is still considering what tags should
    /// be supported, and where. Currently, you can use
    /// the "param" tag to describe the parameters to the
    /// primary constructor.
    /// </remarks>
    /// <param name="FirstName">
    /// This tag will apply to the primary constructor parameter.
    /// </param>
    /// <param name="LastName">
    /// This tag will apply to the primary constructor parameter.
    /// </param>
    public record Person(string FirstName, string LastName);
}
```

Добавление документации может перегрузить исходный код большими наборами комментариев, предназначенными для пользователей библиотеки. Тег `<Include>` используется для отделения XML-комментариев от исходного кода. Исходный код ссылается на XML-файл с помощью тега `<Include>`:

```
/// <include file='xml_include_tag.xml' path='MyDocs/MyMembers[@name="test"]/*' />
class Test
{
    static void Main()
    {
    }
}

/// <include file='xml_include_tag.xml' path='MyDocs/MyMembers[@name="test2"]/*' />
class Test2
{
    public void Test()
    {
    }
}
```

Второй файл, *xml_include_tag.xml*, содержит комментарии к документации.

```
<MyDocs>
    <MyMembers name="test">
        <summary>
        The summary for this type.
        </summary>
    </MyMembers>
    <MyMembers name="test2">
        <summary>
        The summary for this other type.
        </summary>
    </MyMembers>
</MyDocs>
```

Документирование иерархии классов и интерфейсов

Элемент `<inheritdoc>` означает, что тип или член *наследует* комментарии к документации от базового класса или интерфейса. Элемент `<inheritdoc>` с атрибутом `cref` также можно использовать для наследования комментариев от члена того же типа. В следующем примере показано, как использовать этот класс. Обратите внимание, что при добавлении атрибута `inheritdoc` к типу наследуются комментарии члена. Можно запретить использование наследуемых комментариев, записав комментарии к членам в производном типе. Они будут выбраны для наследуемых комментариев.

```
/// <summary>
/// A summary about this class.
/// </summary>
/// <remarks>
/// These remarks would explain more about this class.
/// In this example, these comments also explain the
/// general information about the derived class.
/// </remarks>
public class MainClass
{
}

///<inheritdoc/>
public class DerivedClass : MainClass
{
}

/// <summary>
/// This interface would describe all the methods in
/// its contract.
/// </summary>
/// <remarks>
```

```
/// <remarks>
/// While elided for brevity, each method or property
/// in this interface would contain docs that you want
/// to duplicate in each implementing class.
/// </remarks>
public interface ITestInterface
{
    /// <summary>
    /// This method is part of the test interface.
    /// </summary>
    /// <remarks>
    /// This content would be inherited by classes
    /// that implement this interface when the
    /// implementing class uses "inheritdoc"
    /// </remarks>
    /// <returns>The value of <paramref name="arg" /> </returns>
    /// <param name="arg">The argument to the method</param>
    int Method(int arg);
}

///<inheritdoc cref="ITestInterface"/>
public class ImplementingClass : ITestInterface
{
    // doc comments are inherited here.
    public int Method(int arg) => arg;
}

/// <summary>
/// This class shows hows you can "inherit" the doc
/// comments from one method in another method.
/// </summary>
/// <remarks>
/// You can inherit all comments, or only a specific tag,
/// represented by an xpath expression.
/// </remarks>
public class InheritOnlyReturns
{
    /// <summary>
    /// In this example, this summary is only visible for this method.
    /// </summary>
    /// <returns>A boolean</returns>
    public static bool MyParentMethod(bool x) { return x; }

    /// <inheritdoc cref="MyParentMethod" path="/returns"/>
    public static bool MyChildMethod() { return false; }
}

/// <Summary>
/// This class shows an example ofsharing comments across methods.
/// </Summary>
public class InheritAllButRemarks
{
    /// <summary>
    /// In this example, this summary is visible on all the methods.
    /// </summary>
    /// <remarks>
    /// The remarks can be inherited by other methods
    /// using the xpath expression.
    /// </remarks>
    /// <returns>A boolean</returns>
    public static bool MyParentMethod(bool x) { return x; }

    /// <inheritdoc cref="MyParentMethod" path="//*[not(self::remarks)]"/>
    public static bool MyChildMethod() { return false; }
}
```

Универсальные типы

Используйте тег `<typeparam>` для описания параметров типа в универсальных типах и методах. Значение атрибута `cref` требует нового синтаксиса для ссылки на универсальный метод или класс:

```
/// <summary>
/// This is a generic class.
/// </summary>
/// <remarks>
/// This example shows how to specify the <see cref="GenericClass{T}"/>
/// type as a cref attribute.
/// In generic classes and methods, you'll often want to reference the
/// generic type, or the type parameter.
/// </remarks>
class GenericClass<T>
{
    // Fields and members.
}

/// <Summary>
/// This shows examples of typeparamref and typeparam tags
/// </Summary>
public class ParamsAndParamRefs
{
    /// <summary>
    /// The GetGenericValue method.
    /// </summary>
    /// <remarks>
    /// This sample shows how to specify the <see cref="GetGenericValue"/>
    /// method as a cref attribute.
    /// The parameter and return value are both of an arbitrary type,
    /// <typeparamref name="T"/>
    /// </remarks>
    public static T GetGenericValue<T>(T para)
    {
        return para;
    }
}
```

Пример класса Math

В следующем коде показан реалистичный пример добавления комментариев к документу в библиотеку Math.

```
namespace TaggedLibrary
{
    /*
        The main Math class
        Contains all methods for performing basic math functions
    */
        /// <summary>
    /// The main <c>Math</c> class.
    /// Contains all methods for performing basic math functions.
    /// <list type="bullet">
    /// <item>
    /// <term>Add</term>
    /// <description>Addition Operation</description>
    /// </item>
    /// <item>
    /// <term>Subtract</term>
    /// <description>Subtraction Operation</description>
    /// </item>
    /// <item>
    /// <term>Multiply</term>
    /// <description>Multiplication Operation</description>
    /// </item>
```

```
    /// <item>
    /// <term>Divide</term>
    /// <description>Division Operation</description>
    /// </item>
    /// </list>
    /// </summary>
    /// <remarks>
    /// <para>
    /// This class can add, subtract, multiply and divide.
    /// </para>
    /// <para>
    /// These operations can be performed on both
    /// integers and doubles.
    /// </para>
    /// </remarks>
    public class Math
    {
        // Adds two integers and returns the result
        /// <summary>
        /// Adds two integers <paramref name="a"/> and <paramref name="b"/>
        ///  and returns the result.
        /// </summary>
        /// <returns>
        /// The sum of two integers.
        /// </returns>
        /// <example>
        /// <code>
        /// int c = Math.Add(4, 5);
        /// if (c > 10)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// <exception cref="System.OverflowException">
        /// Thrown when one parameter is <see cref="Int32.MaxValue"/> and the other
        /// is greater than 0.
        /// </exception>
        /// See <see cref="Math.Add(double, double)"/> to add doubles.
        /// <seealso cref="Math.Subtract(int, int)"/>
        /// <seealso cref="Math.Multiply(int, int)"/>
        /// <seealso cref="Math.Divide(int, int)"/>
        /// <param name="a">An integer.</param>
        /// <param name="b">An integer.</param>
        public static int Add(int a, int b)
        {
            // If any parameter is equal to the max value of an integer
            // and the other is greater than zero
            if ((a == int.MaxValue && b > 0) ||
                (b == int.MaxValue && a > 0))
            {
                throw new System.OverflowException();
            }
            return a + b;
        }

        // Adds two doubles and returns the result
        /// <summary>
        /// Adds two doubles <paramref name="a"/> and <paramref name="b"/>
        /// and returns the result.
        /// </summary>
        /// <returns>
        /// The sum of two doubles.
        /// </returns>
        /// <example>
        /// <code>
        /// double c = Math.Add(4.5, 5.4);
        /// if (c > 10)
        /// {
```

```
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// <exception cref="System.OverflowException">
        /// Thrown when one parameter is max and the other
        /// is greater than 0.</exception>
        /// See <see cref="Math.Add(int, int)"/> to add integers.
        /// <seealso cref="Math.Subtract(double, double)"/>
        /// <seealso cref="Math.Multiply(double, double)"/>
        /// <seealso cref="Math.Divide(double, double)"/>
        /// <param name="a">A double precision number.</param>
        /// <param name="b">A double precision number.</param>
        public static double Add(double a, double b)
        {
            // If any parameter is equal to the max value of an integer
            // and the other is greater than zero
            if ((a == double.MaxValue && b > 0)
                || (b == double.MaxValue && a > 0))
            {
                throw new System.OverflowException();
            }

            return a + b;
        }

        // Subtracts an integer from another and returns the result
        /// <summary>
        /// Subtracts <paramref name="b"/> from <paramref name="a"/>
        /// and returns the result.
        /// </summary>
        /// <returns>
        /// The difference between two integers.
        /// </returns>
        /// <example>
        /// <code>
        /// int c = Math.Subtract(4, 5);
        /// if (c > 1)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// See <see cref="Math.Subtract(double, double)"/> to subtract doubles.
        /// <seealso cref="Math.Add(int, int)"/>
        /// <seealso cref="Math.Multiply(int, int)"/>
        /// <seealso cref="Math.Divide(int, int)"/>
        /// <param name="a">An integer.</param>
        /// <param name="b">An integer.</param>
        public static int Subtract(int a, int b)
        {
            return a - b;
        }

        // Subtracts a double from another and returns the result
        /// <summary>
        /// Subtracts a double <paramref name="b"/> from another
        /// double <paramref name="a"/> and returns the result.
        /// </summary>
        /// <returns>
        /// The difference between two doubles.
        /// </returns>
        /// <example>
        /// <code>
        /// double c = Math.Subtract(4.5, 5.4);
        /// if (c > 1)
        /// {
        ///     Console.WriteLine(c);
        /// }
```

```
        /// </code>
        /// </example>
        /// See <see cref="Math.Subtract(int, int)"/> to subtract integers.
        /// <seealso cref="Math.Add(double, double)"/>
        /// <seealso cref="Math.Multiply(double, double)"/>
        /// <seealso cref="Math.Divide(double, double)"/>
        /// <param name="a">A double precision number.</param>
        /// <param name="b">A double precision number.</param>
        public static double Subtract(double a, double b)
        {
            return a - b;
        }

        // Multiplies two integers and returns the result
        /// <summary>
        /// Multiplies two integers <paramref name="a"/>
        /// and <paramref name="b"/> and returns the result.
        /// </summary>
        /// <returns>
        /// The product of two integers.
        /// </returns>
        /// <example>
        /// <code>
        /// int c = Math.Multiply(4, 5);
        /// if (c > 100)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// See <see cref="Math.Multiply(double, double)"/> to multiply doubles.
        /// <seealso cref="Math.Add(int, int)"/>
        /// <seealso cref="Math.Subtract(int, int)"/>
        /// <seealso cref="Math.Divide(int, int)"/>
        /// <param name="a">An integer.</param>
        /// <param name="b">An integer.</param>
        public static int Multiply(int a, int b)
        {
            return a * b;
        }

        // Multiplies two doubles and returns the result
        /// <summary>
        /// Multiplies two doubles <paramref name="a"/> and
        /// <paramref name="b"/> and returns the result.
        /// </summary>
        /// <returns>
        /// The product of two doubles.
        /// </returns>
        /// <example>
        /// <code>
        /// double c = Math.Multiply(4.5, 5.4);
        /// if (c > 100.0)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// See <see cref="Math.Multiply(int, int)"/> to multiply integers.
        /// <seealso cref="Math.Add(double, double)"/>
        /// <seealso cref="Math.Subtract(double, double)"/>
        /// <seealso cref="Math.Divide(double, double)"/>
        /// <param name="a">A double precision number.</param>
        /// <param name="b">A double precision number.</param>
        public static double Multiply(double a, double b)
        {
            return a * b;
        }
```

```
        // Divides an integer by another and returns the result
        /// <summary>
        /// Divides an integer <paramref name="a"/> by another
        /// integer <paramref name="b"/> and returns the result.
        /// </summary>
        /// <returns>
        /// The quotient of two integers.
        /// </returns>
        /// <example>
        /// <code>
        /// int c = Math.Divide(4, 5);
        /// if (c > 1)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// <exception cref="System.DivideByZeroException">
        /// Thrown when <paramref name="b"/> is equal to 0.
        /// </exception>
        /// See <see cref="Math.Divide(double, double)"/> to divide doubles.
        /// <seealso cref="Math.Add(int, int)"/>
        /// <seealso cref="Math.Subtract(int, int)"/>
        /// <seealso cref="Math.Multiply(int, int)"/>
        /// <param name="a">An integer dividend.</param>
        /// <param name="b">An integer divisor.</param>
        public static int Divide(int a, int b)
        {
            return a / b;
        }

        // Divides a double by another and returns the result
        /// <summary>
        /// Divides a double <paramref name="a"/> by another double
        /// <paramref name="b"/> and returns the result.
        /// </summary>
        /// <returns>
        /// The quotient of two doubles.
        /// </returns>
        /// <example>
        /// <code>
        /// double c = Math.Divide(4.5, 5.4);
        /// if (c > 1.0)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// <exception cref="System.DivideByZeroException">
        /// Thrown when <paramref name="b"/> is equal to 0.
        /// </exception>
        /// See <see cref="Math.Divide(int, int)"/> to divide integers.
        /// <seealso cref="Math.Add(double, double)"/>
        /// <seealso cref="Math.Subtract(double, double)"/>
        /// <seealso cref="Math.Multiply(double, double)"/>
        /// <param name="a">A double precision dividend.</param>
        /// <param name="b">A double precision divisor.</param>
        public static double Divide(double a, double b)
        {
            return a / b;
        }
    }
}
```

Может оказаться, что комментарии скрывают код. В последнем примере показано, как можно адаптировать эту библиотеку для использования тега `include`. Вы перемещаете всю документацию в

XML-файл:

```
<docs>
  <members name="math">
   <Math>
    <summary>
      The main <c>Math</c> class.
      Contains all methods for performing basic math functions.
      </summary>
      <remarks>
      <para>This class can add, subtract, multiply and divide.</para>
      <para>These operations can be performed on both integers and doubles.</para>
      </remarks>
    </Math>
    <AddInt>
      <summary>
      Adds two integers <paramref name="a"/> and <paramref name="b"/>
      and returns the result.
      </summary>
      <returns>
      The sum of two integers.
      </returns>
      <example>
      <code>
      int c = Math.Add(4, 5);
      if (c > 10)
      {
        Console.WriteLine(c);
      }
      </code>
      </example>
      <exception cref="System.OverflowException">Thrown when one
      parameter is max
      and the other is greater than 0.</exception>
      See <see cref="Math.Add(double, double)"/> to add doubles.
      <seealso cref="Math.Subtract(int, int)"/>
      <seealso cref="Math.Multiply(int, int)"/>
      <seealso cref="Math.Divide(int, int)"/>
      <param name="a">An integer.</param>
      <param name="b">An integer.</param>
    </AddInt>
    <AddDouble>
      <summary>
      Adds two doubles <paramref name="a"/> and <paramref name="b"/>
      and returns the result.
      </summary>
      <returns>
      The sum of two doubles.
      </returns>
      <example>
      <code>
      double c = Math.Add(4.5, 5.4);
      if (c > 10)
      {
        Console.WriteLine(c);
      }
      </code>
      </example>
      <exception cref="System.OverflowException">Thrown when one parameter is max
      and the other is greater than 0.</exception>
      See <see cref="Math.Add(int, int)"/> to add integers.
      <seealso cref="Math.Subtract(double, double)"/>
      <seealso cref="Math.Multiply(double, double)"/>
      <seealso cref="Math.Divide(double, double)"/>
      <param name="a">A double precision number.</param>
      <param name="b">A double precision number.</param>
    </AddDouble>
    <SubtractInt>
```

```
    <summary>
    Subtracts <paramref name="b"/> from <paramref name="a"/> and
    returns the result.
    </summary>
    <returns>
    The difference between two integers.
    </returns>
    <example>
    <code>
    int c = Math.Subtract(4, 5);
    if (c > 1)
    {
      Console.WriteLine(c);
    }
    </code>
    </example>
    See <see cref="Math.Subtract(double, double)"/> to subtract doubles.
    <seealso cref="Math.Add(int, int)"/>
    <seealso cref="Math.Multiply(int, int)"/>
    <seealso cref="Math.Divide(int, int)"/>
    <param name="a">An integer.</param>
    <param name="b">An integer.</param>
  </SubtractInt>
  <SubtractDouble>
    <summary>
    Subtracts a double <paramref name="b"/> from another
    double <paramref name="a"/> and returns the result.
    </summary>
    <returns>
    The difference between two doubles.
    </returns>
    <example>
    <code>
    double c = Math.Subtract(4.5, 5.4);
    if (c > 1)
    {
      Console.WriteLine(c);
    }
    </code>
    </example>
    See <see cref="Math.Subtract(int, int)"/> to subtract integers.
    <seealso cref="Math.Add(double, double)"/>
    <seealso cref="Math.Multiply(double, double)"/>
    <seealso cref="Math.Divide(double, double)"/>
    <param name="a">A double precision number.</param>
    <param name="b">A double precision number.</param>
  </SubtractDouble>
  <MultiplyInt>
    <summary>
    Multiplies two integers <paramref name="a"/> and
    <paramref name="b"/> and returns the result.
    </summary>
    <returns>
    The product of two integers.
    </returns>
    <example>
    <code>
    int c = Math.Multiply(4, 5);
    if (c > 100)
    {
      Console.WriteLine(c);
    }
    </code>
    </example>
    See <see cref="Math.Multiply(double, double)"/> to multiply doubles.
    <seealso cref="Math.Add(int, int)"/>
    <seealso cref="Math.Subtract(int, int)"/>
    <seealso cref="Math.Divide(int, int)"/>
    <param name="a">An integer.</param>
```

```
    <param name="b">An integer.</param>
  </MultiplyInt>
  <MultiplyDouble>
    <summary>
    Multiplies two doubles <paramref name="a"/> and
    <paramref name="b"/> and returns the result.
    </summary>
    <returns>
    The product of two doubles.
    </returns>
    <example>
    <code>
    double c = Math.Multiply(4.5, 5.4);
    if (c > 100.0)
    {
      Console.WriteLine(c);
    }
    </code>
    </example>
    See <see cref="Math.Multiply(int, int)"/> to multiply integers.
    <seealso cref="Math.Add(double, double)"/>
    <seealso cref="Math.Subtract(double, double)"/>
    <seealso cref="Math.Divide(double, double)"/>
    <param name="a">A double precision number.</param>
    <param name="b">A double precision number.</param>
  </MultiplyDouble>
  <DivideInt>
    <summary>
    Divides an integer <paramref name="a"/> by another integer
    <paramref name="b"/> and returns the result.
    </summary>
    <returns>
    The quotient of two integers.
    </returns>
    <example>
    <code>
    int c = Math.Divide(4, 5);
    if (c > 1)
    {
      Console.WriteLine(c);
    }
    </code>
    </example>
    <exception cref="System.DivideByZeroException">
    Thrown when <paramref name="b"/> is equal to 0.
    </exception>
    See <see cref="Math.Divide(double, double)"/> to divide doubles.
    <seealso cref="Math.Add(int, int)"/>
    <seealso cref="Math.Subtract(int, int)"/>
    <seealso cref="Math.Multiply(int, int)"/>
    <param name="a">An integer dividend.</param>
    <param name="b">An integer divisor.</param>
  </DivideInt>
  <DivideDouble>
    <summary>
    Divides a double <paramref name="a"/> by another
    double <paramref name="b"/> and returns the result.
    </summary>
    <returns>
    The quotient of two doubles.
    </returns>
    <example>
    <code>
    double c = Math.Divide(4.5, 5.4);
    if (c > 1.0)
    {
      Console.WriteLine(c);
    }
    </code>
```

```
      </code>
      </example>
      <exception cref="System.DivideByZeroException">Thrown when <paramref name="b"/> is equal to 0.
</exception>
      See <see cref="Math.Divide(int, int)"/> to divide integers.
      <seealso cref="Math.Add(double, double)"/>
      <seealso cref="Math.Subtract(double, double)"/>
      <seealso cref="Math.Multiply(double, double)"/>
      <param name="a">A double precision dividend.</param>
      <param name="b">A double precision divisor.</param>
    </DivideDouble>
  </members>
</docs>
```

В приведенном выше XML комментарии к документации по каждому члену отображаются непосредственно внутри тега с именем, соответствующим его назначению. Можно выбрать собственную стратегию. Код использует тег `<include>` для ссылки на соответствующий элемент в XML-файле:

```
namespace IncludeTag
{

    /*
        The main Math class
        Contains all methods for performing basic math functions
    */
    /// <include file='include.xml' path='docs/members[@name="math"]/Math/*'/>
    public class Math
    {
        // Adds two integers and returns the result
        /// <include file='include.xml' path='docs/members[@name="math"]/AddInt/*'/>
        public static int Add(int a, int b)
        {
            // If any parameter is equal to the max value of an integer
            // and the other is greater than zero
            if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
                throw new System.OverflowException();

            return a + b;
        }

        // Adds two doubles and returns the result
        /// <include file='include.xml' path='docs/members[@name="math"]/AddDouble/*'/>
        public static double Add(double a, double b)
        {
            // If any parameter is equal to the max value of an integer
            // and the other is greater than zero
            if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
                throw new System.OverflowException();

            return a + b;
        }

        // Subtracts an integer from another and returns the result
        /// <include file='include.xml' path='docs/members[@name="math"]/SubtractInt/*'/>
        public static int Subtract(int a, int b)
        {
            return a - b;
        }

        // Subtracts a double from another and returns the result
        /// <include file='include.xml' path='docs/members[@name="math"]/SubtractDouble/*'/>
        public static double Subtract(double a, double b)
        {
            return a - b;
        }

        // Multiplies two integers and returns the result
```

```
        /// <include file='include.xml' path='docs/members[@name="math"]/MultiplyInt/*'/>
        public static int Multiply(int a, int b)
        {
            return a * b;
        }

        // Multiplies two doubles and returns the result
        /// <include file='include.xml' path='docs/members[@name="math"]/MultiplyDouble/*'/>
        public static double Multiply(double a, double b)
        {
            return a * b;
        }

        // Divides an integer by another and returns the result
        /// <include file='include.xml' path='docs/members[@name="math"]/DivideInt/*'/>
        public static int Divide(int a, int b)
        {
            return a / b;
        }

        // Divides a double by another and returns the result
        /// <include file='include.xml' path='docs/members[@name="math"]/DivideDouble/*'/>
        public static double Divide(double a, double b)
        {
            return a / b;
        }
    }
}
```

- Атрибут `file` представляет имя XML-файла, содержащего документацию.
- Атрибут `path` представляет запрос XPath к `tag name`, находящемуся в указанном `file`.
- Атрибут `name` представляет спецификатор имени в теге, который предшествует комментариям.
- Атрибут `id`, который можно использовать вместо `name`, представляет идентификатор для тега, предшествующего комментариям.

Ошибки компилятора C#

18.12.2021 • 2 minutes to read

Для некоторых ошибок компилятора C# существуют отдельные статьи, которые описывают причины этих ошибок, а иногда и методы их устранения. Используйте один из следующих вариантов действий, чтобы проверить наличие справки по конкретному сообщению об ошибке.

- Если вы используете Visual Studio, выберите номер ошибки (например, CS0029) в окне вывода и нажмите клавишу F1.
- Введите номер ошибки в поле *Фильтр по названию* в содержании.

Если вы все равно не можете найти информацию об ошибке, перейдите в конец этой страницы и отправьте отзыв, указав номер или текст ошибки.

Сведения о настройке ошибок и предупреждений в C# см. в описании параметров компилятора C# или страницы "Сборка" Visual Studio в конструкторе проектов (C#).

> **NOTE**
> Отображаемые на компьютере имена или расположения некоторых элементов пользовательского интерфейса Visual Studio могут отличаться от указанных в следующих инструкциях. Это зависит от имеющегося выпуска Visual Studio и используемых параметров. Дополнительные сведения см. в разделе Персонализация среды IDE.

См. также раздел

- Параметры компилятора C#
- Страница "Сборка" в конструкторе проектов (C#)
- **WarningLevel** (параметры компилятора C#)
- **DisabledWarnings** (параметры компилятора C#)

Introduction

18.12.2021 • 55 minutes to read

C# (pronounced "See Sharp") is a simple, modern, object-oriented, and type-safe programming language. C# has its roots in the C family of languages and will be immediately familiar to C, C++, and Java programmers. C# is standardized by ECMA International as the *ECMA-334* standard and by ISO/IEC as the *ISO/IEC 23270* standard. Microsoft's C# compiler for the .NET Framework is a conforming implementation of both of these standards.

C# is an object-oriented language, but C# further includes support for ***component-oriented*** programming. Contemporary software design increasingly relies on software components in the form of self-contained and self-describing packages of functionality. Key to such components is that they present a programming model with properties, methods, and events; they have attributes that provide declarative information about the component; and they incorporate their own documentation. C# provides language constructs to directly support these concepts, making C# a very natural language in which to create and use software components.

Several C# features aid in the construction of robust and durable applications: ***Garbage collection*** automatically reclaims memory occupied by unused objects; ***exception handling*** provides a structured and extensible approach to error detection and recovery; and the ***type-safe*** design of the language makes it impossible to read from uninitialized variables, to index arrays beyond their bounds, or to perform unchecked type casts.

C# has a ***unified type system***. All C# types, including primitive types such as `int` and `double`, inherit from a single root `object` type. Thus, all types share a set of common operations, and values of any type can be stored, transported, and operated upon in a consistent manner. Furthermore, C# supports both user-defined reference types and value types, allowing dynamic allocation of objects as well as in-line storage of lightweight structures.

To ensure that C# programs and libraries can evolve over time in a compatible manner, much emphasis has been placed on ***versioning*** in C#'s design. Many programming languages pay little attention to this issue, and, as a result, programs written in those languages break more often than necessary when newer versions of dependent libraries are introduced. Aspects of C#'s design that were directly influenced by versioning considerations include the separate `virtual` and `override` modifiers, the rules for method overload resolution, and support for explicit interface member declarations.

The rest of this chapter describes the essential features of the C# language. Although later chapters describe rules and exceptions in a detail-oriented and sometimes mathematical manner, this chapter strives for clarity and brevity at the expense of completeness. The intent is to provide the reader with an introduction to the language that will facilitate the writing of early programs and the reading of later chapters.

Hello world

The "Hello, World" program is traditionally used to introduce a programming language. Here it is in C#:

```
using System;

class Hello
{
    static void Main() {
        Console.WriteLine("Hello, World");
    }
}
```

C# source files typically have the file extension `.cs`. Assuming that the "Hello, World" program is stored in the file `hello.cs`, the program can be compiled with the Microsoft C# compiler using the command line

```
csc hello.cs
```

which produces an executable assembly named `hello.exe`. The output produced by this application when it is run is

```
Hello, World
```

The "Hello, World" program starts with a `using` directive that references the `System` namespace. Namespaces provide a hierarchical means of organizing C# programs and libraries. Namespaces contain types and other namespaces—for example, the `System` namespace contains a number of types, such as the `Console` class referenced in the program, and a number of other namespaces, such as `IO` and `Collections`. A `using` directive that references a given namespace enables unqualified use of the types that are members of that namespace. Because of the `using` directive, the program can use `Console.WriteLine` as shorthand for `System.Console.WriteLine`.

The `Hello` class declared by the "Hello, World" program has a single member, the method named `Main`. The `Main` method is declared with the `static` modifier. While instance methods can reference a particular enclosing object instance using the keyword `this`, static methods operate without reference to a particular object. By convention, a static method named `Main` serves as the entry point of a program.

The output of the program is produced by the `WriteLine` method of the `Console` class in the `System` namespace. This class is provided by the .NET Framework class libraries, which, by default, are automatically referenced by the Microsoft C# compiler. Note that C# itself does not have a separate runtime library. Instead, the .NET Framework is the runtime library of C#.

Program structure

The key organizational concepts in C# are ***programs***, ***namespaces***, ***types***, ***members***, and ***assemblies***. C# programs consist of one or more source files. Programs declare types, which contain members and can be organized into namespaces. Classes and interfaces are examples of types. Fields, methods, properties, and events are examples of members. When C# programs are compiled, they are physically packaged into assemblies. Assemblies typically have the file extension `.exe` or `.dll`, depending on whether they implement ***applications*** or ***libraries***.

The example

```
using System;

namespace Acme.Collections
{
    public class Stack
    {
        Entry top;

        public void Push(object data) {
            top = new Entry(top, data);
        }

        public object Pop() {
            if (top == null) throw new InvalidOperationException();
            object result = top.data;
            top = top.next;
            return result;
        }

        class Entry
        {
            public Entry next;
            public object data;

            public Entry(Entry next, object data) {
                this.next = next;
                this.data = data;
            }
        }
    }
}
```

declares a class named `Stack` in a namespace called `Acme.Collections`. The fully qualified name of this class is `Acme.Collections.Stack`. The class contains several members: a field named `top`, two methods named `Push` and `Pop`, and a nested class named `Entry`. The `Entry` class further contains three members: a field named `next`, a field named `data`, and a constructor. Assuming that the source code of the example is stored in the file `acme.cs`, the command line

```
csc /t:library acme.cs
```

compiles the example as a library (code without a `Main` entry point) and produces an assembly named `acme.dll`.

Assemblies contain executable code in the form of ***Intermediate Language*** (IL) instructions, and symbolic information in the form of ***metadata***. Before it is executed, the IL code in an assembly is automatically converted to processor-specific code by the Just-In-Time (JIT) compiler of .NET Common Language Runtime.

Because an assembly is a self-describing unit of functionality containing both code and metadata, there is no need for `#include` directives and header files in C#. The public types and members contained in a particular assembly are made available in a C# program simply by referencing that assembly when compiling the program. For example, this program uses the `Acme.Collections.Stack` class from the `acme.dll` assembly:

```
using System;
using Acme.Collections;

class Test
{
    static void Main() {
        Stack s = new Stack();
        s.Push(1);
        s.Push(10);
        s.Push(100);
        Console.WriteLine(s.Pop());
        Console.WriteLine(s.Pop());
        Console.WriteLine(s.Pop());
    }
}
```

If the program is stored in the file `test.cs`, when `test.cs` is compiled, the `acme.dll` assembly can be referenced using the compiler's `/r` option:

```
csc /r:acme.dll test.cs
```

This creates an executable assembly named `test.exe`, which, when run, produces the output:

```
100
10
1
```

C# permits the source text of a program to be stored in several source files. When a multi-file C# program is compiled, all of the source files are processed together, and the source files can freely reference each other—conceptually, it is as if all the source files were concatenated into one large file before being processed. Forward declarations are never needed in C# because, with very few exceptions, declaration order is insignificant. C# does not limit a source file to declaring only one public type nor does it require the name of the source file to match a type declared in the source file.

Types and variables

There are two kinds of types in C#: ***value types*** and ***reference types***. Variables of value types directly contain their data whereas variables of reference types store references to their data, the latter being known as objects. With reference types, it is possible for two variables to reference the same object and thus possible for operations on one variable to affect the object referenced by the other variable. With value types, the variables each have their own copy of the data, and it is not possible for operations on one to affect the other (except in the case of `ref` and `out` parameter variables).

C#'s value types are further divided into ***simple types***, ***enum types***, ***struct types***, and ***nullable types***, and C#'s reference types are further divided into ***class types***, ***interface types***, ***array types***, and ***delegate types***.

The following table provides an overview of C#'s type system.

CATEGORY	TYPES	DESCRIPTION
Value types	Simple types	Signed integral: `sbyte`, `short`, `int`, `long`
		Unsigned integral: `byte`, `ushort`, `uint`, `ulong`

CATEGORY	TYPES	DESCRIPTION
		Unicode characters: `char`
		IEEE floating point: `float`, `double`
		High-precision decimal: `decimal`
		Boolean: `bool`
	Enum types	User-defined types of the form `enum E {...}`
	Struct types	User-defined types of the form `struct S {...}`
	Nullable types	Extensions of all other value types with a `null` value
Reference types	Class types	Ultimate base class of all other types: `object`
		Unicode strings: `string`
		User-defined types of the form `class C {...}`
	Interface types	User-defined types of the form `interface I {...}`
	Array types	Single- and multi-dimensional, for example, `int[]` and `int[,]`
	Delegate types	User-defined types of the form e.g. `delegate int D(...)`

The eight integral types provide support for 8-bit, 16-bit, 32-bit, and 64-bit values in signed or unsigned form.

The two floating point types, `float` and `double`, are represented using the 32-bit single-precision and 64-bit double-precision IEEE 754 formats.

The `decimal` type is a 128-bit data type suitable for financial and monetary calculations.

C#'s `bool` type is used to represent boolean values—values that are either `true` or `false`.

Character and string processing in C# uses Unicode encoding. The `char` type represents a UTF-16 code unit, and the `string` type represents a sequence of UTF-16 code units.

The following table summarizes C#'s numeric types.

CATEGORY	BITS	TYPE	RANGE/PRECISION
Signed integral	8	`sbyte`	-128...127

CATEGORY	BITS	TYPE	RANGE/PRECISION
	16	`short`	-32,768...32,767
	32	`int`	-2,147,483,648...2,147,483,647
	64	`long`	-9,223,372,036,854,775,808...9,223,372,036,854,775,807
Unsigned integral	8	`byte`	0...255
	16	`ushort`	0...65,535
	32	`uint`	0...4,294,967,295
	64	`ulong`	0...18,446,744,073,709,551,615
Floating point	32	`float`	1.5 × 10^−45 to 3.4 × 10^38, 7-digit precision
	64	`double`	5.0 × 10^−324 to 1.7 × 10^308, 15-digit precision
Decimal	128	`decimal`	1.0 × 10^−28 to 7.9 × 10^28, 28-digit precision

C# programs use ***type declarations*** to create new types. A type declaration specifies the name and the members of the new type. Five of C#'s categories of types are user-definable: class types, struct types, interface types, enum types, and delegate types.

A class type defines a data structure that contains data members (fields) and function members (methods, properties, and others). Class types support single inheritance and polymorphism, mechanisms whereby derived classes can extend and specialize base classes.

A struct type is similar to a class type in that it represents a structure with data members and function members. However, unlike classes, structs are value types and do not require heap allocation. Struct types do not support user-specified inheritance, and all struct types implicitly inherit from type `object`.

An interface type defines a contract as a named set of public function members. A class or struct that implements an interface must provide implementations of the interface's function members. An interface may inherit from multiple base interfaces, and a class or struct may implement multiple interfaces.

A delegate type represents references to methods with a particular parameter list and return type. Delegates make it possible to treat methods as entities that can be assigned to variables and passed as parameters. Delegates are similar to the concept of function pointers found in some other languages, but unlike function pointers, delegates are object-oriented and type-safe.

Class, struct, interface and delegate types all support generics, whereby they can be parameterized with other

types.

An enum type is a distinct type with named constants. Every enum type has an underlying type, which must be one of the eight integral types. The set of values of an enum type is the same as the set of values of the underlying type.

C# supports single- and multi-dimensional arrays of any type. Unlike the types listed above, array types do not have to be declared before they can be used. Instead, array types are constructed by following a type name with square brackets. For example, `int[]` is a single-dimensional array of `int`, `int[,]` is a two-dimensional array of `int`, and `int[][]` is a single-dimensional array of single-dimensional arrays of `int`.

Nullable types also do not have to be declared before they can be used. For each non-nullable value type `T` there is a corresponding nullable type `T?`, which can hold an additional value `null`. For instance, `int?` is a type that can hold any 32 bit integer or the value `null`.

C#'s type system is unified such that a value of any type can be treated as an object. Every type in C# directly or indirectly derives from the `object` class type, and `object` is the ultimate base class of all types. Values of reference types are treated as objects simply by viewing the values as type `object`. Values of value types are treated as objects by performing ***boxing*** and ***unboxing*** operations. In the following example, an `int` value is converted to `object` and back again to `int`.

```
using System;

class Test
{
    static void Main() {
        int i = 123;
        object o = i;          // Boxing
        int j = (int)o;        // Unboxing
    }
}
```

When a value of a value type is converted to type `object`, an object instance, also called a "box," is allocated to hold the value, and the value is copied into that box. Conversely, when an `object` reference is cast to a value type, a check is made that the referenced object is a box of the correct value type, and, if the check succeeds, the value in the box is copied out.

C#'s unified type system effectively means that value types can become objects "on demand." Because of the unification, general-purpose libraries that use type `object` can be used with both reference types and value types.

There are several kinds of ***variables*** in C#, including fields, array elements, local variables, and parameters. Variables represent storage locations, and every variable has a type that determines what values can be stored in the variable, as shown by the following table.

TYPE OF VARIABLE	POSSIBLE CONTENTS
Non-nullable value type	A value of that exact type
Nullable value type	A null value or a value of that exact type
`object`	A null reference, a reference to an object of any reference type, or a reference to a boxed value of any value type

TYPE OF VARIABLE	POSSIBLE CONTENTS
Class type	A null reference, a reference to an instance of that class type, or a reference to an instance of a class derived from that class type
Interface type	A null reference, a reference to an instance of a class type that implements that interface type, or a reference to a boxed value of a value type that implements that interface type
Array type	A null reference, a reference to an instance of that array type, or a reference to an instance of a compatible array type
Delegate type	A null reference or a reference to an instance of that delegate type

Expressions

Expressions are constructed from ***operands*** and ***operators***. The operators of an expression indicate which operations to apply to the operands. Examples of operators include `+`, `-`, `*`, `/`, and `new`. Examples of operands include literals, fields, local variables, and expressions.

When an expression contains multiple operators, the ***precedence*** of the operators controls the order in which the individual operators are evaluated. For example, the expression `x + y * z` is evaluated as `x + (y * z)` because the `*` operator has higher precedence than the `+` operator.

Most operators can be ***overloaded***. Operator overloading permits user-defined operator implementations to be specified for operations where one or both of the operands are of a user-defined class or struct type.

The following table summarizes C#'s operators, listing the operator categories in order of precedence from highest to lowest. Operators in the same category have equal precedence.

CATEGORY	EXPRESSION	DESCRIPTION
Primary	`x.m`	Member access
	`x(...)`	Method and delegate invocation
	`x[...]`	Array and indexer access
	`x++`	Post-increment
	`x--`	Post-decrement
	`new T(...)`	Object and delegate creation
	`new T(...){...}`	Object creation with initializer
	`new {...}`	Anonymous object initializer
	`new T[...]`	Array creation

CATEGORY	EXPRESSION	DESCRIPTION
	`typeof(T)`	Obtain `System.Type` object for `T`
	`checked(x)`	Evaluate expression in checked context
	`unchecked(x)`	Evaluate expression in unchecked context
	`default(T)`	Obtain default value of type `T`
	`delegate {...}`	Anonymous function (anonymous method)
Unary	`+x`	Identity
	`-x`	Negation
	`!x`	Logical negation
	`~x`	Bitwise negation
	`++x`	Pre-increment
	`--x`	Pre-decrement
	`(T)x`	Explicitly convert `x` to type `T`
	`await x`	Asynchronously wait for `x` to complete
Multiplicative	`x * y`	Multiplication
	`x / y`	Division
	`x % y`	Remainder
Additive	`x + y`	Addition, string concatenation, delegate combination
	`x - y`	Subtraction, delegate removal
Shift	`x << y`	Shift left
	`x >> y`	Shift right
Relational and type testing	`x < y`	Less than
	`x > y`	Greater than
	`x <= y`	Less than or equal

CATEGORY	EXPRESSION	DESCRIPTION
	`x >= y`	Greater than or equal
	`x is T`	Return `true` if `x` is a `T`, `false` otherwise
	`x as T`	Return `x` typed as `T`, or `null` if `x` is not a `T`
Equality	`x == y`	Equal
	`x != y`	Not equal
Logical AND	`x & y`	Integer bitwise AND, boolean logical AND
Logical XOR	`x ^ y`	Integer bitwise XOR, boolean logical XOR
Logical OR	`x | y`	Integer bitwise OR, boolean logical OR
Conditional AND	`x && y`	Evaluates `y` only if `x` is `true`
Conditional OR	`x || y`	Evaluates `y` only if `x` is `false`
Null coalescing	`x ?? y`	Evaluates to `y` if `x` is `null`, to `x` otherwise
Conditional	`x ? y : z`	Evaluates `y` if `x` is `true`, `z` if `x` is `false`
Assignment or anonymous function	`x = y`	Assignment
	`x op= y`	Compound assignment; supported operators are `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=`
	`(T x) => y`	Anonymous function (lambda expression)

Statements

The actions of a program are expressed using ***statements***. C# supports several different kinds of statements, a number of which are defined in terms of embedded statements.

A ***block*** permits multiple statements to be written in contexts where a single statement is allowed. A block consists of a list of statements written between the delimiters `{` and `}`.

Declaration statements are used to declare local variables and constants.

Expression statements are used to evaluate expressions. Expressions that can be used as statements include method invocations, object allocations using the `new` operator, assignments using `=` and the compound assignment operators, increment and decrement operations using the `++` and `--` operators and await

expressions.

Selection statements are used to select one of a number of possible statements for execution based on the value of some expression. In this group are the `if` and `switch` statements.

Iteration statements are used to repeatedly execute an embedded statement. In this group are the `while`, `do`, `for`, and `foreach` statements.

Jump statements are used to transfer control. In this group are the `break`, `continue`, `goto`, `throw`, `return`, and `yield` statements.

The `try` ... `catch` statement is used to catch exceptions that occur during execution of a block, and the `try` ... `finally` statement is used to specify finalization code that is always executed, whether an exception occurred or not.

The `checked` and `unchecked` statements are used to control the overflow checking context for integral-type arithmetic operations and conversions.

The `lock` statement is used to obtain the mutual-exclusion lock for a given object, execute a statement, and then release the lock.

The `using` statement is used to obtain a resource, execute a statement, and then dispose of that resource.

Below are examples of each kind of statement

Local variable declarations

```
static void Main() {
   int a;
   int b = 2, c = 3;
   a = 1;
   Console.WriteLine(a + b + c);
}
```

Local constant declaration

```
static void Main() {
    const float pi = 3.1415927f;
    const int r = 25;
    Console.WriteLine(pi * r * r);
}
```

Expression statement

```
static void Main() {
    int i;
    i = 123;                // Expression statement
    Console.WriteLine(i);   // Expression statement
    i++;                    // Expression statement
    Console.WriteLine(i);   // Expression statement
}
```

`if` statement

```
static void Main(string[] args) {
    if (args.Length == 0) {
        Console.WriteLine("No arguments");
    }
    else {
        Console.WriteLine("One or more arguments");
    }
}
```

`switch` statement

```
static void Main(string[] args) {
    int n = args.Length;
    switch (n) {
        case 0:
            Console.WriteLine("No arguments");
            break;
        case 1:
            Console.WriteLine("One argument");
            break;
        default:
            Console.WriteLine("{0} arguments", n);
            break;
    }
}
```

`while` statement

```
static void Main(string[] args) {
    int i = 0;
    while (i < args.Length) {
        Console.WriteLine(args[i]);
        i++;
    }
}
```

`do` statement

```
static void Main() {
    string s;
    do {
        s = Console.ReadLine();
        if (s != null) Console.WriteLine(s);
    } while (s != null);
}
```

`for` statement

```
static void Main(string[] args) {
    for (int i = 0; i < args.Length; i++) {
        Console.WriteLine(args[i]);
    }
}
```

`foreach` statement

```
static void Main(string[] args) {
    foreach (string s in args) {
        Console.WriteLine(s);
    }
}
```

`break` statement

```
static void Main() {
    while (true) {
        string s = Console.ReadLine();
        if (s == null) break;
        Console.WriteLine(s);
    }
}
```

`continue` statement

```
static void Main(string[] args) {
    for (int i = 0; i < args.Length; i++) {
        if (args[i].StartsWith("/")) continue;
        Console.WriteLine(args[i]);
    }
}
```

`goto` statement

```
static void Main(string[] args) {
    int i = 0;
    goto check;
    loop:
    Console.WriteLine(args[i++]);
    check:
    if (i < args.Length) goto loop;
}
```

`return` statement

```
static int Add(int a, int b) {
    return a + b;
}

static void Main() {
    Console.WriteLine(Add(1, 2));
    return;
}
```

`yield` statement

```
static IEnumerable<int> Range(int from, int to) {
    for (int i = from; i < to; i++) {
        yield return i;
    }
    yield break;
}

static void Main() {
    foreach (int x in Range(-10,10)) {
        Console.WriteLine(x);
    }
}
```

`throw` and `try` statements

```
static double Divide(double x, double y) {
    if (y == 0) throw new DivideByZeroException();
    return x / y;
}

static void Main(string[] args) {
    try {
        if (args.Length != 2) {
            throw new Exception("Two numbers required");
        }
        double x = double.Parse(args[0]);
        double y = double.Parse(args[1]);
        Console.WriteLine(Divide(x, y));
    }
    catch (Exception e) {
        Console.WriteLine(e.Message);
    }
    finally {
        Console.WriteLine("Good bye!");
    }
}
```

`checked` and `unchecked` statements

```
static void Main() {
    int i = int.MaxValue;
    checked {
        Console.WriteLine(i + 1);        // Exception
    }
    unchecked {
        Console.WriteLine(i + 1);        // Overflow
    }
}
```

`lock` statement

```
class Account
{
    decimal balance;
    public void Withdraw(decimal amount) {
        lock (this) {
            if (amount > balance) {
                throw new Exception("Insufficient funds");
            }
            balance -= amount;
        }
    }
}
```

`using` statement

```
static void Main() {
    using (TextWriter w = File.CreateText("test.txt")) {
        w.WriteLine("Line one");
        w.WriteLine("Line two");
        w.WriteLine("Line three");
    }
}
```

Classes and objects

Classes are the most fundamental of C#'s types. A class is a data structure that combines state (fields) and actions (methods and other function members) in a single unit. A class provides a definition for dynamically created ***instances*** of the class, also known as ***objects***. Classes support ***inheritance*** and ***polymorphism***, mechanisms whereby ***derived classes*** can extend and specialize ***base classes***.

New classes are created using class declarations. A class declaration starts with a header that specifies the attributes and modifiers of the class, the name of the class, the base class (if given), and the interfaces implemented by the class. The header is followed by the class body, which consists of a list of member declarations written between the delimiters `{` and `}`.

The following is a declaration of a simple class named `Point`:

```
public class Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Instances of classes are created using the `new` operator, which allocates memory for a new instance, invokes a constructor to initialize the instance, and returns a reference to the instance. The following statements create two `Point` objects and store references to those objects in two variables:

```
Point p1 = new Point(0, 0);
Point p2 = new Point(10, 20);
```

The memory occupied by an object is automatically reclaimed when the object is no longer in use. It is neither necessary nor possible to explicitly deallocate objects in C#.

Members

The members of a class are either *static members* or *instance members*. Static members belong to classes, and instance members belong to objects (instances of classes).

The following table provides an overview of the kinds of members a class can contain.

MEMBER	DESCRIPTION
Constants	Constant values associated with the class
Fields	Variables of the class
Methods	Computations and actions that can be performed by the class
Properties	Actions associated with reading and writing named properties of the class
Indexers	Actions associated with indexing instances of the class like an array
Events	Notifications that can be generated by the class
Operators	Conversions and expression operators supported by the class
Constructors	Actions required to initialize instances of the class or the class itself
Destructors	Actions to perform before instances of the class are permanently discarded
Types	Nested types declared by the class

Accessibility

Each member of a class has an associated accessibility, which controls the regions of program text that are able to access the member. There are five possible forms of accessibility. These are summarized in the following table.

ACCESSIBILITY	MEANING
`public`	Access not limited
`protected`	Access limited to this class or classes derived from this class
`internal`	Access limited to this program
`protected internal`	Access limited to this program or classes derived from this class
`private`	Access limited to this class

Type parameters

A class definition may specify a set of type parameters by following the class name with angle brackets enclosing a list of type parameter names. The type parameters can then be used in the body of the class

declarations to define the members of the class. In the following example, the type parameters of `Pair` are `TFirst` and `TSecond`:

```
public class Pair<TFirst,TSecond>
{
    public TFirst First;
    public TSecond Second;
}
```

A class type that is declared to take type parameters is called a generic class type. Struct, interface and delegate types can also be generic.

When the generic class is used, type arguments must be provided for each of the type parameters:

```
Pair<int,string> pair = new Pair<int,string> { First = 1, Second = "two" };
int i = pair.First;     // TFirst is int
string s = pair.Second; // TSecond is string
```

A generic type with type arguments provided, like `Pair<int,string>` above, is called a constructed type.

Base classes

A class declaration may specify a base class by following the class name and type parameters with a colon and the name of the base class. Omitting a base class specification is the same as deriving from type `object`. In the following example, the base class of `Point3D` is `Point`, and the base class of `Point` is `object`:

```
public class Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

public class Point3D: Point
{
    public int z;

    public Point3D(int x, int y, int z): base(x, y) {
        this.z = z;
    }
}
```

A class inherits the members of its base class. Inheritance means that a class implicitly contains all members of its base class, except for the instance and static constructors, and the destructors of the base class. A derived class can add new members to those it inherits, but it cannot remove the definition of an inherited member. In the previous example, `Point3D` inherits the `x` and `y` fields from `Point`, and every `Point3D` instance contains three fields, `x`, `y`, and `z`.

An implicit conversion exists from a class type to any of its base class types. Therefore, a variable of a class type can reference an instance of that class or an instance of any derived class. For example, given the previous class declarations, a variable of type `Point` can reference either a `Point` or a `Point3D`:

```
Point a = new Point(10, 20);
Point b = new Point3D(10, 20, 30);
```

Fields

A field is a variable that is associated with a class or with an instance of a class.

A field declared with the `static` modifier defines a ***static field***. A static field identifies exactly one storage location. No matter how many instances of a class are created, there is only ever one copy of a static field.

A field declared without the `static` modifier defines an ***instance field***. Every instance of a class contains a separate copy of all the instance fields of that class.

In the following example, each instance of the `Color` class has a separate copy of the `r`, `g`, and `b` instance fields, but there is only one copy of the `Black`, `White`, `Red`, `Green`, and `Blue` static fields:

```
public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);
    private byte r, g, b;

    public Color(byte r, byte g, byte b) {
        this.r = r;
        this.g = g;
        this.b = b;
    }
}
```

As shown in the previous example, ***read-only fields*** may be declared with a `readonly` modifier. Assignment to a `readonly` field can only occur as part of the field's declaration or in a constructor in the same class.

Methods

A ***method*** is a member that implements a computation or action that can be performed by an object or class. ***Static methods*** are accessed through the class. ***Instance methods*** are accessed through instances of the class.

Methods have a (possibly empty) list of ***parameters***, which represent values or variable references passed to the method, and a ***return type***, which specifies the type of the value computed and returned by the method. A method's return type is `void` if it does not return a value.

Like types, methods may also have a set of type parameters, for which type arguments must be specified when the method is called. Unlike types, the type arguments can often be inferred from the arguments of a method call and need not be explicitly given.

The ***signature*** of a method must be unique in the class in which the method is declared. The signature of a method consists of the name of the method, the number of type parameters and the number, modifiers, and types of its parameters. The signature of a method does not include the return type.

Parameters

Parameters are used to pass values or variable references to methods. The parameters of a method get their actual values from the ***arguments*** that are specified when the method is invoked. There are four kinds of parameters: value parameters, reference parameters, output parameters, and parameter arrays.

A ***value parameter*** is used for input parameter passing. A value parameter corresponds to a local variable that gets its initial value from the argument that was passed for the parameter. Modifications to a value parameter do not affect the argument that was passed for the parameter.

Value parameters can be optional, by specifying a default value so that corresponding arguments can be omitted.

A ***reference parameter*** is used for both input and output parameter passing. The argument passed for a reference parameter must be a variable, and during execution of the method, the reference parameter represents the same storage location as the argument variable. A reference parameter is declared with the `ref` modifier. The following example shows the use of `ref` parameters.

```
using System;

class Test
{
    static void Swap(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
    }

    static void Main() {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine("{0} {1}", i, j);            // Outputs "2 1"
    }
}
```

An ***output parameter*** is used for output parameter passing. An output parameter is similar to a reference parameter except that the initial value of the caller-provided argument is unimportant. An output parameter is declared with the `out` modifier. The following example shows the use of `out` parameters.

```
using System;

class Test
{
    static void Divide(int x, int y, out int result, out int remainder) {
        result = x / y;
        remainder = x % y;
    }

    static void Main() {
        int res, rem;
        Divide(10, 3, out res, out rem);
        Console.WriteLine("{0} {1}", res, rem);    // Outputs "3 1"
    }
}
```

A ***parameter array*** permits a variable number of arguments to be passed to a method. A parameter array is declared with the `params` modifier. Only the last parameter of a method can be a parameter array, and the type of a parameter array must be a single-dimensional array type. The `Write` and `WriteLine` methods of the `System.Console` class are good examples of parameter array usage. They are declared as follows.

```
public class Console
{
    public static void Write(string fmt, params object[] args) {...}
    public static void WriteLine(string fmt, params object[] args) {...}
    ...
}
```

Within a method that uses a parameter array, the parameter array behaves exactly like a regular parameter of an array type. However, in an invocation of a method with a parameter array, it is possible to pass either a single argument of the parameter array type or any number of arguments of the element type of the parameter array. In the latter case, an array instance is automatically created and initialized with the given arguments. This example

```
Console.WriteLine("x={0} y={1} z={2}", x, y, z);
```

is equivalent to writing the following.

```
string s = "x={0} y={1} z={2}";
object[] args = new object[3];
args[0] = x;
args[1] = y;
args[2] = z;
Console.WriteLine(s, args);
```

Method body and local variables

A method's body specifies the statements to execute when the method is invoked.

A method body can declare variables that are specific to the invocation of the method. Such variables are called ***local variables***. A local variable declaration specifies a type name, a variable name, and possibly an initial value. The following example declares a local variable `i` with an initial value of zero and a local variable `j` with no initial value.

```
using System;

class Squares
{
    static void Main() {
        int i = 0;
        int j;
        while (i < 10) {
            j = i * i;
            Console.WriteLine("{0} x {0} = {1}", i, j);
            i = i + 1;
        }
    }
}
```

C# requires a local variable to be ***definitely assigned*** before its value can be obtained. For example, if the declaration of the previous `i` did not include an initial value, the compiler would report an error for the subsequent usages of `i` because `i` would not be definitely assigned at those points in the program.

A method can use `return` statements to return control to its caller. In a method returning `void`, `return` statements cannot specify an expression. In a method returning non-`void`, `return` statements must include an expression that computes the return value.

Static and instance methods

A method declared with a `static` modifier is a ***static method***. A static method does not operate on a specific instance and can only directly access static members.

A method declared without a `static` modifier is an ***instance method***. An instance method operates on a specific instance and can access both static and instance members. The instance on which an instance method was invoked can be explicitly accessed as `this`. It is an error to refer to `this` in a static method.

The following `Entity` class has both static and instance members.

```
class Entity
{
    static int nextSerialNo;
    int serialNo;

    public Entity() {
        serialNo = nextSerialNo++;
    }

    public int GetSerialNo() {
        return serialNo;
    }

    public static int GetNextSerialNo() {
        return nextSerialNo;
    }

    public static void SetNextSerialNo(int value) {
        nextSerialNo = value;
    }
}
```

Each `Entity` instance contains a serial number (and presumably some other information that is not shown here). The `Entity` constructor (which is like an instance method) initializes the new instance with the next available serial number. Because the constructor is an instance member, it is permitted to access both the `serialNo` instance field and the `nextSerialNo` static field.

The `GetNextSerialNo` and `SetNextSerialNo` static methods can access the `nextSerialNo` static field, but it would be an error for them to directly access the `serialNo` instance field.

The following example shows the use of the `Entity` class.

```
using System;

class Test
{
    static void Main() {
        Entity.SetNextSerialNo(1000);
        Entity e1 = new Entity();
        Entity e2 = new Entity();
        Console.WriteLine(e1.GetSerialNo());           // Outputs "1000"
        Console.WriteLine(e2.GetSerialNo());           // Outputs "1001"
        Console.WriteLine(Entity.GetNextSerialNo());   // Outputs "1002"
    }
}
```

Note that the `SetNextSerialNo` and `GetNextSerialNo` static methods are invoked on the class whereas the `GetSerialNo` instance method is invoked on instances of the class.

Virtual, override, and abstract methods

When an instance method declaration includes a `virtual` modifier, the method is said to be a ***virtual method***. When no `virtual` modifier is present, the method is said to be a ***non-virtual method***.

When a virtual method is invoked, the ***run-time type*** of the instance for which that invocation takes place determines the actual method implementation to invoke. In a nonvirtual method invocation, the ***compile-time type*** of the instance is the determining factor.

A virtual method can be ***overridden*** in a derived class. When an instance method declaration includes an `override` modifier, the method overrides an inherited virtual method with the same signature. Whereas a virtual method declaration introduces a new method, an override method declaration specializes an existing

inherited virtual method by providing a new implementation of that method.

An ***abstract*** method is a virtual method with no implementation. An abstract method is declared with the `abstract` modifier and is permitted only in a class that is also declared `abstract`. An abstract method must be overridden in every non-abstract derived class.

The following example declares an abstract class, `Expression`, which represents an expression tree node, and three derived classes, `Constant`, `VariableReference`, and `Operation`, which implement expression tree nodes for constants, variable references, and arithmetic operations. (This is similar to, but not to be confused with the expression tree types introduced in Expression tree types).

```
using System;
using System.Collections;

public abstract class Expression
{
    public abstract double Evaluate(Hashtable vars);
}

public class Constant: Expression
{
    double value;

    public Constant(double value) {
        this.value = value;
    }

    public override double Evaluate(Hashtable vars) {
        return value;
    }
}

public class VariableReference: Expression
{
    string name;

    public VariableReference(string name) {
        this.name = name;
    }

    public override double Evaluate(Hashtable vars) {
        object value = vars[name];
        if (value == null) {
            throw new Exception("Unknown variable: " + name);
        }
        return Convert.ToDouble(value);
    }
}

public class Operation: Expression
{
    Expression left;
    char op;
    Expression right;

    public Operation(Expression left, char op, Expression right) {
        this.left = left;
        this.op = op;
        this.right = right;
    }

    public override double Evaluate(Hashtable vars) {
        double x = left.Evaluate(vars);
        double y = right.Evaluate(vars);
        switch (op) {
            case '+': return x + y;
            case '-': return x - y;
            case '*': return x * y;
            case '/': return x / y;
        }
        throw new Exception("Unknown operator");
    }
}
```

The previous four classes can be used to model arithmetic expressions. For example, using instances of these classes, the expression `x + 3` can be represented as follows.

```
Expression e = new Operation(
    new VariableReference("x"),
    '+',
    new Constant(3));
```

The `Evaluate` method of an `Expression` instance is invoked to evaluate the given expression and produce a `double` value. The method takes as an argument a `Hashtable` that contains variable names (as keys of the entries) and values (as values of the entries). The `Evaluate` method is a virtual abstract method, meaning that non-abstract derived classes must override it to provide an actual implementation.

A `Constant`'s implementation of `Evaluate` simply returns the stored constant. A `VariableReference`'s implementation looks up the variable name in the hashtable and returns the resulting value. An `Operation`'s implementation first evaluates the left and right operands (by recursively invoking their `Evaluate` methods) and then performs the given arithmetic operation.

The following program uses the `Expression` classes to evaluate the expression `x * (y + 2)` for different values of `x` and `y`.

```
using System;
using System.Collections;

class Test
{
    static void Main() {
        Expression e = new Operation(
            new VariableReference("x"),
            '*',
            new Operation(
                new VariableReference("y"),
                '+',
                new Constant(2)
            )
        );
        Hashtable vars = new Hashtable();
        vars["x"] = 3;
        vars["y"] = 5;
        Console.WriteLine(e.Evaluate(vars));        // Outputs "21"
        vars["x"] = 1.5;
        vars["y"] = 9;
        Console.WriteLine(e.Evaluate(vars));        // Outputs "16.5"
    }
}
```

Method overloading

Method ***overloading*** permits multiple methods in the same class to have the same name as long as they have unique signatures. When compiling an invocation of an overloaded method, the compiler uses ***overload resolution*** to determine the specific method to invoke. Overload resolution finds the one method that best matches the arguments or reports an error if no single best match can be found. The following example shows overload resolution in effect. The comment for each invocation in the `Main` method shows which method is actually invoked.

```
class Test
{
    static void F() {
        Console.WriteLine("F()");
    }

    static void F(object x) {
        Console.WriteLine("F(object)");
    }

    static void F(int x) {
        Console.WriteLine("F(int)");
    }

    static void F(double x) {
        Console.WriteLine("F(double)");
    }

    static void F<T>(T x) {
        Console.WriteLine("F<T>(T)");
    }

    static void F(double x, double y) {
        Console.WriteLine("F(double, double)");
    }

    static void Main() {
        F();                // Invokes F()
        F(1);               // Invokes F(int)
        F(1.0);             // Invokes F(double)
        F("abc");           // Invokes F(object)
        F((double)1);       // Invokes F(double)
        F((object)1);       // Invokes F(object)
        F<int>(1);          // Invokes F<T>(T)
        F(1, 1);            // Invokes F(double, double)
    }
}
```

As shown by the example, a particular method can always be selected by explicitly casting the arguments to the exact parameter types and/or explicitly supplying type arguments.

Other function members

Members that contain executable code are collectively known as the ***function members*** of a class. The preceding section describes methods, which are the primary kind of function members. This section describes the other kinds of function members supported by C#: constructors, properties, indexers, events, operators, and destructors.

The following code shows a generic class called `List<T>`, which implements a growable list of objects. The class contains several examples of the most common kinds of function members.

```
public class List<T> {
    // Constant...
    const int defaultCapacity = 4;

    // Fields...
    T[] items;
    int count;

    // Constructors...
    public List(int capacity = defaultCapacity) {
        items = new T[capacity];
    }

    // Properties...
```

```
    // Properties...
    public int Count {
        get { return count; }
    }
    public int Capacity {
        get {
            return items.Length;
        }
        set {
            if (value < count) value = count;
            if (value != items.Length) {
                T[] newItems = new T[value];
                Array.Copy(items, 0, newItems, 0, count);
                items = newItems;
            }
        }
    }

    // Indexer...
    public T this[int index] {
        get {
            return items[index];
        }
        set {
            items[index] = value;
            OnChanged();
        }
    }

    // Methods...
    public void Add(T item) {
        if (count == Capacity) Capacity = count * 2;
        items[count] = item;
        count++;
        OnChanged();
    }
    protected virtual void OnChanged() {
        if (Changed != null) Changed(this, EventArgs.Empty);
    }
    public override bool Equals(object other) {
        return Equals(this, other as List<T>);
    }
    static bool Equals(List<T> a, List<T> b) {
        if (a == null) return b == null;
        if (b == null || a.count != b.count) return false;
        for (int i = 0; i < a.count; i++) {
            if (!object.Equals(a.items[i], b.items[i])) {
                return false;
            }
        }
        return true;
    }

    // Event...
    public event EventHandler Changed;

    // Operators...
    public static bool operator ==(List<T> a, List<T> b) {
        return Equals(a, b);
    }
    public static bool operator !=(List<T> a, List<T> b) {
        return !Equals(a, b);
    }
}
```

Constructors

C# supports both instance and static constructors. An ***instance constructor*** is a member that implements the actions required to initialize an instance of a class. A ***static constructor*** is a member that implements the

actions required to initialize a class itself when it is first loaded.

A constructor is declared like a method with no return type and the same name as the containing class. If a constructor declaration includes a `static` modifier, it declares a static constructor. Otherwise, it declares an instance constructor.

Instance constructors can be overloaded. For example, the `List<T>` class declares two instance constructors, one with no parameters and one that takes an `int` parameter. Instance constructors are invoked using the `new` operator. The following statements allocate two `List<string>` instances using each of the constructors of the `List` class.

```
List<string> list1 = new List<string>();
List<string> list2 = new List<string>(10);
```

Unlike other members, instance constructors are not inherited, and a class has no instance constructors other than those actually declared in the class. If no instance constructor is supplied for a class, then an empty one with no parameters is automatically provided.

Properties

Properties are a natural extension of fields. Both are named members with associated types, and the syntax for accessing fields and properties is the same. However, unlike fields, properties do not denote storage locations. Instead, properties have ***accessors*** that specify the statements to be executed when their values are read or written.

A property is declared like a field, except that the declaration ends with a `get` accessor and/or a `set` accessor written between the delimiters `{` and `}` instead of ending in a semicolon. A property that has both a `get` accessor and a `set` accessor is a ***read-write property***, a property that has only a `get` accessor is a ***read-only property***, and a property that has only a `set` accessor is a ***write-only property***.

A `get` accessor corresponds to a parameterless method with a return value of the property type. Except as the target of an assignment, when a property is referenced in an expression, the `get` accessor of the property is invoked to compute the value of the property.

A `set` accessor corresponds to a method with a single parameter named `value` and no return type. When a property is referenced as the target of an assignment or as the operand of `++` or `--`, the `set` accessor is invoked with an argument that provides the new value.

The `List<T>` class declares two properties, `Count` and `Capacity`, which are read-only and read-write, respectively. The following is an example of use of these properties.

```
List<string> names = new List<string>();
names.Capacity = 100;           // Invokes set accessor
int i = names.Count;            // Invokes get accessor
int j = names.Capacity;         // Invokes get accessor
```

Similar to fields and methods, C# supports both instance properties and static properties. Static properties are declared with the `static` modifier, and instance properties are declared without it.

The accessor(s) of a property can be virtual. When a property declaration includes a `virtual`, `abstract`, or `override` modifier, it applies to the accessor(s) of the property.

Indexers

An ***indexer*** is a member that enables objects to be indexed in the same way as an array. An indexer is declared like a property except that the name of the member is `this` followed by a parameter list written between the delimiters `[` and `]`. The parameters are available in the accessor(s) of the indexer. Similar to properties, indexers can be read-write, read-only, and write-only, and the accessor(s) of an indexer can be virtual.

The `List` class declares a single read-write indexer that takes an `int` parameter. The indexer makes it possible to index `List` instances with `int` values. For example

```
List<string> names = new List<string>();
names.Add("Liz");
names.Add("Martha");
names.Add("Beth");
for (int i = 0; i < names.Count; i++) {
    string s = names[i];
    names[i] = s.ToUpper();
}
```

Indexers can be overloaded, meaning that a class can declare multiple indexers as long as the number or types of their parameters differ.

Events

An ***event*** is a member that enables a class or object to provide notifications. An event is declared like a field except that the declaration includes an `event` keyword and the type must be a delegate type.

Within a class that declares an event member, the event behaves just like a field of a delegate type (provided the event is not abstract and does not declare accessors). The field stores a reference to a delegate that represents the event handlers that have been added to the event. If no event handles are present, the field is `null`.

The `List<T>` class declares a single event member called `Changed`, which indicates that a new item has been added to the list. The `Changed` event is raised by the `OnChanged` virtual method, which first checks whether the event is `null` (meaning that no handlers are present). The notion of raising an event is precisely equivalent to invoking the delegate represented by the event—thus, there are no special language constructs for raising events.

Clients react to events through ***event handlers***. Event handlers are attached using the `+=` operator and removed using the `-=` operator. The following example attaches an event handler to the `Changed` event of a `List<string>`.

```
using System;

class Test
{
    static int changeCount;

    static void ListChanged(object sender, EventArgs e) {
        changeCount++;
    }

    static void Main() {
        List<string> names = new List<string>();
        names.Changed += new EventHandler(ListChanged);
        names.Add("Liz");
        names.Add("Martha");
        names.Add("Beth");
        Console.WriteLine(changeCount);        // Outputs "3"
    }
}
```

For advanced scenarios where control of the underlying storage of an event is desired, an event declaration can explicitly provide `add` and `remove` accessors, which are somewhat similar to the `set` accessor of a property.

Operators

An ***operator*** is a member that defines the meaning of applying a particular expression operator to instances of a class. Three kinds of operators can be defined: unary operators, binary operators, and conversion operators.

All operators must be declared as `public` and `static`.

The `List<T>` class declares two operators, `operator==` and `operator!=`, and thus gives new meaning to expressions that apply those operators to `List` instances. Specifically, the operators define equality of two `List<T>` instances as comparing each of the contained objects using their `Equals` methods. The following example uses the `==` operator to compare two `List<int>` instances.

```
using System;

class Test
{
    static void Main() {
        List<int> a = new List<int>();
        a.Add(1);
        a.Add(2);
        List<int> b = new List<int>();
        b.Add(1);
        b.Add(2);
        Console.WriteLine(a == b);        // Outputs "True"
        b.Add(3);
        Console.WriteLine(a == b);        // Outputs "False"
    }
}
```

The first `Console.WriteLine` outputs `True` because the two lists contain the same number of objects with the same values in the same order. Had `List<T>` not defined `operator==`, the first `Console.WriteLine` would have output `False` because `a` and `b` reference different `List<int>` instances.

Destructors

A ***destructor*** is a member that implements the actions required to destruct an instance of a class. Destructors cannot have parameters, they cannot have accessibility modifiers, and they cannot be invoked explicitly. The destructor for an instance is invoked automatically during garbage collection.

The garbage collector is allowed wide latitude in deciding when to collect objects and run destructors. Specifically, the timing of destructor invocations is not deterministic, and destructors may be executed on any thread. For these and other reasons, classes should implement destructors only when no other solutions are feasible.

The `using` statement provides a better approach to object destruction.

Structs

Like classes, ***structs*** are data structures that can contain data members and function members, but unlike classes, structs are value types and do not require heap allocation. A variable of a struct type directly stores the data of the struct, whereas a variable of a class type stores a reference to a dynamically allocated object. Struct types do not support user-specified inheritance, and all struct types implicitly inherit from type `object`.

Structs are particularly useful for small data structures that have value semantics. Complex numbers, points in a coordinate system, or key-value pairs in a dictionary are all good examples of structs. The use of structs rather than classes for small data structures can make a large difference in the number of memory allocations an application performs. For example, the following program creates and initializes an array of 100 points. With `Point` implemented as a class, 101 separate objects are instantiated—one for the array and one each for the 100 elements.

```
class Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

class Test
{
    static void Main() {
        Point[] points = new Point[100];
        for (int i = 0; i < 100; i++) points[i] = new Point(i, i);
    }
}
```

An alternative is to make `Point` a struct.

```
struct Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Now, only one object is instantiated—the one for the array—and the `Point` instances are stored in-line in the array.

Struct constructors are invoked with the `new` operator, but that does not imply that memory is being allocated. Instead of dynamically allocating an object and returning a reference to it, a struct constructor simply returns the struct value itself (typically in a temporary location on the stack), and this value is then copied as necessary.

With classes, it is possible for two variables to reference the same object and thus possible for operations on one variable to affect the object referenced by the other variable. With structs, the variables each have their own copy of the data, and it is not possible for operations on one to affect the other. For example, the output produced by the following code fragment depends on whether `Point` is a class or a struct.

```
Point a = new Point(10, 10);
Point b = a;
a.x = 20;
Console.WriteLine(b.x);
```

If `Point` is a class, the output is `20` because `a` and `b` reference the same object. If `Point` is a struct, the output is `10` because the assignment of `a` to `b` creates a copy of the value, and this copy is unaffected by the subsequent assignment to `a.x`.

The previous example highlights two of the limitations of structs. First, copying an entire struct is typically less efficient than copying an object reference, so assignment and value parameter passing can be more expensive with structs than with reference types. Second, except for `ref` and `out` parameters, it is not possible to create references to structs, which rules out their usage in a number of situations.

Arrays

An ***array*** is a data structure that contains a number of variables that are accessed through computed indices. The variables contained in an array, also called the ***elements*** of the array, are all of the same type, and this type is called the ***element type*** of the array.

Array types are reference types, and the declaration of an array variable simply sets aside space for a reference to an array instance. Actual array instances are created dynamically at run-time using the `new` operator. The `new` operation specifies the ***length*** of the new array instance, which is then fixed for the lifetime of the instance. The indices of the elements of an array range from `0` to `Length - 1`. The `new` operator automatically initializes the elements of an array to their default value, which, for example, is zero for all numeric types and `null` for all reference types.

The following example creates an array of `int` elements, initializes the array, and prints out the contents of the array.

```
using System;

class Test
{
    static void Main() {
        int[] a = new int[10];
        for (int i = 0; i < a.Length; i++) {
            a[i] = i * i;
        }
        for (int i = 0; i < a.Length; i++) {
            Console.WriteLine("a[{0}] = {1}", i, a[i]);
        }
    }
}
```

This example creates and operates on a ***single-dimensional array***. C# also supports ***multi-dimensional arrays***. The number of dimensions of an array type, also known as the ***rank*** of the array type, is one plus the number of commas written between the square brackets of the array type. The following example allocates a one-dimensional, a two-dimensional, and a three-dimensional array.

```
int[] a1 = new int[10];
int[,] a2 = new int[10, 5];
int[,,] a3 = new int[10, 5, 2];
```

The `a1` array contains 10 elements, the `a2` array contains 50 (10 × 5) elements, and the `a3` array contains 100 (10 × 5 × 2) elements.

The element type of an array can be any type, including an array type. An array with elements of an array type is sometimes called a ***jagged array*** because the lengths of the element arrays do not all have to be the same. The following example allocates an array of arrays of `int`:

```
int[][] a = new int[3][];
a[0] = new int[10];
a[1] = new int[5];
a[2] = new int[20];
```

The first line creates an array with three elements, each of type `int[]` and each with an initial value of `null`. The subsequent lines then initialize the three elements with references to individual array instances of varying lengths.

The `new` operator permits the initial values of the array elements to be specified using an ***array initializer***, which is a list of expressions written between the delimiters `{` and `}`. The following example allocates and

initializes an `int[]` with three elements.

```
int[] a = new int[] {1, 2, 3};
```

Note that the length of the array is inferred from the number of expressions between `{` and `}`. Local variable and field declarations can be shortened further such that the array type does not have to be restated.

```
int[] a = {1, 2, 3};
```

Both of the previous examples are equivalent to the following:

```
int[] t = new int[3];
t[0] = 1;
t[1] = 2;
t[2] = 3;
int[] a = t;
```

Interfaces

An ***interface*** defines a contract that can be implemented by classes and structs. An interface can contain methods, properties, events, and indexers. An interface does not provide implementations of the members it defines—it merely specifies the members that must be supplied by classes or structs that implement the interface.

Interfaces may employ ***multiple inheritance***. In the following example, the interface `IComboBox` inherits from both `ITextBox` and `IListBox`.

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

interface IListBox: IControl
{
    void SetItems(string[] items);
}

interface IComboBox: ITextBox, IListBox {}
```

Classes and structs can implement multiple interfaces. In the following example, the class `EditBox` implements both `IControl` and `IDataBound`.

```
interface IDataBound
{
    void Bind(Binder b);
}

public class EditBox: IControl, IDataBound
{
    public void Paint() {...}
    public void Bind(Binder b) {...}
}
```

When a class or struct implements a particular interface, instances of that class or struct can be implicitly converted to that interface type. For example

```
EditBox editBox = new EditBox();
IControl control = editBox;
IDataBound dataBound = editBox;
```

In cases where an instance is not statically known to implement a particular interface, dynamic type casts can be used. For example, the following statements use dynamic type casts to obtain an object's `IControl` and `IDataBound` interface implementations. Because the actual type of the object is `EditBox`, the casts succeed.

```
object obj = new EditBox();
IControl control = (IControl)obj;
IDataBound dataBound = (IDataBound)obj;
```

In the previous `EditBox` class, the `Paint` method from the `IControl` interface and the `Bind` method from the `IDataBound` interface are implemented using `public` members. C# also supports ***explicit interface member implementations***, using which the class or struct can avoid making the members `public`. An explicit interface member implementation is written using the fully qualified interface member name. For example, the `EditBox` class could implement the `IControl.Paint` and `IDataBound.Bind` methods using explicit interface member implementations as follows.

```
public class EditBox: IControl, IDataBound
{
    void IControl.Paint() {...}
    void IDataBound.Bind(Binder b) {...}
}
```

Explicit interface members can only be accessed via the interface type. For example, the implementation of `IControl.Paint` provided by the previous `EditBox` class can only be invoked by first converting the `EditBox` reference to the `IControl` interface type.

```
EditBox editBox = new EditBox();
editBox.Paint();                        // Error, no such method
IControl control = editBox;
control.Paint();                        // Ok
```

Enums

An ***enum type*** is a distinct value type with a set of named constants. The following example declares and uses an enum type named `Color` with three constant values, `Red`, `Green`, and `Blue`.

```
using System;

enum Color
{
    Red,
    Green,
    Blue
}

class Test
{
    static void PrintColor(Color color) {
        switch (color) {
            case Color.Red:
                Console.WriteLine("Red");
                break;
            case Color.Green:
                Console.WriteLine("Green");
                break;
            case Color.Blue:
                Console.WriteLine("Blue");
                break;
            default:
                Console.WriteLine("Unknown color");
                break;
        }
    }

    static void Main() {
        Color c = Color.Red;
        PrintColor(c);
        PrintColor(Color.Blue);
    }
}
```

Each enum type has a corresponding integral type called the ***underlying type*** of the enum type. An enum type that does not explicitly declare an underlying type has an underlying type of `int`. An enum type's storage format and range of possible values are determined by its underlying type. The set of values that an enum type can take on is not limited by its enum members. In particular, any value of the underlying type of an enum can be cast to the enum type and is a distinct valid value of that enum type.

The following example declares an enum type named `Alignment` with an underlying type of `sbyte`.

```
enum Alignment: sbyte
{
    Left = -1,
    Center = 0,
    Right = 1
}
```

As shown by the previous example, an enum member declaration can include a constant expression that specifies the value of the member. The constant value for each enum member must be in the range of the underlying type of the enum. When an enum member declaration does not explicitly specify a value, the member is given the value zero (if it is the first member in the enum type) or the value of the textually preceding enum member plus one.

Enum values can be converted to integral values and vice versa using type casts. For example

```
int i = (int)Color.Blue;        // int i = 2;
Color c = (Color)2;             // Color c = Color.Blue;
```

The default value of any enum type is the integral value zero converted to the enum type. In cases where variables are automatically initialized to a default value, this is the value given to variables of enum types. In order for the default value of an enum type to be easily available, the literal `0` implicitly converts to any enum type. Thus, the following is permitted.

```
Color c = 0;
```

Delegates

A ***delegate type*** represents references to methods with a particular parameter list and return type. Delegates make it possible to treat methods as entities that can be assigned to variables and passed as parameters. Delegates are similar to the concept of function pointers found in some other languages, but unlike function pointers, delegates are object-oriented and type-safe.

The following example declares and uses a delegate type named `Function`.

```
using System;

delegate double Function(double x);

class Multiplier
{
    double factor;

    public Multiplier(double factor) {
        this.factor = factor;
    }

    public double Multiply(double x) {
        return x * factor;
    }
}

class Test
{
    static double Square(double x) {
        return x * x;
    }

    static double[] Apply(double[] a, Function f) {
        double[] result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }

    static void Main() {
        double[] a = {0.0, 0.5, 1.0};
        double[] squares = Apply(a, Square);
        double[] sines = Apply(a, Math.Sin);
        Multiplier m = new Multiplier(2.0);
        double[] doubles =  Apply(a, m.Multiply);
    }
}
```

An instance of the `Function` delegate type can reference any method that takes a `double` argument and returns a `double` value. The `Apply` method applies a given `Function` to the elements of a `double[]`, returning a `double[]` with the results. In the `Main` method, `Apply` is used to apply three different functions to a `double[]`.

A delegate can reference either a static method (such as `Square` or `Math.Sin` in the previous example) or an instance method (such as `m.Multiply` in the previous example). A delegate that references an instance method

also references a particular object, and when the instance method is invoked through the delegate, that object becomes `this` in the invocation.

Delegates can also be created using anonymous functions, which are "inline methods" that are created on the fly. Anonymous functions can see the local variables of the surrounding methods. Thus, the multiplier example above can be written more easily without using a `Multiplier` class:

```
double[] doubles =  Apply(a, (double x) => x * 2.0);
```

An interesting and useful property of a delegate is that it does not know or care about the class of the method it references; all that matters is that the referenced method has the same parameters and return type as the delegate.

Attributes

Types, members, and other entities in a C# program support modifiers that control certain aspects of their behavior. For example, the accessibility of a method is controlled using the `public`, `protected`, `internal`, and `private` modifiers. C# generalizes this capability such that user-defined types of declarative information can be attached to program entities and retrieved at run-time. Programs specify this additional declarative information by defining and using ***attributes***.

The following example declares a `HelpAttribute` attribute that can be placed on program entities to provide links to their associated documentation.

```
using System;

public class HelpAttribute: Attribute
{
    string url;
    string topic;

    public HelpAttribute(string url) {
        this.url = url;
    }

    public string Url {
        get { return url; }
    }

    public string Topic {
        get { return topic; }
        set { topic = value; }
    }
}
```

All attribute classes derive from the `System.Attribute` base class provided by the .NET Framework. Attributes can be applied by giving their name, along with any arguments, inside square brackets just before the associated declaration. If an attribute's name ends in `Attribute`, that part of the name can be omitted when the attribute is referenced. For example, the `HelpAttribute` attribute can be used as follows.

```
[Help("http://msdn.microsoft.com/.../MyClass.htm")]
public class Widget
{
    [Help("http://msdn.microsoft.com/.../MyClass.htm", Topic = "Display")]
    public void Display(string text) {}
}
```

This example attaches a `HelpAttribute` to the `Widget` class and another `HelpAttribute` to the `Display` method in the class. The public constructors of an attribute class control the information that must be provided when the attribute is attached to a program entity. Additional information can be provided by referencing public read-write properties of the attribute class (such as the reference to the `Topic` property previously).

The following example shows how attribute information for a given program entity can be retrieved at run-time using reflection.

```
using System;
using System.Reflection;

class Test
{
    static void ShowHelp(MemberInfo member) {
        HelpAttribute a = Attribute.GetCustomAttribute(member,
            typeof(HelpAttribute)) as HelpAttribute;
        if (a == null) {
            Console.WriteLine("No help for {0}", member);
        }
        else {
            Console.WriteLine("Help for {0}:", member);
            Console.WriteLine("  Url={0}, Topic={1}", a.Url, a.Topic);
        }
    }

    static void Main() {
        ShowHelp(typeof(Widget));
        ShowHelp(typeof(Widget).GetMethod("Display"));
    }
}
```

When a particular attribute is requested through reflection, the constructor for the attribute class is invoked with the information provided in the program source, and the resulting attribute instance is returned. If additional information was provided through properties, those properties are set to the given values before the attribute instance is returned.

Lexical structure

18.12.2021 • 33 minutes to read

Programs

A C# ***program*** consists of one or more ***source files***, known formally as ***compilation units*** (Compilation units). A source file is an ordered sequence of Unicode characters. Source files typically have a one-to-one correspondence with files in a file system, but this correspondence is not required. For maximal portability, it is recommended that files in a file system be encoded with the UTF-8 encoding.

Conceptually speaking, a program is compiled using three steps:

1. Transformation, which converts a file from a particular character repertoire and encoding scheme into a sequence of Unicode characters.
2. Lexical analysis, which translates a stream of Unicode input characters into a stream of tokens.
3. Syntactic analysis, which translates the stream of tokens into executable code.

Grammars

This specification presents the syntax of the C# programming language using two grammars. The ***lexical grammar*** (Lexical grammar) defines how Unicode characters are combined to form line terminators, white space, comments, tokens, and pre-processing directives. The ***syntactic grammar*** (Syntactic grammar) defines how the tokens resulting from the lexical grammar are combined to form C# programs.

Grammar notation

The lexical and syntactic grammars are presented in Backus-Naur form using the notation of the ANTLR grammar tool.

Lexical grammar

The lexical grammar of C# is presented in Lexical analysis, Tokens, and Pre-processing directives. The terminal symbols of the lexical grammar are the characters of the Unicode character set, and the lexical grammar specifies how characters are combined to form tokens (Tokens), white space (White space), comments (Comments), and pre-processing directives (Pre-processing directives).

Every source file in a C# program must conform to the *input* production of the lexical grammar (Lexical analysis).

Syntactic grammar

The syntactic grammar of C# is presented in the chapters and appendices that follow this chapter. The terminal symbols of the syntactic grammar are the tokens defined by the lexical grammar, and the syntactic grammar specifies how tokens are combined to form C# programs.

Every source file in a C# program must conform to the *compilation_unit* production of the syntactic grammar (Compilation units).

Lexical analysis

The *input* production defines the lexical structure of a C# source file. Each source file in a C# program must conform to this lexical grammar production.

```
input
    : input_section?
    ;

input_section
    : input_section_part+
    ;

input_section_part
    : input_element* new_line
    | pp_directive
    ;

input_element
    : whitespace
    | comment
    | token
    ;
```

Five basic elements make up the lexical structure of a C# source file: Line terminators (Line terminators), white space (White space), comments (Comments), tokens (Tokens), and pre-processing directives (Pre-processing directives). Of these basic elements, only tokens are significant in the syntactic grammar of a C# program (Syntactic grammar).

The lexical processing of a C# source file consists of reducing the file into a sequence of tokens which becomes the input to the syntactic analysis. Line terminators, white space, and comments can serve to separate tokens, and pre-processing directives can cause sections of the source file to be skipped, but otherwise these lexical elements have no impact on the syntactic structure of a C# program.

In the case of interpolated string literals (Interpolated string literals) a single token is initially produced by lexical analysis, but is broken up into several input elements which are repeatedly subjected to lexical analysis until all interpolated string literals have been resolved. The resulting tokens then serve as input to the syntactic analysis.

When several lexical grammar productions match a sequence of characters in a source file, the lexical processing always forms the longest possible lexical element. For example, the character sequence `//` is processed as the beginning of a single-line comment because that lexical element is longer than a single `/` token.

Line terminators

Line terminators divide the characters of a C# source file into lines.

```
new_line
    : '<Carriage return character (U+000D)>'
    | '<Line feed character (U+000A)>'
    | '<Carriage return character (U+000D) followed by line feed character (U+000A)>'
    | '<Next line character (U+0085)>'
    | '<Line separator character (U+2028)>'
    | '<Paragraph separator character (U+2029)>'
    ;
```

For compatibility with source code editing tools that add end-of-file markers, and to enable a source file to be viewed as a sequence of properly terminated lines, the following transformations are applied, in order, to every source file in a C# program:

- If the last character of the source file is a Control-Z character (`U+001A`), this character is deleted.
- A carriage-return character (`U+000D`) is added to the end of the source file if that source file is non-empty and if the last character of the source file is not a carriage return (`U+000D`), a line feed (`U+000A`), a line separator (`U+2028`), or a paragraph separator (`U+2029`).

Comments

Two forms of comments are supported: single-line comments and delimited comments. ***Single-line comments*** start with the characters `//` and extend to the end of the source line. ***Delimited comments*** start with the characters `/*` and end with the characters `*/`. Delimited comments may span multiple lines.

```
comment
    : single_line_comment
    | delimited_comment
    ;

single_line_comment
    : '//' input_character*
    ;

input_character
    : '<Any Unicode character except a new_line_character>'
    ;

new_line_character
    : '<Carriage return character (U+000D)>'
    | '<Line feed character (U+000A)>'
    | '<Next line character (U+0085)>'
    | '<Line separator character (U+2028)>'
    | '<Paragraph separator character (U+2029)>'
    ;

delimited_comment
    : '/*' delimited_comment_section* asterisk+ '/'
    ;

delimited_comment_section
    : '/'
    | asterisk* not_slash_or_asterisk
    ;

asterisk
    : '*'
    ;

not_slash_or_asterisk
    : '<Any Unicode character except / or *>'
    ;
```

Comments do not nest. The character sequences `/*` and `*/` have no special meaning within a `//` comment, and the character sequences `//` and `/*` have no special meaning within a delimited comment.

Comments are not processed within character and string literals.

The example

```
/* Hello, world program
   This program writes "hello, world" to the console
*/
class Hello
{
    static void Main() {
        System.Console.WriteLine("hello, world");
    }
}
```

includes a delimited comment.

The example

```
// Hello, world program
// This program writes "hello, world" to the console
//
class Hello // any name will do for this class
{
    static void Main() { // this method must be named "Main"
        System.Console.WriteLine("hello, world");
    }
}
```

shows several single-line comments.

White space

White space is defined as any character with Unicode class Zs (which includes the space character) as well as the horizontal tab character, the vertical tab character, and the form feed character.

```
whitespace
    : '<Any character with Unicode class Zs>'
    | '<Horizontal tab character (U+0009)>'
    | '<Vertical tab character (U+000B)>'
    | '<Form feed character (U+000C)>'
    ;
```

Tokens

There are several kinds of tokens: identifiers, keywords, literals, operators, and punctuators. White space and comments are not tokens, though they act as separators for tokens.

```
token
    : identifier
    | keyword
    | integer_literal
    | real_literal
    | character_literal
    | string_literal
    | interpolated_string_literal
    | operator_or_punctuator
    ;
```

Unicode character escape sequences

A Unicode character escape sequence represents a Unicode character. Unicode character escape sequences are processed in identifiers (Identifiers), character literals (Character literals), and regular string literals (String literals). A Unicode character escape is not processed in any other location (for example, to form an operator, punctuator, or keyword).

```
unicode_escape_sequence
    : '\\u' hex_digit hex_digit hex_digit hex_digit
    | '\\U' hex_digit hex_digit hex_digit hex_digit hex_digit hex_digit hex_digit hex_digit
    ;
```

A Unicode escape sequence represents the single Unicode character formed by the hexadecimal number following the "`\u`" or "`\U`" characters. Since C# uses a 16-bit encoding of Unicode code points in characters and string values, a Unicode character in the range U+10000 to U+10FFFF is not permitted in a character literal and is represented using a Unicode surrogate pair in a string literal. Unicode characters with code points above 0x10FFFF are not supported.

Multiple translations are not performed. For instance, the string literal "`\u005Cu005C`" is equivalent to "`\u005C`" rather than "`\`". The Unicode value `\u005C` is the character "`\`".

The example

```
class Class1
{
    static void Test(bool \u0066) {
        char c = '\u0066';
        if (\u0066)
            System.Console.WriteLine(c.ToString());
    }
}
```

shows several uses of `\u0066`, which is the escape sequence for the letter "`f`". The program is equivalent to

```
class Class1
{
    static void Test(bool f) {
        char c = 'f';
        if (f)
            System.Console.WriteLine(c.ToString());
    }
}
```

Identifiers

The rules for identifiers given in this section correspond exactly to those recommended by the Unicode Standard Annex 31, except that underscore is allowed as an initial character (as is traditional in the C programming language), Unicode escape sequences are permitted in identifiers, and the "`@`" character is allowed as a prefix to enable keywords to be used as identifiers.

```
identifier
    : available_identifier
    | '@' identifier_or_keyword
    ;

available_identifier
    : '<An identifier_or_keyword that is not a keyword>'
    ;

identifier_or_keyword
    : identifier_start_character identifier_part_character*
    ;

identifier_start_character
    : letter_character
    | '_'
    ;

identifier_part_character
    : letter_character
    | decimal_digit_character
    | connecting_character
    | combining_character
    | formatting_character
    ;

letter_character
    : '<A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl>'
    | '<A unicode_escape_sequence representing a character of classes Lu, Ll, Lt, Lm, Lo, or Nl>'
    ;

combining_character
    : '<A Unicode character of classes Mn or Mc>'
    | '<A unicode_escape_sequence representing a character of classes Mn or Mc>'
    ;

decimal_digit_character
    : '<A Unicode character of the class Nd>'
    | '<A unicode_escape_sequence representing a character of the class Nd>'
    ;

connecting_character
    : '<A Unicode character of the class Pc>'
    | '<A unicode_escape_sequence representing a character of the class Pc>'
    ;

formatting_character
    : '<A Unicode character of the class Cf>'
    | '<A unicode_escape_sequence representing a character of the class Cf>'
    ;
```

For information on the Unicode character classes mentioned above, see The Unicode Standard, Version 3.0, section 4.5.

Examples of valid identifiers include "`identifier1`", "`_identifier2`", and "`@if`".

An identifier in a conforming program must be in the canonical format defined by Unicode Normalization Form C, as defined by Unicode Standard Annex 15. The behavior when encountering an identifier not in Normalization Form C is implementation-defined; however, a diagnostic is not required.

The prefix "`@`" enables the use of keywords as identifiers, which is useful when interfacing with other programming languages. The character `@` is not actually part of the identifier, so the identifier might be seen in other languages as a normal identifier, without the prefix. An identifier with an `@` prefix is called a ***verbatim identifier***. Use of the `@` prefix for identifiers that are not keywords is permitted, but strongly discouraged as a

matter of style.

The example:

```
class @class
{
    public static void @static(bool @bool) {
        if (@bool)
            System.Console.WriteLine("true");
        else
            System.Console.WriteLine("false");
    }
}

class Class1
{
    static void M() {
        cl\u0061ss.st\u0061tic(true);
    }
}
```

defines a class named "`class`" with a static method named "`static`" that takes a parameter named "`bool`". Note that since Unicode escapes are not permitted in keywords, the token "`cl\u0061ss`" is an identifier, and is the same identifier as "`@class`".

Two identifiers are considered the same if they are identical after the following transformations are applied, in order:

- The prefix "`@`", if used, is removed.
- Each *unicode_escape_sequence* is transformed into its corresponding Unicode character.
- Any *formatting_character*s are removed.

Identifiers containing two consecutive underscore characters (`U+005F`) are reserved for use by the implementation. For example, an implementation might provide extended keywords that begin with two underscores.

Keywords

A ***keyword*** is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier except when prefaced by the `@` character.

```
keyword
    : 'abstract' | 'as'       | 'base'       | 'bool'      | 'break'
    | 'byte'     | 'case'     | 'catch'      | 'char'      | 'checked'
    | 'class'    | 'const'    | 'continue'   | 'decimal'   | 'default'
    | 'delegate' | 'do'       | 'double'     | 'else'      | 'enum'
    | 'event'    | 'explicit' | 'extern'     | 'false'     | 'finally'
    | 'fixed'    | 'float'    | 'for'        | 'foreach'   | 'goto'
    | 'if'       | 'implicit' | 'in'         | 'int'       | 'interface'
    | 'internal' | 'is'       | 'lock'       | 'long'      | 'namespace'
    | 'new'      | 'null'     | 'object'     | 'operator'  | 'out'
    | 'override' | 'params'   | 'private'    | 'protected' | 'public'
    | 'readonly' | 'ref'      | 'return'     | 'sbyte'     | 'sealed'
    | 'short'    | 'sizeof'   | 'stackalloc' | 'static'    | 'string'
    | 'struct'   | 'switch'   | 'this'       | 'throw'     | 'true'
    | 'try'      | 'typeof'   | 'uint'       | 'ulong'     | 'unchecked'
    | 'unsafe'   | 'ushort'   | 'using'      | 'virtual'   | 'void'
    | 'volatile' | 'while'
    ;
```

In some places in the grammar, specific identifiers have special meaning, but are not keywords. Such identifiers

are sometimes referred to as "contextual keywords". For example, within a property declaration, the "`get`" and "`set`" identifiers have special meaning (Accessors). An identifier other than `get` or `set` is never permitted in these locations, so this use does not conflict with a use of these words as identifiers. In other cases, such as with the identifier "`var`" in implicitly typed local variable declarations (Local variable declarations), a contextual keyword can conflict with declared names. In such cases, the declared name takes precedence over the use of the identifier as a contextual keyword.

Literals

A *literal* is a source code representation of a value.

```
literal
    : boolean_literal
    | integer_literal
    | real_literal
    | character_literal
    | string_literal
    | null_literal
    ;
```

Boolean literals

There are two boolean literal values: `true` and `false`.

```
boolean_literal
    : 'true'
    | 'false'
    ;
```

The type of a *boolean_literal* is `bool`.

Integer literals

Integer literals are used to write values of types `int`, `uint`, `long`, and `ulong`. Integer literals have two possible forms: decimal and hexadecimal.

```
integer_literal
    : decimal_integer_literal
    | hexadecimal_integer_literal
    ;

decimal_integer_literal
    : decimal_digit+ integer_type_suffix?
    ;

decimal_digit
    : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
    ;

integer_type_suffix
    : 'U' | 'u' | 'L' | 'l' | 'UL' | 'Ul' | 'uL' | 'ul' | 'LU' | 'Lu' | 'lU' | 'lu'
    ;

hexadecimal_integer_literal
    : '0x' hex_digit+ integer_type_suffix?
    | '0X' hex_digit+ integer_type_suffix?
    ;

hex_digit
    : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
    | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f';
```

The type of an integer literal is determined as follows:

- If the literal has no suffix, it has the first of these types in which its value can be represented: `int`, `uint`, `long`, `ulong`.
- If the literal is suffixed by `U` or `u`, it has the first of these types in which its value can be represented: `uint`, `ulong`.
- If the literal is suffixed by `L` or `l`, it has the first of these types in which its value can be represented: `long`, `ulong`.
- If the literal is suffixed by `UL`, `Ul`, `uL`, `ul`, `LU`, `Lu`, `lU`, or `lu`, it is of type `ulong`.

If the value represented by an integer literal is outside the range of the `ulong` type, a compile-time error occurs.

As a matter of style, it is suggested that "`L`" be used instead of "`l`" when writing literals of type `long`, since it is easy to confuse the letter "`l`" with the digit "`1`".

To permit the smallest possible `int` and `long` values to be written as decimal integer literals, the following two rules exist:

- When a *decimal_integer_literal* with the value 2147483648 (2^31) and no *integer_type_suffix* appears as the token immediately following a unary minus operator token (Unary minus operator), the result is a constant of type `int` with the value -2147483648 (-2^31). In all other situations, such a *decimal_integer_literal* is of type `uint`.
- When a *decimal_integer_literal* with the value 9223372036854775808 (2^63) and no *integer_type_suffix* or the *integer_type_suffix* `L` or `l` appears as the token immediately following a unary minus operator token (Unary minus operator), the result is a constant of type `long` with the value -9223372036854775808 (-2^63). In all other situations, such a *decimal_integer_literal* is of type `ulong`.

Real literals

Real literals are used to write values of types `float`, `double`, and `decimal`.

```
real_literal
    : decimal_digit+ '.' decimal_digit+ exponent_part? real_type_suffix?
    | '.' decimal_digit+ exponent_part? real_type_suffix?
    | decimal_digit+ exponent_part real_type_suffix?
    | decimal_digit+ real_type_suffix
    ;

exponent_part
    : 'e' sign? decimal_digit+
    | 'E' sign? decimal_digit+
    ;

sign
    : '+'
    | '-'
    ;

real_type_suffix
    : 'F' | 'f' | 'D' | 'd' | 'M' | 'm'
    ;
```

If no *real_type_suffix* is specified, the type of the real literal is `double`. Otherwise, the real type suffix determines the type of the real literal, as follows:

- A real literal suffixed by `F` or `f` is of type `float`. For example, the literals `1f`, `1.5f`, `1e10f`, and `123.456F` are all of type `float`.
- A real literal suffixed by `D` or `d` is of type `double`. For example, the literals `1d`, `1.5d`, `1e10d`, and `123.456D` are all of type `double`.
- A real literal suffixed by `M` or `m` is of type `decimal`. For example, the literals `1m`, `1.5m`, `1e10m`, and

`123.456M` are all of type `decimal`. This literal is converted to a `decimal` value by taking the exact value, and, if necessary, rounding to the nearest representable value using banker's rounding (The decimal type). Any scale apparent in the literal is preserved unless the value is rounded or the value is zero (in which latter case the sign and scale will be 0). Hence, the literal `2.900m` will be parsed to form the decimal with sign `0`, coefficient `2900`, and scale `3`.

If the specified literal cannot be represented in the indicated type, a compile-time error occurs.

The value of a real literal of type `float` or `double` is determined by using the IEEE "round to nearest" mode.

Note that in a real literal, decimal digits are always required after the decimal point. For example, `1.3F` is a real literal but `1.F` is not.

Character literals

A character literal represents a single character, and usually consists of a character in quotes, as in `'a'`.

Note: The ANTLR grammar notation makes the following confusing! In ANTLR, when you write `\'` it stands for a single quote `'`. And when you write `\\` it stands for a single backslash `\`. Therefore the first rule for a character literal means it starts with a single quote, then a character, then a single quote. And the eleven possible simple escape sequences are `\'`, `\"`, `\\`, `\0`, `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, `\v`.

```
character_literal
    : '\'' character '\''
    ;

character
    : single_character
    | simple_escape_sequence
    | hexadecimal_escape_sequence
    | unicode_escape_sequence
    ;

single_character
    : '<Any character except \' (U+0027), \\ (U+005C), and new_line_character>'
    ;

simple_escape_sequence
    : '\\\'' | '\\"' | '\\\\' | '\\0' | '\\a' | '\\b' | '\\f' | '\\n' | '\\r' | '\\t' | '\\v'
    ;

hexadecimal_escape_sequence
    : '\\x' hex_digit hex_digit? hex_digit? hex_digit?;
```

A character that follows a backslash character (`\`) in a *character* must be one of the following characters: `'`, `"`, `\`, `0`, `a`, `b`, `f`, `n`, `r`, `t`, `u`, `U`, `x`, `v`. Otherwise, a compile-time error occurs.

A hexadecimal escape sequence represents a single Unicode character, with the value formed by the hexadecimal number following "`\x`".

If the value represented by a character literal is greater than `U+FFFF`, a compile-time error occurs.

A Unicode character escape sequence (Unicode character escape sequences) in a character literal must be in the range `U+0000` to `U+FFFF`.

A simple escape sequence represents a Unicode character encoding, as described in the table below.

ESCAPE SEQUENCE	CHARACTER NAME	UNICODE ENCODING
`\'`	Single quote	`0x0027`

ESCAPE SEQUENCE	CHARACTER NAME	UNICODE ENCODING
`\"`	Double quote	`0x0022`
`\\`	Backslash	`0x005C`
`\0`	Null	`0x0000`
`\a`	Alert	`0x0007`
`\b`	Backspace	`0x0008`
`\f`	Form feed	`0x000C`
`\n`	New line	`0x000A`
`\r`	Carriage return	`0x000D`
`\t`	Horizontal tab	`0x0009`
`\v`	Vertical tab	`0x000B`

The type of a *character_literal* is `char`.

String literals

C# supports two forms of string literals: ***regular string literals*** and ***verbatim string literals***.

A regular string literal consists of zero or more characters enclosed in double quotes, as in `"hello"`, and may include both simple escape sequences (such as `\t` for the tab character), and hexadecimal and Unicode escape sequences.

A verbatim string literal consists of an `@` character followed by a double-quote character, zero or more characters, and a closing double-quote character. A simple example is `@"hello"`. In a verbatim string literal, the characters between the delimiters are interpreted verbatim, the only exception being a *quote_escape_sequence*. In particular, simple escape sequences, and hexadecimal and Unicode escape sequences are not processed in verbatim string literals. A verbatim string literal may span multiple lines.

```
string_literal
    : regular_string_literal
    | verbatim_string_literal
    ;

regular_string_literal
    : '"' regular_string_literal_character* '"'
    ;

regular_string_literal_character
    : single_regular_string_literal_character
    | simple_escape_sequence
    | hexadecimal_escape_sequence
    | unicode_escape_sequence
    ;

single_regular_string_literal_character
    : '<Any character except " (U+0022), \\ (U+005C), and new_line_character>'
    ;

verbatim_string_literal
    : '@"' verbatim_string_literal_character* '"'
    ;

verbatim_string_literal_character
    : single_verbatim_string_literal_character
    | quote_escape_sequence
    ;

single_verbatim_string_literal_character
    : '<any character except ">'
    ;

quote_escape_sequence
    : '""'
    ;
```

A character that follows a backslash character (`\`) in a *regular_string_literal_character* must be one of the following characters: `'`, `"`, `\`, `0`, `a`, `b`, `f`, `n`, `r`, `t`, `u`, `U`, `x`, `v`. Otherwise, a compile-time error occurs.

The example

```
string a = "hello, world";                  // hello, world
string b = @"hello, world";                 // hello, world

string c = "hello \t world";                // hello      world
string d = @"hello \t world";               // hello \t world

string e = "Joe said \"Hello\" to me";      // Joe said "Hello" to me
string f = @"Joe said ""Hello"" to me";     // Joe said "Hello" to me

string g = "\\\\server\\share\\file.txt";   // \\server\share\file.txt
string h = @"\\server\share\file.txt";      // \\server\share\file.txt

string i = "one\r\ntwo\r\nthree";
string j = @"one
two
three";
```

shows a variety of string literals. The last string literal, `j`, is a verbatim string literal that spans multiple lines. The characters between the quotation marks, including white space such as new line characters, are preserved verbatim.

Since a hexadecimal escape sequence can have a variable number of hex digits, the string literal `"\x123"` contains a single character with hex value 123. To create a string containing the character with hex value 12 followed by the character 3, one could write `"\x00123"` or `"\x12" + "3"` instead.

The type of a *string_literal* is `string`.

Each string literal does not necessarily result in a new string instance. When two or more string literals that are equivalent according to the string equality operator (String equality operators) appear in the same program, these string literals refer to the same string instance. For instance, the output produced by

```
class Test
{
    static void Main() {
        object a = "hello";
        object b = "hello";
        System.Console.WriteLine(a == b);
    }
}
```

is `True` because the two literals refer to the same string instance.

Interpolated string literals

Interpolated string literals are similar to string literals, but contain holes delimited by `{` and `}`, wherein expressions can occur. At runtime, the expressions are evaluated with the purpose of having their textual forms substituted into the string at the place where the hole occurs. The syntax and semantics of string interpolation are described in section (Interpolated strings).

Like string literals, interpolated string literals can be either regular or verbatim. Interpolated regular string literals are delimited by `$"` and `"`, and interpolated verbatim string literals are delimited by `$@"` and `"`.

Like other literals, lexical analysis of an interpolated string literal initially results in a single token, as per the grammar below. However, before syntactic analysis, the single token of an interpolated string literal is broken into several tokens for the parts of the string enclosing the holes, and the input elements occurring in the holes are lexically analysed again. This may in turn produce more interpolated string literals to be processed, but, if lexically correct, will eventually lead to a sequence of tokens for syntactic analysis to process.

```
interpolated_string_literal
    : '$' interpolated_regular_string_literal
    | '$' interpolated_verbatim_string_literal
    ;

interpolated_regular_string_literal
    : interpolated_regular_string_whole
    | interpolated_regular_string_start  interpolated_regular_string_literal_body
interpolated_regular_string_end
    ;

interpolated_regular_string_literal_body
    : regular_balanced_text
    | interpolated_regular_string_literal_body interpolated_regular_string_mid regular_balanced_text
    ;

interpolated_regular_string_whole
    : '"' interpolated_regular_string_character* '"'
    ;

interpolated_regular_string_start
    : '"' interpolated_regular_string_character* '{'
    ;

interpolated_regular_string_mid
```

```
    : interpolation_format? '}' interpolated_regular_string_characters_after_brace? '{'
    ;

interpolated_regular_string_end
    : interpolation_format? '}' interpolated_regular_string_characters_after_brace? '""'
    ;

interpolated_regular_string_characters_after_brace
    : interpolated_regular_string_character_no_brace
    | interpolated_regular_string_characters_after_brace interpolated_regular_string_character
    ;

interpolated_regular_string_character
    : single_interpolated_regular_string_character
    | simple_escape_sequence
    | hexadecimal_escape_sequence
    | unicode_escape_sequence
    | open_brace_escape_sequence
    | close_brace_escape_sequence
    ;

interpolated_regular_string_character_no_brace
    : '<Any interpolated_regular_string_character except close_brace_escape_sequence and any
hexadecimal_escape_sequence or unicode_escape_sequence designating } (U+007D)>'
    ;

single_interpolated_regular_string_character
    : '<Any character except \" (U+0022), \\ (U+005C), { (U+007B), } (U+007D), and new_line_character>'
    ;

open_brace_escape_sequence
    : '{{'
    ;

close_brace_escape_sequence
    : '}}'
    ;

regular_balanced_text
    : regular_balanced_text_part+
    ;

regular_balanced_text_part
    : single_regular_balanced_text_character
    | delimited_comment
    | '@' identifier_or_keyword
    | string_literal
    | interpolated_string_literal
    | '(' regular_balanced_text ')'
    | '[' regular_balanced_text ']'
    | '{' regular_balanced_text '}'
    ;

single_regular_balanced_text_character
    : '<Any character except / (U+002F), @ (U+0040), \" (U+0022), $ (U+0024), ( (U+0028), ) (U+0029), [
(U+005B), ] (U+005D), { (U+007B), } (U+007D) and new_line_character>'
    | '</ (U+002F), if not directly followed by / (U+002F) or * (U+002A)>'
    ;

interpolation_format
    : ':' interpolation_format_character+
    ;

interpolation_format_character
    : '<Any character except \" (U+0022), : (U+003A), { (U+007B) and } (U+007D)>'
    ;

interpolated_verbatim_string_literal
    : interpolated_verbatim_string_whole
```

```
    | interpolated_verbatim_string_start interpolated_verbatim_string_literal_body
interpolated_verbatim_string_end
    ;

interpolated_verbatim_string_literal_body
    : verbatim_balanced_text
    | interpolated_verbatim_string_literal_body interpolated_verbatim_string_mid verbatim_balanced_text
    ;

interpolated_verbatim_string_whole
    : '@"' interpolated_verbatim_string_character* '"'
    ;

interpolated_verbatim_string_start
    : '@"' interpolated_verbatim_string_character* '{'
    ;

interpolated_verbatim_string_mid
    : interpolation_format? '}' interpolated_verbatim_string_characters_after_brace? '{'
    ;

interpolated_verbatim_string_end
    : interpolation_format? '}' interpolated_verbatim_string_characters_after_brace? '"'
    ;

interpolated_verbatim_string_characters_after_brace
    : interpolated_verbatim_string_character_no_brace
    | interpolated_verbatim_string_characters_after_brace interpolated_verbatim_string_character
    ;

interpolated_verbatim_string_character
    : single_interpolated_verbatim_string_character
    | quote_escape_sequence
    | open_brace_escape_sequence
    | close_brace_escape_sequence
    ;

interpolated_verbatim_string_character_no_brace
    : '<Any interpolated_verbatim_string_character except close_brace_escape_sequence>'
    ;

single_interpolated_verbatim_string_character
    : '<Any character except \" (U+0022), { (U+007B) and } (U+007D)>'
    ;

verbatim_balanced_text
    : verbatim_balanced_text_part+
    ;

verbatim_balanced_text_part
    : single_verbatim_balanced_text_character
    | comment
    | '@' identifier_or_keyword
    | string_literal
    | interpolated_string_literal
    | '(' verbatim_balanced_text ')'
    | '[' verbatim_balanced_text ']'
    | '{' verbatim_balanced_text '}'
    ;

single_verbatim_balanced_text_character
    : '<Any character except / (U+002F), @ (U+0040), \" (U+0022), $ (U+0024), ( (U+0028), ) (U+0029), [
(U+005B), ] (U+005D), { (U+007B) and } (U+007D)>'
    | '</ (U+002F), if not directly followed by / (U+002F) or * (U+002A)>'
    ;
```

An *interpolated_string_literal* token is reinterpreted as multiple tokens and other input elements as follows, in

order of occurrence in the *interpolated_string_literal*:

- Occurrences of the following are reinterpreted as separate individual tokens: the leading `$` sign, *interpolated_regular_string_whole*, *interpolated_regular_string_start*, *interpolated_regular_string_mid*, *interpolated_regular_string_end*, *interpolated_verbatim_string_whole*, *interpolated_verbatim_string_start*, *interpolated_verbatim_string_mid* and *interpolated_verbatim_string_end*.
- Occurrences of *regular_balanced_text* and *verbatim_balanced_text* between these are reprocessed as an *input_section* (Lexical analysis) and are reinterpreted as the resulting sequence of input elements. These may in turn include interpolated string literal tokens to be reinterpreted.

Syntactic analysis will recombine the tokens into an *interpolated_string_expression* (Interpolated strings).

Examples TODO

The null literal

```
null_literal
    : 'null'
    ;
```

The *null_literal* can be implicitly converted to a reference type or nullable type.

Operators and punctuators

There are several kinds of operators and punctuators. Operators are used in expressions to describe operations involving one or more operands. For example, the expression `a + b` uses the `+` operator to add the two operands `a` and `b`. Punctuators are for grouping and separating.

```
operator_or_punctuator
    : '{'  | '}'  | '['  | ']'  | '('   | ')'  | '.'  | ','  | ':'  | ';'
    | '+'  | '-'  | '*'  | '/'  | '%'   | '&'  | '|'  | '^'  | '!'  | '~'
    | '='  | '<'  | '>'  | '?'  | '??'  | '::' | '++' | '--' | '&&' | '||'
    | '->' | '==' | '!=' | '<=' | '>='  | '+=' | '-=' | '*=' | '/=' | '%='
    | '&=' | '|=' | '^=' | '<<' | '<<=' | '=>'
    ;

right_shift
    : '>>'
    ;

right_shift_assignment
    : '>>='
    ;
```

The vertical bar in the *right_shift* and *right_shift_assignment* productions are used to indicate that, unlike other productions in the syntactic grammar, no characters of any kind (not even whitespace) are allowed between the tokens. These productions are treated specially in order to enable the correct handling of *type_parameter_list*s (Type parameters).

Pre-processing directives

The pre-processing directives provide the ability to conditionally skip sections of source files, to report error and warning conditions, and to delineate distinct regions of source code. The term "pre-processing directives" is used only for consistency with the C and C++ programming languages. In C#, there is no separate pre-processing step; pre-processing directives are processed as part of the lexical analysis phase.

```
pp_directive
    : pp_declaration
    | pp_conditional
    | pp_line
    | pp_diagnostic
    | pp_region
    | pp_pragma
    ;
```

The following pre-processing directives are available:

- `#define` and `#undef`, which are used to define and undefine, respectively, conditional compilation symbols (Declaration directives).
- `#if`, `#elif`, `#else`, and `#endif`, which are used to conditionally skip sections of source code (Conditional compilation directives).
- `#line`, which is used to control line numbers emitted for errors and warnings (Line directives).
- `#error` and `#warning`, which are used to issue errors and warnings, respectively (Diagnostic directives).
- `#region` and `#endregion`, which are used to explicitly mark sections of source code (Region directives).
- `#pragma`, which is used to specify optional contextual information to the compiler (Pragma directives).

A pre-processing directive always occupies a separate line of source code and always begins with a `#` character and a pre-processing directive name. White space may occur before the `#` character and between the `#` character and the directive name.

A source line containing a `#define`, `#undef`, `#if`, `#elif`, `#else`, `#endif`, `#line`, or `#endregion` directive may end with a single-line comment. Delimited comments (the `/* */` style of comments) are not permitted on source lines containing pre-processing directives.

Pre-processing directives are not tokens and are not part of the syntactic grammar of C#. However, pre-processing directives can be used to include or exclude sequences of tokens and can in that way affect the meaning of a C# program. For example, when compiled, the program:

```
#define A
#undef B

class C
{
#if A
    void F() {}
#else
    void G() {}
#endif

#if B
    void H() {}
#else
    void I() {}
#endif
}
```

results in the exact same sequence of tokens as the program:

```
class C
{
    void F() {}
    void I() {}
}
```

Thus, whereas lexically, the two programs are quite different, syntactically, they are identical.

Conditional compilation symbols

The conditional compilation functionality provided by the `#if`, `#elif`, `#else`, and `#endif` directives is controlled through pre-processing expressions (Pre-processing expressions) and conditional compilation symbols.

```
conditional_symbol
    : '<Any identifier_or_keyword except true or false>'
    ;
```

A conditional compilation symbol has two possible states: ***defined*** or ***undefined***. At the beginning of the lexical processing of a source file, a conditional compilation symbol is undefined unless it has been explicitly defined by an external mechanism (such as a command-line compiler option). When a `#define` directive is processed, the conditional compilation symbol named in that directive becomes defined in that source file. The symbol remains defined until an `#undef` directive for that same symbol is processed, or until the end of the source file is reached. An implication of this is that `#define` and `#undef` directives in one source file have no effect on other source files in the same program.

When referenced in a pre-processing expression, a defined conditional compilation symbol has the boolean value `true`, and an undefined conditional compilation symbol has the boolean value `false`. There is no requirement that conditional compilation symbols be explicitly declared before they are referenced in pre-processing expressions. Instead, undeclared symbols are simply undefined and thus have the value `false`.

The name space for conditional compilation symbols is distinct and separate from all other named entities in a C# program. Conditional compilation symbols can only be referenced in `#define` and `#undef` directives and in pre-processing expressions.

Pre-processing expressions

Pre-processing expressions can occur in `#if` and `#elif` directives. The operators `!`, `==`, `!=`, `&&` and `||` are permitted in pre-processing expressions, and parentheses may be used for grouping.

```
pp_expression
    : whitespace? pp_or_expression whitespace?
    ;

pp_or_expression
    : pp_and_expression
    | pp_or_expression whitespace? '||' whitespace? pp_and_expression
    ;

pp_and_expression
    : pp_equality_expression
    | pp_and_expression whitespace? '&&' whitespace? pp_equality_expression
    ;

pp_equality_expression
    : pp_unary_expression
    | pp_equality_expression whitespace? '==' whitespace? pp_unary_expression
    | pp_equality_expression whitespace? '!=' whitespace? pp_unary_expression
    ;

pp_unary_expression
    : pp_primary_expression
    | '!' whitespace? pp_unary_expression
    ;

pp_primary_expression
    : 'true'
    | 'false'
    | conditional_symbol
    | '(' whitespace? pp_expression whitespace? ')'
    ;
```

When referenced in a pre-processing expression, a defined conditional compilation symbol has the boolean value `true`, and an undefined conditional compilation symbol has the boolean value `false`.

Evaluation of a pre-processing expression always yields a boolean value. The rules of evaluation for a pre-processing expression are the same as those for a constant expression (Constant expressions), except that the only user-defined entities that can be referenced are conditional compilation symbols.

Declaration directives

The declaration directives are used to define or undefine conditional compilation symbols.

```
pp_declaration
    : whitespace? '#' whitespace? 'define' whitespace conditional_symbol pp_new_line
    | whitespace? '#' whitespace? 'undef' whitespace conditional_symbol pp_new_line
    ;

pp_new_line
    : whitespace? single_line_comment? new_line
    ;
```

The processing of a `#define` directive causes the given conditional compilation symbol to become defined, starting with the source line that follows the directive. Likewise, the processing of an `#undef` directive causes the given conditional compilation symbol to become undefined, starting with the source line that follows the directive.

Any `#define` and `#undef` directives in a source file must occur before the first *token* (Tokens) in the source file; otherwise a compile-time error occurs. In intuitive terms, `#define` and `#undef` directives must precede any "real code" in the source file.

The example:

```
#define Enterprise

#if Professional || Enterprise
    #define Advanced
#endif

namespace Megacorp.Data
{
    #if Advanced
    class PivotTable {...}
    #endif
}
```

is valid because the `#define` directives precede the first token (the `namespace` keyword) in the source file.

The following example results in a compile-time error because a `#define` follows real code:

```
#define A
namespace N
{
    #define B
    #if B
    class Class1 {}
    #endif
}
```

A `#define` may define a conditional compilation symbol that is already defined, without there being any intervening `#undef` for that symbol. The example below defines a conditional compilation symbol `A` and then defines it again.

```
#define A
#define A
```

A `#undef` may "undefine" a conditional compilation symbol that is not defined. The example below defines a conditional compilation symbol `A` and then undefines it twice; although the second `#undef` has no effect, it is still valid.

```
#define A
#undef A
#undef A
```

Conditional compilation directives

The conditional compilation directives are used to conditionally include or exclude portions of a source file.

```
pp_conditional
    : pp_if_section pp_elif_section* pp_else_section? pp_endif
    ;

pp_if_section
    : whitespace? '#' whitespace? 'if' whitespace pp_expression pp_new_line conditional_section?
    ;

pp_elif_section
    : whitespace? '#' whitespace? 'elif' whitespace pp_expression pp_new_line conditional_section?
    ;

pp_else_section:
    | whitespace? '#' whitespace? 'else' pp_new_line conditional_section?
    ;

pp_endif
    : whitespace? '#' whitespace? 'endif' pp_new_line
    ;

conditional_section
    : input_section
    | skipped_section
    ;

skipped_section
    : skipped_section_part+
    ;

skipped_section_part
    : skipped_characters? new_line
    | pp_directive
    ;

skipped_characters
    : whitespace? not_number_sign input_character*
    ;

not_number_sign
    : '<Any input_character except #>'
    ;
```

As indicated by the syntax, conditional compilation directives must be written as sets consisting of, in order, an `#if` directive, zero or more `#elif` directives, zero or one `#else` directive, and an `#endif` directive. Between the directives are conditional sections of source code. Each section is controlled by the immediately preceding directive. A conditional section may itself contain nested conditional compilation directives provided these directives form complete sets.

A *pp_conditional* selects at most one of the contained *conditional_section*s for normal lexical processing:

- The *pp_expression*s of the `#if` and `#elif` directives are evaluated in order until one yields `true`. If an expression yields `true`, the *conditional_section* of the corresponding directive is selected.
- If all *pp_expression*s yield `false`, and if an `#else` directive is present, the *conditional_section* of the `#else` directive is selected.
- Otherwise, no *conditional_section* is selected.

The selected *conditional_section*, if any, is processed as a normal *input_section*: the source code contained in the section must adhere to the lexical grammar; tokens are generated from the source code in the section; and pre-processing directives in the section have the prescribed effects.

The remaining *conditional_section*s, if any, are processed as *skipped_section*s: except for pre-processing directives, the source code in the section need not adhere to the lexical grammar; no tokens are generated from

the source code in the section; and pre-processing directives in the section must be lexically correct but are not otherwise processed. Within a *conditional_section* that is being processed as a *skipped_section*, any nested *conditional_section*s (contained in nested `#if` ... `#endif` and `#region` ... `#endregion` constructs) are also processed as *skipped_section*s.

The following example illustrates how conditional compilation directives can nest:

```
#define Debug       // Debugging on
#undef Trace        // Tracing off

class PurchaseTransaction
{
    void Commit() {
        #if Debug
            CheckConsistency();
            #if Trace
                WriteToLog(this.ToString());
            #endif
        #endif
        CommitHelper();
    }
}
```

Except for pre-processing directives, skipped source code is not subject to lexical analysis. For example, the following is valid despite the unterminated comment in the `#else` section:

```
#define Debug        // Debugging on

class PurchaseTransaction
{
    void Commit() {
        #if Debug
            CheckConsistency();
        #else
            /* Do something else
        #endif
    }
}
```

Note, however, that pre-processing directives are required to be lexically correct even in skipped sections of source code.

Pre-processing directives are not processed when they appear inside multi-line input elements. For example, the program:

```
class Hello
{
    static void Main() {
        System.Console.WriteLine(@"hello,
#if Debug
        world
#else
        Nebraska
#endif
        ");
    }
}
```

results in the output:

```
hello,
#if Debug
        world
#else
        Nebraska
#endif
```

In peculiar cases, the set of pre-processing directives that is processed might depend on the evaluation of the *pp_expression*. The example:

```
#if X
    /*
#else
    /* */ class Q { }
#endif
```

always produces the same token stream (`class` `Q` `{` `}`), regardless of whether or not `X` is defined. If `X` is defined, the only processed directives are `#if` and `#endif`, due to the multi-line comment. If `X` is undefined, then three directives (`#if` , `#else` , `#endif`) are part of the directive set.

Diagnostic directives

The diagnostic directives are used to explicitly generate error and warning messages that are reported in the same way as other compile-time errors and warnings.

```
pp_diagnostic
    : whitespace? '#' whitespace? 'error' pp_message
    | whitespace? '#' whitespace? 'warning' pp_message
    ;

pp_message
    : new_line
    | whitespace input_character* new_line
    ;
```

The example:

```
#warning Code review needed before check-in

#if Debug && Retail
    #error A build can't be both debug and retail
#endif

class Test {...}
```

always produces a warning ("Code review needed before check-in"), and produces a compile-time error ("A build can't be both debug and retail") if the conditional symbols `Debug` and `Retail` are both defined. Note that a *pp_message* can contain arbitrary text; specifically, it need not contain well-formed tokens, as shown by the single quote in the word `can't` .

Region directives

The region directives are used to explicitly mark regions of source code.

```
pp_region
    : pp_start_region conditional_section? pp_end_region
    ;

pp_start_region
    : whitespace? '#' whitespace? 'region' pp_message
    ;

pp_end_region
    : whitespace? '#' whitespace? 'endregion' pp_message
    ;
```

No semantic meaning is attached to a region; regions are intended for use by the programmer or by automated tools to mark a section of source code. The message specified in a `#region` or `#endregion` directive likewise has no semantic meaning; it merely serves to identify the region. Matching `#region` and `#endregion` directives may have different *pp_message*s.

The lexical processing of a region:

```
#region
...
#endregion
```

corresponds exactly to the lexical processing of a conditional compilation directive of the form:

```
#if true
...
#endif
```

Line directives

Line directives may be used to alter the line numbers and source file names that are reported by the compiler in output such as warnings and errors, and that are used by caller info attributes (Caller info attributes).

Line directives are most commonly used in meta-programming tools that generate C# source code from some other text input.

```
pp_line
    : whitespace? '#' whitespace? 'line' whitespace line_indicator pp_new_line
    ;

line_indicator
    : decimal_digit+ whitespace file_name
    | decimal_digit+
    | 'default'
    | 'hidden'
    ;

file_name
    : '"' file_name_character+ '"'
    ;

file_name_character
    : '<Any input_character except ">'
    ;
```

When no `#line` directives are present, the compiler reports true line numbers and source file names in its output. When processing a `#line` directive that includes a *line_indicator* that is not `default`, the compiler treats the line after the directive as having the given line number (and file name, if specified).

A `#line default` directive reverses the effect of all preceding #line directives. The compiler reports true line information for subsequent lines, precisely as if no `#line` directives had been processed.

A `#line hidden` directive has no effect on the file and line numbers reported in error messages, but does affect source level debugging. When debugging, all lines between a `#line hidden` directive and the subsequent `#line` directive (that is not `#line hidden`) have no line number information. When stepping through code in the debugger, these lines will be skipped entirely.

Note that a *file_name* differs from a regular string literal in that escape characters are not processed; the "`\`" character simply designates an ordinary backslash character within a *file_name*.

Pragma directives

The `#pragma` preprocessing directive is used to specify optional contextual information to the compiler. The information supplied in a `#pragma` directive will never change program semantics.

```
pp_pragma
    : whitespace? '#' whitespace? 'pragma' whitespace pragma_body pp_new_line
    ;

pragma_body
    : pragma_warning_body
    ;
```

C# provides `#pragma` directives to control compiler warnings. Future versions of the language may include additional `#pragma` directives. To ensure interoperability with other C# compilers, the Microsoft C# compiler does not issue compilation errors for unknown `#pragma` directives; such directives do however generate warnings.

Pragma warning

The `#pragma warning` directive is used to disable or restore all or a particular set of warning messages during compilation of the subsequent program text.

```
pragma_warning_body
    : 'warning' whitespace warning_action
    | 'warning' whitespace warning_action whitespace warning_list
    ;

warning_action
    : 'disable'
    | 'restore'
    ;

warning_list
    : decimal_digit+ (whitespace? ',' whitespace? decimal_digit+)*
    ;
```

A `#pragma warning` directive that omits the warning list affects all warnings. A `#pragma warning` directive that includes a warning list affects only those warnings that are specified in the list.

A `#pragma warning disable` directive disables all or the given set of warnings.

A `#pragma warning restore` directive restores all or the given set of warnings to the state that was in effect at the beginning of the compilation unit. Note that if a particular warning was disabled externally, a `#pragma warning restore` (whether for all or the specific warning) will not re-enable that warning.

The following example shows use of `#pragma warning` to temporarily disable the warning reported when obsoleted members are referenced, using the warning number from the Microsoft C# compiler.

```
using System;

class Program
{
    [Obsolete]
    static void Foo() {}

    static void Main() {
#pragma warning disable 612
    Foo();
#pragma warning restore 612
    }
}
```

Basic concepts

18.12.2021 • 48 minutes to read

Application Startup

An assembly that has an ***entry point*** is called an ***application***. When an application is run, a new ***application domain*** is created. Several different instantiations of an application may exist on the same machine at the same time, and each has its own application domain.

An application domain enables application isolation by acting as a container for application state. An application domain acts as a container and boundary for the types defined in the application and the class libraries it uses. Types loaded into one application domain are distinct from the same type loaded into another application domain, and instances of objects are not directly shared between application domains. For instance, each application domain has its own copy of static variables for these types, and a static constructor for a type is run at most once per application domain. Implementations are free to provide implementation-specific policy or mechanisms for the creation and destruction of application domains.

Application startup occurs when the execution environment calls a designated method, which is referred to as the application's entry point. This entry point method is always named `Main`, and can have one of the following signatures:

```
static void Main() {...}

static void Main(string[] args) {...}

static int Main() {...}

static int Main(string[] args) {...}
```

As shown, the entry point may optionally return an `int` value. This return value is used in application termination (Application termination).

The entry point may optionally have one formal parameter. The parameter may have any name, but the type of the parameter must be `string[]`. If the formal parameter is present, the execution environment creates and passes a `string[]` argument containing the command-line arguments that were specified when the application was started. The `string[]` argument is never null, but it may have a length of zero if no command-line arguments were specified.

Since C# supports method overloading, a class or struct may contain multiple definitions of some method, provided each has a different signature. However, within a single program, no class or struct may contain more than one method called `Main` whose definition qualifies it to be used as an application entry point. Other overloaded versions of `Main` are permitted, however, provided they have more than one parameter, or their only parameter is other than type `string[]`.

An application can be made up of multiple classes or structs. It is possible for more than one of these classes or structs to contain a method called `Main` whose definition qualifies it to be used as an application entry point. In such cases, an external mechanism (such as a command-line compiler option) must be used to select one of these `Main` methods as the entry point.

In C#, every method must be defined as a member of a class or struct. Ordinarily, the declared accessibility (Declared accessibility) of a method is determined by the access modifiers (Access modifiers) specified in its declaration, and similarly the declared accessibility of a type is determined by the access modifiers specified in

its declaration. In order for a given method of a given type to be callable, both the type and the member must be accessible. However, the application entry point is a special case. Specifically, the execution environment can access the application's entry point regardless of its declared accessibility and regardless of the declared accessibility of its enclosing type declarations.

The application entry point method may not be in a generic class declaration.

In all other respects, entry point methods behave like those that are not entry points.

Application termination

Application termination returns control to the execution environment.

If the return type of the application's ***entry point*** method is `int`, the value returned serves as the application's ***termination status code***. The purpose of this code is to allow communication of success or failure to the execution environment.

If the return type of the entry point method is `void`, reaching the right brace (`}`) which terminates that method, or executing a `return` statement that has no expression, results in a termination status code of `0`.

Prior to an application's termination, destructors for all of its objects that have not yet been garbage collected are called, unless such cleanup has been suppressed (by a call to the library method `GC.SuppressFinalize`, for example).

Declarations

Declarations in a C# program define the constituent elements of the program. C# programs are organized using namespaces (Namespaces), which can contain type declarations and nested namespace declarations. Type declarations (Type declarations) are used to define classes (Classes), structs (Structs), interfaces (Interfaces), enums (Enums), and delegates (Delegates). The kinds of members permitted in a type declaration depend on the form of the type declaration. For instance, class declarations can contain declarations for constants (Constants), fields (Fields), methods (Methods), properties (Properties), events (Events), indexers (Indexers), operators (Operators), instance constructors (Instance constructors), static constructors (Static constructors), destructors (Destructors), and nested types (Nested types).

A declaration defines a name in the ***declaration space*** to which the declaration belongs. Except for overloaded members (Signatures and overloading), it is a compile-time error to have two or more declarations that introduce members with the same name in a declaration space. It is never possible for a declaration space to contain different kinds of members with the same name. For example, a declaration space can never contain a field and a method by the same name.

There are several different types of declaration spaces, as described in the following.

- Within all source files of a program, *namespace_member_declaration*s with no enclosing *namespace_declaration* are members of a single combined declaration space called the ***global declaration space***.
- Within all source files of a program, *namespace_member_declaration*s within *namespace_declaration*s that have the same fully qualified namespace name are members of a single combined declaration space.
- Each class, struct, or interface declaration creates a new declaration space. Names are introduced into this declaration space through *class_member_declaration*s, *struct_member_declaration*s, *interface_member_declaration*s, or *type_parameter*s. Except for overloaded instance constructor declarations and static constructor declarations, a class or struct cannot contain a member declaration with the same name as the class or struct. A class, struct, or interface permits the declaration of overloaded methods and indexers. Furthermore, a class or struct permits the declaration of overloaded instance constructors and operators. For example, a class, struct, or interface may contain multiple method declarations with the same

name, provided these method declarations differ in their signature (Signatures and overloading). Note that base classes do not contribute to the declaration space of a class, and base interfaces do not contribute to the declaration space of an interface. Thus, a derived class or interface is allowed to declare a member with the same name as an inherited member. Such a member is said to ***hide*** the inherited member.

- Each delegate declaration creates a new declaration space. Names are introduced into this declaration space through formal parameters (*fixed_parameter*s and *parameter_array*s) and *type_parameter*s.
- Each enumeration declaration creates a new declaration space. Names are introduced into this declaration space through *enum_member_declarations*.
- Each method declaration, indexer declaration, operator declaration, instance constructor declaration and anonymous function creates a new declaration space called a ***local variable declaration space***. Names are introduced into this declaration space through formal parameters (*fixed_parameter*s and *parameter_array*s) and *type_parameter*s. The body of the function member or anonymous function, if any, is considered to be nested within the local variable declaration space. It is an error for a local variable declaration space and a nested local variable declaration space to contain elements with the same name. Thus, within a nested declaration space it is not possible to declare a local variable or constant with the same name as a local variable or constant in an enclosing declaration space. It is possible for two declaration spaces to contain elements with the same name as long as neither declaration space contains the other.
- Each *block* or *switch_block* , as well as a *for*, *foreach* and *using* statement, creates a local variable declaration space for local variables and local constants . Names are introduced into this declaration space through *local_variable_declaration*s and *local_constant_declaration*s. Note that blocks that occur as or within the body of a function member or anonymous function are nested within the local variable declaration space declared by those functions for their parameters. Thus it is an error to have e.g. a method with a local variable and a parameter of the same name.
- Each *block* or *switch_block* creates a separate declaration space for labels. Names are introduced into this declaration space through *labeled_statement*s, and the names are referenced through *goto_statement*s. The ***label declaration space*** of a block includes any nested blocks. Thus, within a nested block it is not possible to declare a label with the same name as a label in an enclosing block.

The textual order in which names are declared is generally of no significance. In particular, textual order is not significant for the declaration and use of namespaces, constants, methods, properties, events, indexers, operators, instance constructors, destructors, static constructors, and types. Declaration order is significant in the following ways:

- Declaration order for field declarations and local variable declarations determines the order in which their initializers (if any) are executed.
- Local variables must be defined before they are used (Scopes).
- Declaration order for enum member declarations (Enum members) is significant when *constant_expression* values are omitted.

The declaration space of a namespace is "open ended", and two namespace declarations with the same fully qualified name contribute to the same declaration space. For example

```
namespace Megacorp.Data
{
    class Customer
    {
        ...
    }
}

namespace Megacorp.Data
{
    class Order
    {
        ...
    }
}
```

The two namespace declarations above contribute to the same declaration space, in this case declaring two classes with the fully qualified names `Megacorp.Data.Customer` and `Megacorp.Data.Order`. Because the two declarations contribute to the same declaration space, it would have caused a compile-time error if each contained a declaration of a class with the same name.

As specified above, the declaration space of a block includes any nested blocks. Thus, in the following example, the `F` and `G` methods result in a compile-time error because the name `i` is declared in the outer block and cannot be redeclared in the inner block. However, the `H` and `I` methods are valid since the two `i`'s are declared in separate non-nested blocks.

```
class A
{
    void F() {
        int i = 0;
        if (true) {
            int i = 1;
        }
    }

    void G() {
        if (true) {
            int i = 0;
        }
        int i = 1;
    }

    void H() {
        if (true) {
            int i = 0;
        }
        if (true) {
            int i = 1;
        }
    }

    void I() {
        for (int i = 0; i < 10; i++)
            H();
        for (int i = 0; i < 10; i++)
            H();
    }
}
```

Members

Namespaces and types have ***members***. The members of an entity are generally available through the use of a qualified name that starts with a reference to the entity, followed by a "`.`" token, followed by the name of the member.

Members of a type are either declared in the type declaration or ***inherited*** from the base class of the type. When a type inherits from a base class, all members of the base class, except instance constructors, destructors and static constructors, become members of the derived type. The declared accessibility of a base class member does not control whether the member is inherited—inheritance extends to any member that isn't an instance constructor, static constructor, or destructor. However, an inherited member may not be accessible in a derived type, either because of its declared accessibility (Declared accessibility) or because it is hidden by a declaration in the type itself (Hiding through inheritance).

Namespace members

Namespaces and types that have no enclosing namespace are members of the ***global namespace***. This corresponds directly to the names declared in the global declaration space.

Namespaces and types declared within a namespace are members of that namespace. This corresponds directly to the names declared in the declaration space of the namespace.

Namespaces have no access restrictions. It is not possible to declare private, protected, or internal namespaces, and namespace names are always publicly accessible.

Struct members

The members of a struct are the members declared in the struct and the members inherited from the struct's direct base class `System.ValueType` and the indirect base class `object`.

The members of a simple type correspond directly to the members of the struct type aliased by the simple type:

- The members of `sbyte` are the members of the `System.SByte` struct.
- The members of `byte` are the members of the `System.Byte` struct.
- The members of `short` are the members of the `System.Int16` struct.
- The members of `ushort` are the members of the `System.UInt16` struct.
- The members of `int` are the members of the `System.Int32` struct.
- The members of `uint` are the members of the `System.UInt32` struct.
- The members of `long` are the members of the `System.Int64` struct.
- The members of `ulong` are the members of the `System.UInt64` struct.
- The members of `char` are the members of the `System.Char` struct.
- The members of `float` are the members of the `System.Single` struct.
- The members of `double` are the members of the `System.Double` struct.
- The members of `decimal` are the members of the `System.Decimal` struct.
- The members of `bool` are the members of the `System.Boolean` struct.

Enumeration members

The members of an enumeration are the constants declared in the enumeration and the members inherited from the enumeration's direct base class `System.Enum` and the indirect base classes `System.ValueType` and `object`.

Class members

The members of a class are the members declared in the class and the members inherited from the base class (except for class `object` which has no base class). The members inherited from the base class include the constants, fields, methods, properties, events, indexers, operators, and types of the base class, but not the instance constructors, destructors and static constructors of the base class. Base class members are inherited without regard to their accessibility.

A class declaration may contain declarations of constants, fields, methods, properties, events, indexers, operators, instance constructors, destructors, static constructors and types.

The members of `object` and `string` correspond directly to the members of the class types they alias:

- The members of `object` are the members of the `System.Object` class.
- The members of `string` are the members of the `System.String` class.

Interface members

The members of an interface are the members declared in the interface and in all base interfaces of the interface. The members in class `object` are not, strictly speaking, members of any interface (Interface members). However, the members in class `object` are available via member lookup in any interface type (Member lookup).

Array members

The members of an array are the members inherited from class `System.Array`.

Delegate members

The members of a delegate are the members inherited from class `System.Delegate`.

Member access

Declarations of members allow control over member access. The accessibility of a member is established by the declared accessibility (Declared accessibility) of the member combined with the accessibility of the immediately containing type, if any.

When access to a particular member is allowed, the member is said to be ***accessible***. Conversely, when access to a particular member is disallowed, the member is said to be ***inaccessible***. Access to a member is permitted when the textual location in which the access takes place is included in the accessibility domain (Accessibility domains) of the member.

Declared accessibility

The *declared accessibility* of a member can be one of the following:

- Public, which is selected by including a `public` modifier in the member declaration. The intuitive meaning of `public` is "access not limited".
- Protected, which is selected by including a `protected` modifier in the member declaration. The intuitive meaning of `protected` is "access limited to the containing class or types derived from the containing class".
- Internal, which is selected by including an `internal` modifier in the member declaration. The intuitive meaning of `internal` is "access limited to this program".
- Protected internal (meaning protected or internal), which is selected by including both a `protected` and an `internal` modifier in the member declaration. The intuitive meaning of `protected internal` is "access limited to this program or types derived from the containing class".
- Private, which is selected by including a `private` modifier in the member declaration. The intuitive meaning of `private` is "access limited to the containing type".

Depending on the context in which a member declaration takes place, only certain types of declared accessibility are permitted. Furthermore, when a member declaration does not include any access modifiers, the context in which the declaration takes place determines the default declared accessibility.

- Namespaces implicitly have `public` declared accessibility. No access modifiers are allowed on namespace declarations.
- Types declared in compilation units or namespaces can have `public` or `internal` declared accessibility and default to `internal` declared accessibility.
- Class members can have any of the five kinds of declared accessibility and default to `private` declared

accessibility. (Note that a type declared as a member of a class can have any of the five kinds of declared accessibility, whereas a type declared as a member of a namespace can have only `public` or `internal` declared accessibility.)

- Struct members can have `public`, `internal`, or `private` declared accessibility and default to `private` declared accessibility because structs are implicitly sealed. Struct members introduced in a struct (that is, not inherited by that struct) cannot have `protected` or `protected internal` declared accessibility. (Note that a type declared as a member of a struct can have `public`, `internal`, or `private` declared accessibility, whereas a type declared as a member of a namespace can have only `public` or `internal` declared accessibility.)
- Interface members implicitly have `public` declared accessibility. No access modifiers are allowed on interface member declarations.
- Enumeration members implicitly have `public` declared accessibility. No access modifiers are allowed on enumeration member declarations.

Accessibility domains

The ***accessibility domain*** of a member consists of the (possibly disjoint) sections of program text in which access to the member is permitted. For purposes of defining the accessibility domain of a member, a member is said to be ***top-level*** if it is not declared within a type, and a member is said to be ***nested*** if it is declared within another type. Furthermore, the ***program text*** of a program is defined as all program text contained in all source files of the program, and the program text of a type is defined as all program text contained in the *type_declaration*s of that type (including, possibly, types that are nested within the type).

The accessibility domain of a predefined type (such as `object`, `int`, or `double`) is unlimited.

The accessibility domain of a top-level unbound type `T` (Bound and unbound types) that is declared in a program `P` is defined as follows:

- If the declared accessibility of `T` is `public`, the accessibility domain of `T` is the program text of `P` and any program that references `P`.
- If the declared accessibility of `T` is `internal`, the accessibility domain of `T` is the program text of `P`.

From these definitions it follows that the accessibility domain of a top-level unbound type is always at least the program text of the program in which that type is declared.

The accessibility domain for a constructed type `T<A1, ..., An>` is the intersection of the accessibility domain of the unbound generic type `T` and the accessibility domains of the type arguments `A1, ..., An`.

The accessibility domain of a nested member `M` declared in a type `T` within a program `P` is defined as follows (noting that `M` itself may possibly be a type):

- If the declared accessibility of `M` is `public`, the accessibility domain of `M` is the accessibility domain of `T`.
- If the declared accessibility of `M` is `protected internal`, let `D` be the union of the program text of `P` and the program text of any type derived from `T`, which is declared outside `P`. The accessibility domain of `M` is the intersection of the accessibility domain of `T` with `D`.
- If the declared accessibility of `M` is `protected`, let `D` be the union of the program text of `T` and the program text of any type derived from `T`. The accessibility domain of `M` is the intersection of the accessibility domain of `T` with `D`.
- If the declared accessibility of `M` is `internal`, the accessibility domain of `M` is the intersection of the accessibility domain of `T` with the program text of `P`.
- If the declared accessibility of `M` is `private`, the accessibility domain of `M` is the program text of `T`.

From these definitions it follows that the accessibility domain of a nested member is always at least the program text of the type in which the member is declared. Furthermore, it follows that the accessibility domain of a member is never more inclusive than the accessibility domain of the type in which the member is declared.

In intuitive terms, when a type or member `M` is accessed, the following steps are evaluated to ensure that the access is permitted:

- First, if `M` is declared within a type (as opposed to a compilation unit or a namespace), a compile-time error occurs if that type is not accessible.
- Then, if `M` is `public`, the access is permitted.
- Otherwise, if `M` is `protected internal`, the access is permitted if it occurs within the program in which `M` is declared, or if it occurs within a class derived from the class in which `M` is declared and takes place through the derived class type (Protected access for instance members).
- Otherwise, if `M` is `protected`, the access is permitted if it occurs within the class in which `M` is declared, or if it occurs within a class derived from the class in which `M` is declared and takes place through the derived class type (Protected access for instance members).
- Otherwise, if `M` is `internal`, the access is permitted if it occurs within the program in which `M` is declared.
- Otherwise, if `M` is `private`, the access is permitted if it occurs within the type in which `M` is declared.
- Otherwise, the type or member is inaccessible, and a compile-time error occurs.

In the example

```
public class A
{
    public static int X;
    internal static int Y;
    private static int Z;
}

internal class B
{
    public static int X;
    internal static int Y;
    private static int Z;

    public class C
    {
        public static int X;
        internal static int Y;
        private static int Z;
    }

    private class D
    {
        public static int X;
        internal static int Y;
        private static int Z;
    }
}
```

the classes and members have the following accessibility domains:

- The accessibility domain of `A` and `A.X` is unlimited.
- The accessibility domain of `A.Y`, `B`, `B.X`, `B.Y`, `B.C`, `B.C.X`, and `B.C.Y` is the program text of the containing program.
- The accessibility domain of `A.Z` is the program text of `A`.
- The accessibility domain of `B.Z` and `B.D` is the program text of `B`, including the program text of `B.C` and `B.D`.
- The accessibility domain of `B.C.Z` is the program text of `B.C`.
- The accessibility domain of `B.D.X` and `B.D.Y` is the program text of `B`, including the program text of `B.C`

and `B.D`.

- The accessibility domain of `B.D.Z` is the program text of `B.D`.

As the example illustrates, the accessibility domain of a member is never larger than that of a containing type. For example, even though all `X` members have public declared accessibility, all but `A.X` have accessibility domains that are constrained by a containing type.

As described in Members, all members of a base class, except for instance constructors, destructors and static constructors, are inherited by derived types. This includes even private members of a base class. However, the accessibility domain of a private member includes only the program text of the type in which the member is declared. In the example

```
class A
{
    int x;

    static void F(B b) {
        b.x = 1;        // Ok
    }
}

class B: A
{
    static void F(B b) {
        b.x = 1;        // Error, x not accessible
    }
}
```

the `B` class inherits the private member `x` from the `A` class. Because the member is private, it is only accessible within the *class_body* of `A`. Thus, the access to `b.x` succeeds in the `A.F` method, but fails in the `B.F` method.

Protected access for instance members

When a `protected` instance member is accessed outside the program text of the class in which it is declared, and when a `protected internal` instance member is accessed outside the program text of the program in which it is declared, the access must take place within a class declaration that derives from the class in which it is declared. Furthermore, the access is required to take place through an instance of that derived class type or a class type constructed from it. This restriction prevents one derived class from accessing protected members of other derived classes, even when the members are inherited from the same base class.

Let `B` be a base class that declares a protected instance member `M`, and let `D` be a class that derives from `B`. Within the *class_body* of `D`, access to `M` can take one of the following forms:

- An unqualified *type_name* or *primary_expression* of the form `M`.
- A *primary_expression* of the form `E.M`, provided the type of `E` is `T` or a class derived from `T`, where `T` is the class type `D`, or a class type constructed from `D`
- A *primary_expression* of the form `base.M`.

In addition to these forms of access, a derived class can access a protected instance constructor of a base class in a *constructor_initializer* (Constructor initializers).

In the example

```
public class A
{
    protected int x;

    static void F(A a, B b) {
        a.x = 1;        // Ok
        b.x = 1;        // Ok
    }
}

public class B: A
{
    static void F(A a, B b) {
        a.x = 1;        // Error, must access through instance of B
        b.x = 1;        // Ok
    }
}
```

within `A`, it is possible to access `x` through instances of both `A` and `B`, since in either case the access takes place through an instance of `A` or a class derived from `A`. However, within `B`, it is not possible to access `x` through an instance of `A`, since `A` does not derive from `B`.

In the example

```
class C<T>
{
    protected T x;
}

class D<T>: C<T>
{
    static void F() {
        D<T> dt = new D<T>();
        D<int> di = new D<int>();
        D<string> ds = new D<string>();
        dt.x = default(T);
        di.x = 123;
        ds.x = "test";
    }
}
```

the three assignments to `x` are permitted because they all take place through instances of class types constructed from the generic type.

Accessibility constraints

Several constructs in the C# language require a type to be ***at least as accessible as*** a member or another type. A type `T` is said to be at least as accessible as a member or type `M` if the accessibility domain of `T` is a superset of the accessibility domain of `M`. In other words, `T` is at least as accessible as `M` if `T` is accessible in all contexts in which `M` is accessible.

The following accessibility constraints exist:

- The direct base class of a class type must be at least as accessible as the class type itself.
- The explicit base interfaces of an interface type must be at least as accessible as the interface type itself.
- The return type and parameter types of a delegate type must be at least as accessible as the delegate type itself.
- The type of a constant must be at least as accessible as the constant itself.
- The type of a field must be at least as accessible as the field itself.
- The return type and parameter types of a method must be at least as accessible as the method itself.

- The type of a property must be at least as accessible as the property itself.
- The type of an event must be at least as accessible as the event itself.
- The type and parameter types of an indexer must be at least as accessible as the indexer itself.
- The return type and parameter types of an operator must be at least as accessible as the operator itself.
- The parameter types of an instance constructor must be at least as accessible as the instance constructor itself.

In the example

```
class A {...}

public class B: A {...}
```

the `B` class results in a compile-time error because `A` is not at least as accessible as `B`.

Likewise, in the example

```
class A {...}

public class B
{
    A F() {...}

    internal A G() {...}

    public A H() {...}
}
```

the `H` method in `B` results in a compile-time error because the return type `A` is not at least as accessible as the method.

Signatures and overloading

Methods, instance constructors, indexers, and operators are characterized by their ***signatures***:

- The signature of a method consists of the name of the method, the number of type parameters and the type and kind (value, reference, or output) of each of its formal parameters, considered in the order left to right. For these purposes, any type parameter of the method that occurs in the type of a formal parameter is identified not by its name, but by its ordinal position in the type argument list of the method. The signature of a method specifically does not include the return type, the `params` modifier that may be specified for the right-most parameter, nor the optional type parameter constraints.
- The signature of an instance constructor consists of the type and kind (value, reference, or output) of each of its formal parameters, considered in the order left to right. The signature of an instance constructor specifically does not include the `params` modifier that may be specified for the right-most parameter.
- The signature of an indexer consists of the type of each of its formal parameters, considered in the order left to right. The signature of an indexer specifically does not include the element type, nor does it include the `params` modifier that may be specified for the right-most parameter.
- The signature of an operator consists of the name of the operator and the type of each of its formal parameters, considered in the order left to right. The signature of an operator specifically does not include the result type.

Signatures are the enabling mechanism for ***overloading*** of members in classes, structs, and interfaces:

- Overloading of methods permits a class, struct, or interface to declare multiple methods with the same name, provided their signatures are unique within that class, struct, or interface.

- Overloading of instance constructors permits a class or struct to declare multiple instance constructors, provided their signatures are unique within that class or struct.
- Overloading of indexers permits a class, struct, or interface to declare multiple indexers, provided their signatures are unique within that class, struct, or interface.
- Overloading of operators permits a class or struct to declare multiple operators with the same name, provided their signatures are unique within that class or struct.

Although `out` and `ref` parameter modifiers are considered part of a signature, members declared in a single type cannot differ in signature solely by `ref` and `out`. A compile-time error occurs if two members are declared in the same type with signatures that would be the same if all parameters in both methods with `out` modifiers were changed to `ref` modifiers. For other purposes of signature matching (e.g., hiding or overriding), `ref` and `out` are considered part of the signature and do not match each other. (This restriction is to allow C# programs to be easily translated to run on the Common Language Infrastructure (CLI), which does not provide a way to define methods that differ solely in `ref` and `out`.)

For the purposes of signatures, the types `object` and `dynamic` are considered the same. Members declared in a single type can therefore not differ in signature solely by `object` and `dynamic`.

The following example shows a set of overloaded method declarations along with their signatures.

```
interface ITest
{
    void F();                       // F()

    void F(int x);                  // F(int)

    void F(ref int x);              // F(ref int)

    void F(out int x);              // F(out int)      error

    void F(int x, int y);           // F(int, int)

    int F(string s);                // F(string)

    int F(int x);                   // F(int)          error

    void F(string[] a);             // F(string[])

    void F(params string[] a);      // F(string[])     error
}
```

Note that any `ref` and `out` parameter modifiers (Method parameters) are part of a signature. Thus, `F(int)` and `F(ref int)` are unique signatures. However, `F(ref int)` and `F(out int)` cannot be declared within the same interface because their signatures differ solely by `ref` and `out`. Also, note that the return type and the `params` modifier are not part of a signature, so it is not possible to overload solely based on return type or on the inclusion or exclusion of the `params` modifier. As such, the declarations of the methods `F(int)` and `F(params string[])` identified above result in a compile-time error.

Scopes

The ***scope*** of a name is the region of program text within which it is possible to refer to the entity declared by the name without qualification of the name. Scopes can be ***nested***, and an inner scope may redeclare the meaning of a name from an outer scope (this does not, however, remove the restriction imposed by Declarations that within a nested block it is not possible to declare a local variable with the same name as a local variable in an enclosing block). The name from the outer scope is then said to be ***hidden*** in the region of program text covered by the inner scope, and access to the outer name is only possible by qualifying the name.

- The scope of a namespace member declared by a *namespace_member_declaration* (Namespace members) with no enclosing *namespace_declaration* is the entire program text.
- The scope of a namespace member declared by a *namespace_member_declaration* within a *namespace_declaration* whose fully qualified name is `N` is the *namespace_body* of every *namespace_declaration* whose fully qualified name is `N` or starts with `N`, followed by a period.
- The scope of name defined by an *extern_alias_directive* extends over the *using_directive*s, *global_attributes* and *namespace_member_declaration*s of its immediately containing compilation unit or namespace body. An *extern_alias_directive* does not contribute any new members to the underlying declaration space. In other words, an *extern_alias_directive* is not transitive, but, rather, affects only the compilation unit or namespace body in which it occurs.
- The scope of a name defined or imported by a *using_directive* (Using directives) extends over the *namespace_member_declaration*s of the *compilation_unit* or *namespace_body* in which the *using_directive* occurs. A *using_directive* may make zero or more namespace, type or member names available within a particular *compilation_unit* or *namespace_body*, but does not contribute any new members to the underlying declaration space. In other words, a *using_directive* is not transitive but rather affects only the *compilation_unit* or *namespace_body* in which it occurs.
- The scope of a type parameter declared by a *type_parameter_list* on a *class_declaration* (Class declarations) is the *class_base*, *type_parameter_constraints_clause*s, and *class_body* of that *class_declaration*.
- The scope of a type parameter declared by a *type_parameter_list* on a *struct_declaration* (Struct declarations) is the *struct_interfaces*, *type_parameter_constraints_clause*s, and *struct_body* of that *struct_declaration*.
- The scope of a type parameter declared by a *type_parameter_list* on an *interface_declaration* (Interface declarations) is the *interface_base*, *type_parameter_constraints_clause*s, and *interface_body* of that *interface_declaration*.
- The scope of a type parameter declared by a *type_parameter_list* on a *delegate_declaration* (Delegate declarations) is the *return_type*, *formal_parameter_list*, and *type_parameter_constraints_clause*s of that *delegate_declaration*.
- The scope of a member declared by a *class_member_declaration* (Class body) is the *class_body* in which the declaration occurs. In addition, the scope of a class member extends to the *class_body* of those derived classes that are included in the accessibility domain (Accessibility domains) of the member.
- The scope of a member declared by a *struct_member_declaration* (Struct members) is the *struct_body* in which the declaration occurs.
- The scope of a member declared by an *enum_member_declaration* (Enum members) is the *enum_body* in which the declaration occurs.
- The scope of a parameter declared in a *method_declaration* (Methods) is the *method_body* of that *method_declaration*.
- The scope of a parameter declared in an *indexer_declaration* (Indexers) is the *accessor_declarations* of that *indexer_declaration*.
- The scope of a parameter declared in an *operator_declaration* (Operators) is the *block* of that *operator_declaration*.
- The scope of a parameter declared in a *constructor_declaration* (Instance constructors) is the *constructor_initializer* and *block* of that *constructor_declaration*.
- The scope of a parameter declared in a *lambda_expression* (Anonymous function expressions) is the *anonymous_function_body* of that *lambda_expression*
- The scope of a parameter declared in an *anonymous_method_expression* (Anonymous function expressions) is the *block* of that *anonymous_method_expression*.
- The scope of a label declared in a *labeled_statement* (Labeled statements) is the *block* in which the declaration occurs.
- The scope of a local variable declared in a *local_variable_declaration* (Local variable declarations) is the block in which the declaration occurs.

- The scope of a local variable declared in a *switch_block* of a `switch` statement (The switch statement) is the *switch_block*.
- The scope of a local variable declared in a *for_initializer* of a `for` statement (The for statement) is the *for_initializer*, the *for_condition*, the *for_iterator*, and the contained *statement* of the `for` statement.
- The scope of a local constant declared in a *local_constant_declaration* (Local constant declarations) is the block in which the declaration occurs. It is a compile-time error to refer to a local constant in a textual position that precedes its *constant_declarator*.
- The scope of a variable declared as part of a *foreach_statement*, *using_statement*, *lock_statement* or *query_expression* is determined by the expansion of the given construct.

Within the scope of a namespace, class, struct, or enumeration member it is possible to refer to the member in a textual position that precedes the declaration of the member. For example

```
class A
{
    void F() {
        i = 1;
    }

    int i = 0;
}
```

Here, it is valid for `F` to refer to `i` before it is declared.

Within the scope of a local variable, it is a compile-time error to refer to the local variable in a textual position that precedes the *local_variable_declarator* of the local variable. For example

```
class A
{
    int i = 0;

    void F() {
        i = 1;                  // Error, use precedes declaration
        int i;
        i = 2;
    }

    void G() {
        int j = (j = 1);        // Valid
    }

    void H() {
        int a = 1, b = ++a;     // Valid
    }
}
```

In the `F` method above, the first assignment to `i` specifically does not refer to the field declared in the outer scope. Rather, it refers to the local variable and it results in a compile-time error because it textually precedes the declaration of the variable. In the `G` method, the use of `j` in the initializer for the declaration of `j` is valid because the use does not precede the *local_variable_declarator*. In the `H` method, a subsequent *local_variable_declarator* correctly refers to a local variable declared in an earlier *local_variable_declarator* within the same *local_variable_declaration*.

The scoping rules for local variables are designed to guarantee that the meaning of a name used in an expression context is always the same within a block. If the scope of a local variable were to extend only from its declaration to the end of the block, then in the example above, the first assignment would assign to the instance variable and the second assignment would assign to the local variable, possibly leading to compile-time errors if

the statements of the block were later to be rearranged.

The meaning of a name within a block may differ based on the context in which the name is used. In the example

```
using System;

class A {}

class Test
{
    static void Main() {
        string A = "hello, world";
        string s = A;                           // expression context

        Type t = typeof(A);                     // type context

        Console.WriteLine(s);                   // writes "hello, world"
        Console.WriteLine(t);                   // writes "A"
    }
}
```

the name `A` is used in an expression context to refer to the local variable `A` and in a type context to refer to the class `A`.

Name hiding

The scope of an entity typically encompasses more program text than the declaration space of the entity. In particular, the scope of an entity may include declarations that introduce new declaration spaces containing entities of the same name. Such declarations cause the original entity to become ***hidden***. Conversely, an entity is said to be ***visible*** when it is not hidden.

Name hiding occurs when scopes overlap through nesting and when scopes overlap through inheritance. The characteristics of the two types of hiding are described in the following sections.

Hiding through nesting

Name hiding through nesting can occur as a result of nesting namespaces or types within namespaces, as a result of nesting types within classes or structs, and as a result of parameter and local variable declarations.

In the example

```
class A
{
    int i = 0;

    void F() {
        int i = 1;
    }

    void G() {
        i = 1;
    }
}
```

within the `F` method, the instance variable `i` is hidden by the local variable `i`, but within the `G` method, `i` still refers to the instance variable.

When a name in an inner scope hides a name in an outer scope, it hides all overloaded occurrences of that name. In the example

```
class Outer
{
    static void F(int i) {}

    static void F(string s) {}

    class Inner
    {
        void G() {
            F(1);              // Invokes Outer.Inner.F
            F("Hello");        // Error
        }

        static void F(long l) {}
    }
}
```

the call `F(1)` invokes the `F` declared in `Inner` because all outer occurrences of `F` are hidden by the inner declaration. For the same reason, the call `F("Hello")` results in a compile-time error.

Hiding through inheritance

Name hiding through inheritance occurs when classes or structs redeclare names that were inherited from base classes. This type of name hiding takes one of the following forms:

- A constant, field, property, event, or type introduced in a class or struct hides all base class members with the same name.
- A method introduced in a class or struct hides all non-method base class members with the same name, and all base class methods with the same signature (method name and parameter count, modifiers, and types).
- An indexer introduced in a class or struct hides all base class indexers with the same signature (parameter count and types).

The rules governing operator declarations (Operators) make it impossible for a derived class to declare an operator with the same signature as an operator in a base class. Thus, operators never hide one another.

Contrary to hiding a name from an outer scope, hiding an accessible name from an inherited scope causes a warning to be reported. In the example

```
class Base
{
    public void F() {}
}

class Derived: Base
{
    public void F() {}        // Warning, hiding an inherited name
}
```

the declaration of `F` in `Derived` causes a warning to be reported. Hiding an inherited name is specifically not an error, since that would preclude separate evolution of base classes. For example, the above situation might have come about because a later version of `Base` introduced an `F` method that wasn't present in an earlier version of the class. Had the above situation been an error, then any change made to a base class in a separately versioned class library could potentially cause derived classes to become invalid.

The warning caused by hiding an inherited name can be eliminated through use of the `new` modifier:

```
class Base
{
    public void F() {}
}

class Derived: Base
{
    new public void F() {}
}
```

The `new` modifier indicates that the `F` in `Derived` is "new", and that it is indeed intended to hide the inherited member.

A declaration of a new member hides an inherited member only within the scope of the new member.

```
class Base
{
    public static void F() {}
}

class Derived: Base
{
    new private static void F() {}    // Hides Base.F in Derived only
}

class MoreDerived: Derived
{
    static void G() { F(); }          // Invokes Base.F
}
```

In the example above, the declaration of `F` in `Derived` hides the `F` that was inherited from `Base`, but since the new `F` in `Derived` has private access, its scope does not extend to `MoreDerived`. Thus, the call `F()` in `MoreDerived.G` is valid and will invoke `Base.F`.

Namespace and type names

Several contexts in a C# program require a *namespace_name* or a *type_name* to be specified.

```
namespace_name
    : namespace_or_type_name
    ;

type_name
    : namespace_or_type_name
    ;

namespace_or_type_name
    : identifier type_argument_list?
    | namespace_or_type_name '.' identifier type_argument_list?
    | qualified_alias_member
    ;
```

A *namespace_name* is a *namespace_or_type_name* that refers to a namespace. Following resolution as described below, the *namespace_or_type_name* of a *namespace_name* must refer to a namespace, or otherwise a compile-time error occurs. No type arguments (Type arguments) can be present in a *namespace_name* (only types can have type arguments).

A *type_name* is a *namespace_or_type_name* that refers to a type. Following resolution as described below, the *namespace_or_type_name* of a *type_name* must refer to a type, or otherwise a compile-time error occurs.

If the *namespace_or_type_name* is a qualified-alias-member its meaning is as described in Namespace alias qualifiers. Otherwise, a *namespace_or_type_name* has one of four forms:

- `I`
- `I<A1, ..., Ak>`
- `N.I`
- `N.I<A1, ..., Ak>`

where `I` is a single identifier, `N` is a *namespace_or_type_name* and `<A1, ..., Ak>` is an optional *type_argument_list*. When no *type_argument_list* is specified, consider `k` to be zero.

The meaning of a *namespace_or_type_name* is determined as follows:

- If the *namespace_or_type_name* is of the form `I` or of the form `I<A1, ..., Ak>`:
  - If `K` is zero and the *namespace_or_type_name* appears within a generic method declaration (Methods) and if that declaration includes a type parameter (Type parameters) with name `I`, then the *namespace_or_type_name* refers to that type parameter.
  - Otherwise, if the *namespace_or_type_name* appears within a type declaration, then for each instance type `T` (The instance type), starting with the instance type of that type declaration and continuing with the instance type of each enclosing class or struct declaration (if any):
    - If `K` is zero and the declaration of `T` includes a type parameter with name `I`, then the *namespace_or_type_name* refers to that type parameter.
    - Otherwise, if the *namespace_or_type_name* appears within the body of the type declaration, and `T` or any of its base types contain a nested accessible type having name `I` and `K` type parameters, then the *namespace_or_type_name* refers to that type constructed with the given type arguments. If there is more than one such type, the type declared within the more derived type is selected. Note that non-type members (constants, fields, methods, properties, indexers, operators, instance constructors, destructors, and static constructors) and type members with a different number of type parameters are ignored when determining the meaning of the *namespace_or_type_name*.
  - If the previous steps were unsuccessful then, for each namespace `N`, starting with the namespace in which the *namespace_or_type_name* occurs, continuing with each enclosing namespace (if any), and ending with the global namespace, the following steps are evaluated until an entity is located:
    - If `K` is zero and `I` is the name of a namespace in `N`, then:
      - If the location where the *namespace_or_type_name* occurs is enclosed by a namespace declaration for `N` and the namespace declaration contains an *extern_alias_directive* or *using_alias_directive* that associates the name `I` with a namespace or type, then the *namespace_or_type_name* is ambiguous and a compile-time error occurs.
      - Otherwise, the *namespace_or_type_name* refers to the namespace named `I` in `N`.
    - Otherwise, if `N` contains an accessible type having name `I` and `K` type parameters, then:
      - If `K` is zero and the location where the *namespace_or_type_name* occurs is enclosed by a namespace declaration for `N` and the namespace declaration contains an *extern_alias_directive* or *using_alias_directive* that associates the name `I` with a namespace or type, then the *namespace_or_type_name* is ambiguous and a compile-time error occurs.
      - Otherwise, the *namespace_or_type_name* refers to the type constructed with the given type arguments.
    - Otherwise, if the location where the *namespace_or_type_name* occurs is enclosed by a namespace declaration for `N`:
      - If `K` is zero and the namespace declaration contains an *extern_alias_directive* or *using_alias_directive* that associates the name `I` with an imported namespace or type,

then the *namespace_or_type_name* refers to that namespace or type.
    - Otherwise, if the namespaces and type declarations imported by the *using_namespace_directive*s and *using_alias_directive*s of the namespace declaration contain exactly one accessible type having name `I` and `K` type parameters, then the *namespace_or_type_name* refers to that type constructed with the given type arguments.
    - Otherwise, if the namespaces and type declarations imported by the *using_namespace_directive*s and *using_alias_directive*s of the namespace declaration contain more than one accessible type having name `I` and `K` type parameters, then the *namespace_or_type_name* is ambiguous and an error occurs.
  - Otherwise, the *namespace_or_type_name* is undefined and a compile-time error occurs.
- Otherwise, the *namespace_or_type_name* is of the form `N.I` or of the form `N.I<A1, ..., Ak>`. `N` is first resolved as a *namespace_or_type_name*. If the resolution of `N` is not successful, a compile-time error occurs. Otherwise, `N.I` or `N.I<A1, ..., Ak>` is resolved as follows:
  - If `K` is zero and `N` refers to a namespace and `N` contains a nested namespace with name `I`, then the *namespace_or_type_name* refers to that nested namespace.
  - Otherwise, if `N` refers to a namespace and `N` contains an accessible type having name `I` and `K` type parameters, then the *namespace_or_type_name* refers to that type constructed with the given type arguments.
  - Otherwise, if `N` refers to a (possibly constructed) class or struct type and `N` or any of its base classes contain a nested accessible type having name `I` and `K` type parameters, then the *namespace_or_type_name* refers to that type constructed with the given type arguments. If there is more than one such type, the type declared within the more derived type is selected. Note that if the meaning of `N.I` is being determined as part of resolving the base class specification of `N` then the direct base class of `N` is considered to be object (Base classes).
  - Otherwise, `N.I` is an invalid *namespace_or_type_name*, and a compile-time error occurs.

A *namespace_or_type_name* is permitted to reference a static class (Static classes) only if

- The *namespace_or_type_name* is the `T` in a *namespace_or_type_name* of the form `T.I`, or
- The *namespace_or_type_name* is the `T` in a *typeof_expression* (Argument lists1) of the form `typeof(T)`.

Fully qualified names

Every namespace and type has a ***fully qualified name***, which uniquely identifies the namespace or type amongst all others. The fully qualified name of a namespace or type `N` is determined as follows:

- If `N` is a member of the global namespace, its fully qualified name is `N`.
- Otherwise, its fully qualified name is `S.N`, where `S` is the fully qualified name of the namespace or type in which `N` is declared.

In other words, the fully qualified name of `N` is the complete hierarchical path of identifiers that lead to `N`, starting from the global namespace. Because every member of a namespace or type must have a unique name, it follows that the fully qualified name of a namespace or type is always unique.

The example below shows several namespace and type declarations along with their associated fully qualified names.

```
class A {}                // A

namespace X               // X
{
    class B               // X.B
    {
        class C {}        // X.B.C
    }

    namespace Y           // X.Y
    {
        class D {}        // X.Y.D
    }
}

namespace X.Y             // X.Y
{
    class E {}            // X.Y.E
}
```

Automatic memory management

C# employs automatic memory management, which frees developers from manually allocating and freeing the memory occupied by objects. Automatic memory management policies are implemented by a ***garbage collector***. The memory management life cycle of an object is as follows:

1. When the object is created, memory is allocated for it, the constructor is run, and the object is considered live.
2. If the object, or any part of it, cannot be accessed by any possible continuation of execution, other than the running of destructors, the object is considered no longer in use, and it becomes eligible for destruction. The C# compiler and the garbage collector may choose to analyze code to determine which references to an object may be used in the future. For instance, if a local variable that is in scope is the only existing reference to an object, but that local variable is never referred to in any possible continuation of execution from the current execution point in the procedure, the garbage collector may (but is not required to) treat the object as no longer in use.
3. Once the object is eligible for destruction, at some unspecified later time the destructor (Destructors) (if any) for the object is run. Under normal circumstances the destructor for the object is run once only, though implementation-specific APIs may allow this behavior to be overridden.
4. Once the destructor for an object is run, if that object, or any part of it, cannot be accessed by any possible continuation of execution, including the running of destructors, the object is considered inaccessible and the object becomes eligible for collection.
5. Finally, at some time after the object becomes eligible for collection, the garbage collector frees the memory associated with that object.

The garbage collector maintains information about object usage, and uses this information to make memory management decisions, such as where in memory to locate a newly created object, when to relocate an object, and when an object is no longer in use or inaccessible.

Like other languages that assume the existence of a garbage collector, C# is designed so that the garbage collector may implement a wide range of memory management policies. For instance, C# does not require that destructors be run or that objects be collected as soon as they are eligible, or that destructors be run in any particular order, or on any particular thread.

The behavior of the garbage collector can be controlled, to some degree, via static methods on the class `System.GC`. This class can be used to request a collection to occur, destructors to be run (or not run), and so forth.

Since the garbage collector is allowed wide latitude in deciding when to collect objects and run destructors, a conforming implementation may produce output that differs from that shown by the following code. The program

```
using System;

class A
{
    ~A() {
        Console.WriteLine("Destruct instance of A");
    }
}

class B
{
    object Ref;

    public B(object o) {
        Ref = o;
    }

    ~B() {
        Console.WriteLine("Destruct instance of B");
    }
}

class Test
{
    static void Main() {
        B b = new B(new A());
        b = null;
        GC.Collect();
        GC.WaitForPendingFinalizers();
    }
}
```

creates an instance of class `A` and an instance of class `B`. These objects become eligible for garbage collection when the variable `b` is assigned the value `null`, since after this time it is impossible for any user-written code to access them. The output could be either

```
Destruct instance of A
Destruct instance of B
```

or

```
Destruct instance of B
Destruct instance of A
```

because the language imposes no constraints on the order in which objects are garbage collected.

In subtle cases, the distinction between "eligible for destruction" and "eligible for collection" can be important. For example,

```
using System;

class A
{
    ~A() {
        Console.WriteLine("Destruct instance of A");
    }

    public void F() {
        Console.WriteLine("A.F");
        Test.RefA = this;
    }
}

class B
{
    public A Ref;

    ~B() {
        Console.WriteLine("Destruct instance of B");
        Ref.F();
    }
}

class Test
{
    public static A RefA;
    public static B RefB;

    static void Main() {
        RefB = new B();
        RefA = new A();
        RefB.Ref = RefA;
        RefB = null;
        RefA = null;

        // A and B now eligible for destruction
        GC.Collect();
        GC.WaitForPendingFinalizers();

        // B now eligible for collection, but A is not
        if (RefA != null)
            Console.WriteLine("RefA is not null");
    }
}
```

In the above program, if the garbage collector chooses to run the destructor of `A` before the destructor of `B`, then the output of this program might be:

```
Destruct instance of A
Destruct instance of B
A.F
RefA is not null
```

Note that although the instance of `A` was not in use and `A`'s destructor was run, it is still possible for methods of `A` (in this case, `F`) to be called from another destructor. Also, note that running of a destructor may cause an object to become usable from the mainline program again. In this case, the running of `B`'s destructor caused an instance of `A` that was previously not in use to become accessible from the live reference `Test.RefA`. After the call to `WaitForPendingFinalizers`, the instance of `B` is eligible for collection, but the instance of `A` is not, because of the reference `Test.RefA`.

To avoid confusion and unexpected behavior, it is generally a good idea for destructors to only perform cleanup

on data stored in their object's own fields, and not to perform any actions on referenced objects or static fields.

An alternative to using destructors is to let a class implement the `System.IDisposable` interface. This allows the client of the object to determine when to release the resources of the object, typically by accessing the object as a resource in a `using` statement (The using statement).

Execution order

Execution of a C# program proceeds such that the side effects of each executing thread are preserved at critical execution points. A ***side effect*** is defined as a read or write of a volatile field, a write to a non-volatile variable, a write to an external resource, and the throwing of an exception. The critical execution points at which the order of these side effects must be preserved are references to volatile fields (Volatile fields), `lock` statements (The lock statement), and thread creation and termination. The execution environment is free to change the order of execution of a C# program, subject to the following constraints:

- Data dependence is preserved within a thread of execution. That is, the value of each variable is computed as if all statements in the thread were executed in original program order.
- Initialization ordering rules are preserved (Field initialization and Variable initializers).
- The ordering of side effects is preserved with respect to volatile reads and writes (Volatile fields). Additionally, the execution environment need not evaluate part of an expression if it can deduce that that expression's value is not used and that no needed side effects are produced (including any caused by calling a method or accessing a volatile field). When program execution is interrupted by an asynchronous event (such as an exception thrown by another thread), it is not guaranteed that the observable side effects are visible in the original program order.

Types

18.12.2021 • 33 minutes to read

The types of the C# language are divided into two main categories: ***value types*** and ***reference types***. Both value types and reference types may be ***generic types***, which take one or more ***type parameters***. Type parameters can designate both value types and reference types.

```
type
    : value_type
    | reference_type
    | type_parameter
    | type_unsafe
    ;
```

The final category of types, pointers, is available only in unsafe code. This is discussed further in Pointer types.

Value types differ from reference types in that variables of the value types directly contain their data, whereas variables of the reference types store ***references*** to their data, the latter being known as ***objects***. With reference types, it is possible for two variables to reference the same object, and thus possible for operations on one variable to affect the object referenced by the other variable. With value types, the variables each have their own copy of the data, and it is not possible for operations on one to affect the other.

C#'s type system is unified such that a value of any type can be treated as an object. Every type in C# directly or indirectly derives from the `object` class type, and `object` is the ultimate base class of all types. Values of reference types are treated as objects simply by viewing the values as type `object`. Values of value types are treated as objects by performing boxing and unboxing operations (Boxing and unboxing).

Value types

A value type is either a struct type or an enumeration type. C# provides a set of predefined struct types called the ***simple types***. The simple types are identified through reserved words.

```
value_type
    : struct_type
    | enum_type
    ;

struct_type
    : type_name
    | simple_type
    | nullable_type
    ;

simple_type
    : numeric_type
    | 'bool'
    ;

numeric_type
    : integral_type
    | floating_point_type
    | 'decimal'
    ;

integral_type
    : 'sbyte'
    | 'byte'
    | 'short'
    | 'ushort'
    | 'int'
    | 'uint'
    | 'long'
    | 'ulong'
    | 'char'
    ;

floating_point_type
    : 'float'
    | 'double'
    ;

nullable_type
    : non_nullable_value_type '?'
    ;

non_nullable_value_type
    : type
    ;

enum_type
    : type_name
    ;
```

Unlike a variable of a reference type, a variable of a value type can contain the value `null` only if the value type is a nullable type. For every non-nullable value type there is a corresponding nullable value type denoting the same set of values plus the value `null`.

Assignment to a variable of a value type creates a copy of the value being assigned. This differs from assignment to a variable of a reference type, which copies the reference but not the object identified by the reference.

The System.ValueType type

All value types implicitly inherit from the class `System.ValueType`, which, in turn, inherits from class `object`. It is not possible for any type to derive from a value type, and value types are thus implicitly sealed (Sealed classes).

Note that `System.ValueType` is not itself a *value_type*. Rather, it is a *class_type* from which all *value_type*s are

automatically derived.

Default constructors

All value types implicitly declare a public parameterless instance constructor called the ***default constructor***. The default constructor returns a zero-initialized instance known as the ***default value*** for the value type:

- For all *simple_type*s, the default value is the value produced by a bit pattern of all zeros:
  - For `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, and `ulong`, the default value is `0`.
  - For `char`, the default value is `'\x0000'`.
  - For `float`, the default value is `0.0f`.
  - For `double`, the default value is `0.0d`.
  - For `decimal`, the default value is `0.0m`.
  - For `bool`, the default value is `false`.
- For an *enum_type* `E`, the default value is `0`, converted to the type `E`.
- For a *struct_type*, the default value is the value produced by setting all value type fields to their default value and all reference type fields to `null`.
- For a *nullable_type* the default value is an instance for which the `HasValue` property is false and the `Value` property is undefined. The default value is also known as the ***null value*** of the nullable type.

Like any other instance constructor, the default constructor of a value type is invoked using the `new` operator. For efficiency reasons, this requirement is not intended to actually have the implementation generate a constructor call. In the example below, variables `i` and `j` are both initialized to zero.

```
class A
{
    void F() {
        int i = 0;
        int j = new int();
    }
}
```

Because every value type implicitly has a public parameterless instance constructor, it is not possible for a struct type to contain an explicit declaration of a parameterless constructor. A struct type is however permitted to declare parameterized instance constructors (Constructors).

Struct types

A struct type is a value type that can declare constants, fields, methods, properties, indexers, operators, instance constructors, static constructors, and nested types. The declaration of struct types is described in Struct declarations.

Simple types

C# provides a set of predefined struct types called the ***simple types***. The simple types are identified through reserved words, but these reserved words are simply aliases for predefined struct types in the `System` namespace, as described in the table below.

RESERVED WORD	ALIASED TYPE
`sbyte`	`System.SByte`
`byte`	`System.Byte`
`short`	`System.Int16`

RESERVED WORD	ALIASED TYPE
`ushort`	`System.UInt16`
`int`	`System.Int32`
`uint`	`System.UInt32`
`long`	`System.Int64`
`ulong`	`System.UInt64`
`char`	`System.Char`
`float`	`System.Single`
`double`	`System.Double`
`bool`	`System.Boolean`
`decimal`	`System.Decimal`

Because a simple type aliases a struct type, every simple type has members. For example, `int` has the members declared in `System.Int32` and the members inherited from `System.Object`, and the following statements are permitted:

```
int i = int.MaxValue;           // System.Int32.MaxValue constant
string s = i.ToString();        // System.Int32.ToString() instance method
string t = 123.ToString();      // System.Int32.ToString() instance method
```

The simple types differ from other struct types in that they permit certain additional operations:

- Most simple types permit values to be created by writing *literals* (Literals). For example, `123` is a literal of type `int` and `'a'` is a literal of type `char`. C# makes no provision for literals of struct types in general, and non-default values of other struct types are ultimately always created through instance constructors of those struct types.
- When the operands of an expression are all simple type constants, it is possible for the compiler to evaluate the expression at compile-time. Such an expression is known as a *constant_expression* (Constant expressions). Expressions involving operators defined by other struct types are not considered to be constant expressions.
- Through `const` declarations it is possible to declare constants of the simple types (Constants). It is not possible to have constants of other struct types, but a similar effect is provided by `static readonly` fields.
- Conversions involving simple types can participate in evaluation of conversion operators defined by other struct types, but a user-defined conversion operator can never participate in evaluation of another user-defined operator (Evaluation of user-defined conversions).

Integral types

C# supports nine integral types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, and `char`. The integral types have the following sizes and ranges of values:

- The `sbyte` type represents signed 8-bit integers with values between -128 and 127.
- The `byte` type represents unsigned 8-bit integers with values between 0 and 255.

- The `short` type represents signed 16-bit integers with values between -32768 and 32767.
- The `ushort` type represents unsigned 16-bit integers with values between 0 and 65535.
- The `int` type represents signed 32-bit integers with values between -2147483648 and 2147483647.
- The `uint` type represents unsigned 32-bit integers with values between 0 and 4294967295.
- The `long` type represents signed 64-bit integers with values between -9223372036854775808 and 9223372036854775807.
- The `ulong` type represents unsigned 64-bit integers with values between 0 and 18446744073709551615.
- The `char` type represents unsigned 16-bit integers with values between 0 and 65535. The set of possible values for the `char` type corresponds to the Unicode character set. Although `char` has the same representation as `ushort`, not all operations permitted on one type are permitted on the other.

The integral-type unary and binary operators always operate with signed 32-bit precision, unsigned 32-bit precision, signed 64-bit precision, or unsigned 64-bit precision:

- For the unary `+` and `~` operators, the operand is converted to type `T`, where `T` is the first of `int`, `uint`, `long`, and `ulong` that can fully represent all possible values of the operand. The operation is then performed using the precision of type `T`, and the type of the result is `T`.
- For the unary `-` operator, the operand is converted to type `T`, where `T` is the first of `int` and `long` that can fully represent all possible values of the operand. The operation is then performed using the precision of type `T`, and the type of the result is `T`. The unary `-` operator cannot be applied to operands of type `ulong`.
- For the binary `+`, `-`, `*`, `/`, `%`, `&`, `^`, `|`, `==`, `!=`, `>`, `<`, `>=`, and `<=` operators, the operands are converted to type `T`, where `T` is the first of `int`, `uint`, `long`, and `ulong` that can fully represent all possible values of both operands. The operation is then performed using the precision of type `T`, and the type of the result is `T` (or `bool` for the relational operators). It is not permitted for one operand to be of type `long` and the other to be of type `ulong` with the binary operators.
- For the binary `<<` and `>>` operators, the left operand is converted to type `T`, where `T` is the first of `int`, `uint`, `long`, and `ulong` that can fully represent all possible values of the operand. The operation is then performed using the precision of type `T`, and the type of the result is `T`.

The `char` type is classified as an integral type, but it differs from the other integral types in two ways:

- There are no implicit conversions from other types to the `char` type. In particular, even though the `sbyte`, `byte`, and `ushort` types have ranges of values that are fully representable using the `char` type, implicit conversions from `sbyte`, `byte`, or `ushort` to `char` do not exist.
- Constants of the `char` type must be written as *character_literal*s or as *integer_literal*s in combination with a cast to type `char`. For example, `(char)10` is the same as `'\x000A'`.

The `checked` and `unchecked` operators and statements are used to control overflow checking for integral-type arithmetic operations and conversions (The checked and unchecked operators). In a `checked` context, an overflow produces a compile-time error or causes a `System.OverflowException` to be thrown. In an `unchecked` context, overflows are ignored and any high-order bits that do not fit in the destination type are discarded.

Floating point types

C# supports two floating point types: `float` and `double`. The `float` and `double` types are represented using the 32-bit single-precision and 64-bit double-precision IEEE 754 formats, which provide the following sets of values:

- Positive zero and negative zero. In most situations, positive zero and negative zero behave identically as the simple value zero, but certain operations distinguish between the two (Division operator).
- Positive infinity and negative infinity. Infinities are produced by such operations as dividing a non-zero number by zero. For example, `1.0 / 0.0` yields positive infinity, and `-1.0 / 0.0` yields negative infinity.

- The ***Not-a-Number*** value, often abbreviated NaN. NaNs are produced by invalid floating-point operations, such as dividing zero by zero.
- The finite set of non-zero values of the form `s * m * 2^e`, where `s` is 1 or -1, and `m` and `e` are determined by the particular floating-point type: For `float`, `0 < m < 2^24` and `-149 <= e <= 104`, and for `double`, `0 < m < 2^53` and `-1075 <= e <= 970`. Denormalized floating-point numbers are considered valid non-zero values.

The `float` type can represent values ranging from approximately `1.5 * 10^-45` to `3.4 * 10^38` with a precision of 7 digits.

The `double` type can represent values ranging from approximately `5.0 * 10^-324` to `1.7 × 10^308` with a precision of 15-16 digits.

If one of the operands of a binary operator is of a floating-point type, then the other operand must be of an integral type or a floating-point type, and the operation is evaluated as follows:

- If one of the operands is of an integral type, then that operand is converted to the floating-point type of the other operand.
- Then, if either of the operands is of type `double`, the other operand is converted to `double`, the operation is performed using at least `double` range and precision, and the type of the result is `double` (or `bool` for the relational operators).
- Otherwise, the operation is performed using at least `float` range and precision, and the type of the result is `float` (or `bool` for the relational operators).

The floating-point operators, including the assignment operators, never produce exceptions. Instead, in exceptional situations, floating-point operations produce zero, infinity, or NaN, as described below:

- If the result of a floating-point operation is too small for the destination format, the result of the operation becomes positive zero or negative zero.
- If the result of a floating-point operation is too large for the destination format, the result of the operation becomes positive infinity or negative infinity.
- If a floating-point operation is invalid, the result of the operation becomes NaN.
- If one or both operands of a floating-point operation is NaN, the result of the operation becomes NaN.

Floating-point operations may be performed with higher precision than the result type of the operation. For example, some hardware architectures support an "extended" or "long double" floating-point type with greater range and precision than the `double` type, and implicitly perform all floating-point operations using this higher precision type. Only at excessive cost in performance can such hardware architectures be made to perform floating-point operations with less precision, and rather than require an implementation to forfeit both performance and precision, C# allows a higher precision type to be used for all floating-point operations. Other than delivering more precise results, this rarely has any measurable effects. However, in expressions of the form `x * y / z`, where the multiplication produces a result that is outside the `double` range, but the subsequent division brings the temporary result back into the `double` range, the fact that the expression is evaluated in a higher range format may cause a finite result to be produced instead of an infinity.

The decimal type

The `decimal` type is a 128-bit data type suitable for financial and monetary calculations. The `decimal` type can represent values ranging from `1.0 * 10^-28` to approximately `7.9 * 10^28` with 28-29 significant digits.

The finite set of values of type `decimal` are of the form `(-1)^s * c * 10^-e`, where the sign `s` is 0 or 1, the coefficient `c` is given by `0 <= *c* < 2^96`, and the scale `e` is such that `0 <= e <= 28`.The `decimal` type does not support signed zeros, infinities, or NaN's. A `decimal` is represented as a 96-bit integer scaled by a power of ten. For `decimal`s with an absolute value less than `1.0m`, the value is exact to the 28th decimal place, but no further. For `decimal`s with an absolute value greater than or equal to `1.0m`, the value is exact to 28 or 29 digits.

Contrary to the `float` and `double` data types, decimal fractional numbers such as 0.1 can be represented exactly in the `decimal` representation. In the `float` and `double` representations, such numbers are often infinite fractions, making those representations more prone to round-off errors.

If one of the operands of a binary operator is of type `decimal`, then the other operand must be of an integral type or of type `decimal`. If an integral type operand is present, it is converted to `decimal` before the operation is performed.

The result of an operation on values of type `decimal` is that which would result from calculating an exact result (preserving scale, as defined for each operator) and then rounding to fit the representation. Results are rounded to the nearest representable value, and, when a result is equally close to two representable values, to the value that has an even number in the least significant digit position (this is known as "banker's rounding"). A zero result always has a sign of 0 and a scale of 0.

If a decimal arithmetic operation produces a value less than or equal to `5 * 10^-29` in absolute value, the result of the operation becomes zero. If a `decimal` arithmetic operation produces a result that is too large for the `decimal` format, a `System.OverflowException` is thrown.

The `decimal` type has greater precision but smaller range than the floating-point types. Thus, conversions from the floating-point types to `decimal` might produce overflow exceptions, and conversions from `decimal` to the floating-point types might cause loss of precision. For these reasons, no implicit conversions exist between the floating-point types and `decimal`, and without explicit casts, it is not possible to mix floating-point and `decimal` operands in the same expression.

The bool type

The `bool` type represents boolean logical quantities. The possible values of type `bool` are `true` and `false`.

No standard conversions exist between `bool` and other types. In particular, the `bool` type is distinct and separate from the integral types, and a `bool` value cannot be used in place of an integral value, and vice versa.

In the C and C++ languages, a zero integral or floating-point value, or a null pointer can be converted to the boolean value `false`, and a non-zero integral or floating-point value, or a non-null pointer can be converted to the boolean value `true`. In C#, such conversions are accomplished by explicitly comparing an integral or floating-point value to zero, or by explicitly comparing an object reference to `null`.

Enumeration types

An enumeration type is a distinct type with named constants. Every enumeration type has an underlying type, which must be `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` or `ulong`. The set of values of the enumeration type is the same as the set of values of the underlying type. Values of the enumeration type are not restricted to the values of the named constants. Enumeration types are defined through enumeration declarations (Enum declarations).

Nullable types

A nullable type can represent all values of its ***underlying type*** plus an additional null value. A nullable type is written `T?`, where `T` is the underlying type. This syntax is shorthand for `System.Nullable<T>`, and the two forms can be used interchangeably.

A ***non-nullable value type*** conversely is any value type other than `System.Nullable<T>` and its shorthand `T?` (for any `T`), plus any type parameter that is constrained to be a non-nullable value type (that is, any type parameter with a `struct` constraint). The `System.Nullable<T>` type specifies the value type constraint for `T` (Type parameter constraints), which means that the underlying type of a nullable type can be any non-nullable value type. The underlying type of a nullable type cannot be a nullable type or a reference type. For example, `int??` and `string?` are invalid types.

An instance of a nullable type `T?` has two public read-only properties:

- A `HasValue` property of type `bool`
- A `Value` property of type `T`

An instance for which `HasValue` is true is said to be non-null. A non-null instance contains a known value and `Value` returns that value.

An instance for which `HasValue` is false is said to be null. A null instance has an undefined value. Attempting to read the `Value` of a null instance causes a `System.InvalidOperationException` to be thrown. The process of accessing the `Value` property of a nullable instance is referred to as ***unwrapping***.

In addition to the default constructor, every nullable type `T?` has a public constructor that takes a single argument of type `T`. Given a value `x` of type `T`, a constructor invocation of the form

```
new T?(x)
```

creates a non-null instance of `T?` for which the `Value` property is `x`. The process of creating a non-null instance of a nullable type for a given value is referred to as ***wrapping***.

Implicit conversions are available from the `null` literal to `T?` (Null literal conversions) and from `T` to `T?` (Implicit nullable conversions).

Reference types

A reference type is a class type, an interface type, an array type, or a delegate type.

```
reference_type
    : class_type
    | interface_type
    | array_type
    | delegate_type
    ;

class_type
    : type_name
    | 'object'
    | 'dynamic'
    | 'string'
    ;

interface_type
    : type_name
    ;

array_type
    : non_array_type rank_specifier+
    ;

non_array_type
    : type
    ;

rank_specifier
    : '[' dim_separator* ']'
    ;

dim_separator
    : ','
    ;

delegate_type
    : type_name
    ;
```

A reference type value is a reference to an ***instance*** of the type, the latter known as an ***object***. The special value `null` is compatible with all reference types and indicates the absence of an instance.

Class types

A class type defines a data structure that contains data members (constants and fields), function members (methods, properties, events, indexers, operators, instance constructors, destructors and static constructors), and nested types. Class types support inheritance, a mechanism whereby derived classes can extend and specialize base classes. Instances of class types are created using *object_creation_expression*s (Object creation expressions).

Class types are described in Classes.

Certain predefined class types have special meaning in the C# language, as described in the table below.

CLASS TYPE	DESCRIPTION
`System.Object`	The ultimate base class of all other types. See The object type.
`System.String`	The string type of the C# language. See The string type.
`System.ValueType`	The base class of all value types. See The System.ValueType type.

CLASS TYPE	DESCRIPTION
`System.Enum`	The base class of all enum types. See Enums.
`System.Array`	The base class of all array types. See Arrays.
`System.Delegate`	The base class of all delegate types. See Delegates.
`System.Exception`	The base class of all exception types. See Exceptions.

The object type

The `object` class type is the ultimate base class of all other types. Every type in C# directly or indirectly derives from the `object` class type.

The keyword `object` is simply an alias for the predefined class `System.Object`.

The dynamic type

The `dynamic` type, like `object`, can reference any object. When operators are applied to expressions of type `dynamic`, their resolution is deferred until the program is run. Thus, if the operator cannot legally be applied to the referenced object, no error is given during compilation. Instead an exception will be thrown when resolution of the operator fails at run-time.

Its purpose is to allow dynamic binding, which is described in detail in Dynamic binding.

`dynamic` is considered identical to `object` except in the following respects:

- Operations on expressions of type `dynamic` can be dynamically bound (Dynamic binding).
- Type inference (Type inference) will prefer `dynamic` over `object` if both are candidates.

Because of this equivalence, the following holds:

- There is an implicit identity conversion between `object` and `dynamic`, and between constructed types that are the same when replacing `dynamic` with `object`
- Implicit and explicit conversions to and from `object` also apply to and from `dynamic`.
- Method signatures that are the same when replacing `dynamic` with `object` are considered the same signature
- The type `dynamic` is indistinguishable from `object` at run-time.
- An expression of the type `dynamic` is referred to as a ***dynamic expression***.

The string type

The `string` type is a sealed class type that inherits directly from `object`. Instances of the `string` class represent Unicode character strings.

Values of the `string` type can be written as string literals (String literals).

The keyword `string` is simply an alias for the predefined class `System.String`.

Interface types

An interface defines a contract. A class or struct that implements an interface must adhere to its contract. An interface may inherit from multiple base interfaces, and a class or struct may implement multiple interfaces.

Interface types are described in Interfaces.

Array types

An array is a data structure that contains zero or more variables which are accessed through computed indices.

The variables contained in an array, also called the elements of the array, are all of the same type, and this type is called the element type of the array.

Array types are described in Arrays.

Delegate types

A delegate is a data structure that refers to one or more methods. For instance methods, it also refers to their corresponding object instances.

The closest equivalent of a delegate in C or C++ is a function pointer, but whereas a function pointer can only reference static functions, a delegate can reference both static and instance methods. In the latter case, the delegate stores not only a reference to the method's entry point, but also a reference to the object instance on which to invoke the method.

Delegate types are described in Delegates.

Boxing and unboxing

The concept of boxing and unboxing is central to C#'s type system. It provides a bridge between *value_type*s and *reference_type*s by permitting any value of a *value_type* to be converted to and from type `object`. Boxing and unboxing enables a unified view of the type system wherein a value of any type can ultimately be treated as an object.

Boxing conversions

A boxing conversion permits a *value_type* to be implicitly converted to a *reference_type*. The following boxing conversions exist:

- From any *value_type* to the type `object`.
- From any *value_type* to the type `System.ValueType`.
- From any *non_nullable_value_type* to any *interface_type* implemented by the *value_type*.
- From any *nullable_type* to any *interface_type* implemented by the underlying type of the *nullable_type*.
- From any *enum_type* to the type `System.Enum`.
- From any *nullable_type* with an underlying *enum_type* to the type `System.Enum`.
- Note that an implicit conversion from a type parameter will be executed as a boxing conversion if at run-time it ends up converting from a value type to a reference type (Implicit conversions involving type parameters).

Boxing a value of a *non_nullable_value_type* consists of allocating an object instance and copying the *non_nullable_value_type* value into that instance.

Boxing a value of a *nullable_type* produces a null reference if it is the `null` value (`HasValue` is `false`), or the result of unwrapping and boxing the underlying value otherwise.

The actual process of boxing a value of a *non_nullable_value_type* is best explained by imagining the existence of a generic ***boxing class***, which behaves as if it were declared as follows:

```
sealed class Box<T>: System.ValueType
{
    T value;

    public Box(T t) {
        value = t;
    }
}
```

Boxing of a value `v` of type `T` now consists of executing the expression `new Box<T>(v)`, and returning the resulting instance as a value of type `object`. Thus, the statements

```
int i = 123;
object box = i;
```

conceptually correspond to

```
int i = 123;
object box = new Box<int>(i);
```

A boxing class like `Box<T>` above doesn't actually exist and the dynamic type of a boxed value isn't actually a class type. Instead, a boxed value of type `T` has the dynamic type `T`, and a dynamic type check using the `is` operator can simply reference type `T`. For example,

```
int i = 123;
object box = i;
if (box is int) {
    Console.Write("Box contains an int");
}
```

will output the string "`Box contains an int`" on the console.

A boxing conversion implies making a copy of the value being boxed. This is different from a conversion of a *reference_type* to type `object`, in which the value continues to reference the same instance and simply is regarded as the less derived type `object`. For example, given the declaration

```
struct Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

the following statements

```
Point p = new Point(10, 10);
object box = p;
p.x = 20;
Console.Write(((Point)box).x);
```

will output the value 10 on the console because the implicit boxing operation that occurs in the assignment of `p` to `box` causes the value of `p` to be copied. Had `Point` been declared a `class` instead, the value 20 would be output because `p` and `box` would reference the same instance.

Unboxing conversions

An unboxing conversion permits a *reference_type* to be explicitly converted to a *value_type*. The following unboxing conversions exist:

- From the type `object` to any *value_type*.
- From the type `System.ValueType` to any *value_type*.
- From any *interface_type* to any *non_nullable_value_type* that implements the *interface_type*.
- From any *interface_type* to any *nullable_type* whose underlying type implements the *interface_type*.
- From the type `System.Enum` to any *enum_type*.

- From the type `System.Enum` to any *nullable_type* with an underlying *enum_type*.
- Note that an explicit conversion to a type parameter will be executed as an unboxing conversion if at run-time it ends up converting from a reference type to a value type (Explicit dynamic conversions).

An unboxing operation to a *non_nullable_value_type* consists of first checking that the object instance is a boxed value of the given *non_nullable_value_type*, and then copying the value out of the instance.

Unboxing to a *nullable_type* produces the null value of the *nullable_type* if the source operand is `null`, or the wrapped result of unboxing the object instance to the underlying type of the *nullable_type* otherwise.

Referring to the imaginary boxing class described in the previous section, an unboxing conversion of an object `box` to a *value_type* `T` consists of executing the expression `((Box<T>)box).value`. Thus, the statements

```
object box = 123;
int i = (int)box;
```

conceptually correspond to

```
object box = new Box<int>(123);
int i = ((Box<int>)box).value;
```

For an unboxing conversion to a given *non_nullable_value_type* to succeed at run-time, the value of the source operand must be a reference to a boxed value of that *non_nullable_value_type*. If the source operand is `null`, a `System.NullReferenceException` is thrown. If the source operand is a reference to an incompatible object, a `System.InvalidCastException` is thrown.

For an unboxing conversion to a given *nullable_type* to succeed at run-time, the value of the source operand must be either `null` or a reference to a boxed value of the underlying *non_nullable_value_type* of the *nullable_type*. If the source operand is a reference to an incompatible object, a `System.InvalidCastException` is thrown.

Constructed types

A generic type declaration, by itself, denotes an ***unbound generic type*** that is used as a "blueprint" to form many different types, by way of applying ***type arguments***. The type arguments are written within angle brackets (`<` and `>`) immediately following the name of the generic type. A type that includes at least one type argument is called a ***constructed type***. A constructed type can be used in most places in the language in which a type name can appear. An unbound generic type can only be used within a *typeof_expression* (The typeof operator).

Constructed types can also be used in expressions as simple names (Simple names) or when accessing a member (Member access).

When a *namespace_or_type_name* is evaluated, only generic types with the correct number of type parameters are considered. Thus, it is possible to use the same identifier to identify different types, as long as the types have different numbers of type parameters. This is useful when mixing generic and non-generic classes in the same program:

```
namespace Widgets
{
    class Queue {...}
    class Queue<TElement> {...}
}

namespace MyApplication
{
    using Widgets;

    class X
    {
        Queue q1;            // Non-generic Widgets.Queue
        Queue<int> q2;       // Generic Widgets.Queue
    }
}
```

A *type_name* might identify a constructed type even though it doesn't specify type parameters directly. This can occur where a type is nested within a generic class declaration, and the instance type of the containing declaration is implicitly used for name lookup (Nested types in generic classes):

```
class Outer<T>
{
    public class Inner {...}

    public Inner i;               // Type of i is Outer<T>.Inner
}
```

In unsafe code, a constructed type cannot be used as an *unmanaged_type* (Pointer types).

Type arguments

Each argument in a type argument list is simply a *type*.

```
type_argument_list
    : '<' type_arguments '>'
    ;

type_arguments
    : type_argument (',' type_argument)*
    ;

type_argument
    : type
    ;
```

In unsafe code (Unsafe code), a *type_argument* may not be a pointer type. Each type argument must satisfy any constraints on the corresponding type parameter (Type parameter constraints).

Open and closed types

All types can be classified as either ***open types*** or ***closed types***. An open type is a type that involves type parameters. More specifically:

- A type parameter defines an open type.
- An array type is an open type if and only if its element type is an open type.
- A constructed type is an open type if and only if one or more of its type arguments is an open type. A constructed nested type is an open type if and only if one or more of its type arguments or the type arguments of its containing type(s) is an open type.

A closed type is a type that is not an open type.

At run-time, all of the code within a generic type declaration is executed in the context of a closed constructed type that was created by applying type arguments to the generic declaration. Each type parameter within the generic type is bound to a particular run-time type. The run-time processing of all statements and expressions always occurs with closed types, and open types occur only during compile-time processing.

Each closed constructed type has its own set of static variables, which are not shared with any other closed constructed types. Since an open type does not exist at run-time, there are no static variables associated with an open type. Two closed constructed types are the same type if they are constructed from the same unbound generic type, and their corresponding type arguments are the same type.

Bound and unbound types

The term ***unbound type*** refers to a non-generic type or an unbound generic type. The term ***bound type*** refers to a non-generic type or a constructed type.

An unbound type refers to the entity declared by a type declaration. An unbound generic type is not itself a type, and cannot be used as the type of a variable, argument or return value, or as a base type. The only construct in which an unbound generic type can be referenced is the `typeof` expression (The typeof operator).

Satisfying constraints

Whenever a constructed type or generic method is referenced, the supplied type arguments are checked against the type parameter constraints declared on the generic type or method (Type parameter constraints). For each `where` clause, the type argument `A` that corresponds to the named type parameter is checked against each constraint as follows:

- If the constraint is a class type, an interface type, or a type parameter, let `C` represent that constraint with the supplied type arguments substituted for any type parameters that appear in the constraint. To satisfy the constraint, it must be the case that type `A` is convertible to type `C` by one of the following:
  - An identity conversion (Identity conversion)
  - An implicit reference conversion (Implicit reference conversions)
  - A boxing conversion (Boxing conversions), provided that type A is a non-nullable value type.
  - An implicit reference, boxing or type parameter conversion from a type parameter `A` to `C`.
- If the constraint is the reference type constraint (`class`), the type `A` must satisfy one of the following:
  - `A` is an interface type, class type, delegate type or array type. Note that `System.ValueType` and `System.Enum` are reference types that satisfy this constraint.
  - `A` is a type parameter that is known to be a reference type (Type parameter constraints).
- If the constraint is the value type constraint (`struct`), the type `A` must satisfy one of the following:
  - `A` is a struct type or enum type, but not a nullable type. Note that `System.ValueType` and `System.Enum` are reference types that do not satisfy this constraint.
  - `A` is a type parameter having the value type constraint (Type parameter constraints).
- If the constraint is the constructor constraint `new()`, the type `A` must not be `abstract` and must have a public parameterless constructor. This is satisfied if one of the following is true:
  - `A` is a value type, since all value types have a public default constructor (Default constructors).
  - `A` is a type parameter having the constructor constraint (Type parameter constraints).
  - `A` is a type parameter having the value type constraint (Type parameter constraints).
  - `A` is a class that is not `abstract` and contains an explicitly declared `public` constructor with no parameters.
  - `A` is not `abstract` and has a default constructor (Default constructors).

A compile-time error occurs if one or more of a type parameter's constraints are not satisfied by the given type arguments.

Since type parameters are not inherited, constraints are never inherited either. In the example below, `D` needs to specify the constraint on its type parameter `T` so that `T` satisfies the constraint imposed by the base class `B<T>`. In contrast, class `E` need not specify a constraint, because `List<T>` implements `IEnumerable` for any `T`.

```
class B<T> where T: IEnumerable {...}

class D<T>: B<T> where T: IEnumerable {...}

class E<T>: B<List<T>> {...}
```

Type parameters

A type parameter is an identifier designating a value type or reference type that the parameter is bound to at run-time.

```
type_parameter
    : identifier
    ;
```

Since a type parameter can be instantiated with many different actual type arguments, type parameters have slightly different operations and restrictions than other types. These include:

- A type parameter cannot be used directly to declare a base class (Base class) or interface (Variant type parameter lists).
- The rules for member lookup on type parameters depend on the constraints, if any, applied to the type parameter. They are detailed in Member lookup.
- The available conversions for a type parameter depend on the constraints, if any, applied to the type parameter. They are detailed in Implicit conversions involving type parameters and Explicit dynamic conversions.
- The literal `null` cannot be converted to a type given by a type parameter, except if the type parameter is known to be a reference type (Implicit conversions involving type parameters). However, a `default` expression (Default value expressions) can be used instead. In addition, a value with a type given by a type parameter can be compared with `null` using `==` and `!=` (Reference type equality operators) unless the type parameter has the value type constraint.
- A `new` expression (Object creation expressions) can only be used with a type parameter if the type parameter is constrained by a *constructor_constraint* or the value type constraint (Type parameter constraints).
- A type parameter cannot be used anywhere within an attribute.
- A type parameter cannot be used in a member access (Member access) or type name (Namespace and type names) to identify a static member or a nested type.
- In unsafe code, a type parameter cannot be used as an *unmanaged_type* (Pointer types).

As a type, type parameters are purely a compile-time construct. At run-time, each type parameter is bound to a run-time type that was specified by supplying a type argument to the generic type declaration. Thus, the type of a variable declared with a type parameter will, at run-time, be a closed constructed type (Open and closed types). The run-time execution of all statements and expressions involving type parameters uses the actual type that was supplied as the type argument for that parameter.

Expression tree types

Expression trees permit lambda expressions to be represented as data structures instead of executable code. Expression trees are values of ***expression tree types*** of the form `System.Linq.Expressions.Expression<D>`,

where `D` is any delegate type. For the remainder of this specification we will refer to these types using the shorthand `Expression<D>`.

If a conversion exists from a lambda expression to a delegate type `D`, a conversion also exists to the expression tree type `Expression<D>`. Whereas the conversion of a lambda expression to a delegate type generates a delegate that references executable code for the lambda expression, conversion to an expression tree type creates an expression tree representation of the lambda expression.

Expression trees are efficient in-memory data representations of lambda expressions and make the structure of the lambda expression transparent and explicit.

Just like a delegate type `D`, `Expression<D>` is said to have parameter and return types, which are the same as those of `D`.

The following example represents a lambda expression both as executable code and as an expression tree. Because a conversion exists to `Func<int,int>`, a conversion also exists to `Expression<Func<int,int>>`:

```
Func<int,int> del = x => x + 1;                  // Code

Expression<Func<int,int>> exp = x => x + 1;      // Data
```

Following these assignments, the delegate `del` references a method that returns `x + 1`, and the expression tree `exp` references a data structure that describes the expression `x => x + 1`.

The exact definition of the generic type `Expression<D>` as well as the precise rules for constructing an expression tree when a lambda expression is converted to an expression tree type, are both outside the scope of this specification.

Two things are important to make explicit:

- Not all lambda expressions can be converted to expression trees. For instance, lambda expressions with statement bodies, and lambda expressions containing assignment expressions cannot be represented. In these cases, a conversion still exists, but will fail at compile-time. These exceptions are detailed in Anonymous function conversions.
- `Expression<D>` offers an instance method `Compile` which produces a delegate of type `D`:

  ```
  Func<int,int> del2 = exp.Compile();
  ```

  Invoking this delegate causes the code represented by the expression tree to be executed. Thus, given the definitions above, del and del2 are equivalent, and the following two statements will have the same effect:

  ```
  int i1 = del(1);

  int i2 = del2(1);
  ```

  After executing this code, `i1` and `i2` will both have the value `2`.

Variables

18.12.2021 • 32 minutes to read

Variables represent storage locations. Every variable has a type that determines what values can be stored in the variable. C# is a type-safe language, and the C# compiler guarantees that values stored in variables are always of the appropriate type. The value of a variable can be changed through assignment or through use of the `++` and `--` operators.

A variable must be ***definitely assigned*** (Definite assignment) before its value can be obtained.

As described in the following sections, variables are either ***initially assigned*** or ***initially unassigned***. An initially assigned variable has a well-defined initial value and is always considered definitely assigned. An initially unassigned variable has no initial value. For an initially unassigned variable to be considered definitely assigned at a certain location, an assignment to the variable must occur in every possible execution path leading to that location.

Variable categories

C# defines seven categories of variables: static variables, instance variables, array elements, value parameters, reference parameters, output parameters, and local variables. The sections that follow describe each of these categories.

In the example

```
class A
{
    public static int x;
    int y;

    void F(int[] v, int a, ref int b, out int c) {
        int i = 1;
        c = a + b++;
    }
}
```

`x` is a static variable, `y` is an instance variable, `v[0]` is an array element, `a` is a value parameter, `b` is a reference parameter, `c` is an output parameter, and `i` is a local variable.

Static variables

A field declared with the `static` modifier is called a ***static variable***. A static variable comes into existence before execution of the static constructor (Static constructors) for its containing type, and ceases to exist when the associated application domain ceases to exist.

The initial value of a static variable is the default value (Default values) of the variable's type.

For purposes of definite assignment checking, a static variable is considered initially assigned.

Instance variables

A field declared without the `static` modifier is called an ***instance variable***.

Instance variables in classes

An instance variable of a class comes into existence when a new instance of that class is created, and ceases to exist when there are no references to that instance and the instance's destructor (if any) has executed.

The initial value of an instance variable of a class is the default value (Default values) of the variable's type.

For the purpose of definite assignment checking, an instance variable of a class is considered initially assigned.

Instance variables in structs

An instance variable of a struct has exactly the same lifetime as the struct variable to which it belongs. In other words, when a variable of a struct type comes into existence or ceases to exist, so too do the instance variables of the struct.

The initial assignment state of an instance variable of a struct is the same as that of the containing struct variable. In other words, when a struct variable is considered initially assigned, so too are its instance variables, and when a struct variable is considered initially unassigned, its instance variables are likewise unassigned.

Array elements

The elements of an array come into existence when an array instance is created, and cease to exist when there are no references to that array instance.

The initial value of each of the elements of an array is the default value (Default values) of the type of the array elements.

For the purpose of definite assignment checking, an array element is considered initially assigned.

Value parameters

A parameter declared without a `ref` or `out` modifier is a ***value parameter***.

A value parameter comes into existence upon invocation of the function member (method, instance constructor, accessor, or operator) or anonymous function to which the parameter belongs, and is initialized with the value of the argument given in the invocation. A value parameter normally ceases to exist upon return of the function member or anonymous function. However, if the value parameter is captured by an anonymous function (Anonymous function expressions), its life time extends at least until the delegate or expression tree created from that anonymous function is eligible for garbage collection.

For the purpose of definite assignment checking, a value parameter is considered initially assigned.

Reference parameters

A parameter declared with a `ref` modifier is a ***reference parameter***.

A reference parameter does not create a new storage location. Instead, a reference parameter represents the same storage location as the variable given as the argument in the function member or anonymous function invocation. Thus, the value of a reference parameter is always the same as the underlying variable.

The following definite assignment rules apply to reference parameters. Note the different rules for output parameters described in Output parameters.

- A variable must be definitely assigned (Definite assignment) before it can be passed as a reference parameter in a function member or delegate invocation.
- Within a function member or anonymous function, a reference parameter is considered initially assigned.

Within an instance method or instance accessor of a struct type, the `this` keyword behaves exactly as a reference parameter of the struct type (This access).

Output parameters

A parameter declared with an `out` modifier is an ***output parameter***.

An output parameter does not create a new storage location. Instead, an output parameter represents the same storage location as the variable given as the argument in the function member or delegate invocation. Thus, the value of an output parameter is always the same as the underlying variable.

The following definite assignment rules apply to output parameters. Note the different rules for reference parameters described in Reference parameters.

- A variable need not be definitely assigned before it can be passed as an output parameter in a function member or delegate invocation.
- Following the normal completion of a function member or delegate invocation, each variable that was passed as an output parameter is considered assigned in that execution path.
- Within a function member or anonymous function, an output parameter is considered initially unassigned.
- Every output parameter of a function member or anonymous function must be definitely assigned (Definite assignment) before the function member or anonymous function returns normally.

Within an instance constructor of a struct type, the `this` keyword behaves exactly as an output parameter of the struct type (This access).

Local variables

A ***local variable*** is declared by a *local_variable_declaration*, which may occur in a *block*, a *for_statement*, a *switch_statement* or a *using_statement*, or by a *foreach_statement* or a *specific_catch_clause* for a *try_statement*.

The lifetime of a local variable is the portion of program execution during which storage is guaranteed to be reserved for it. This lifetime extends at least from entry into the *block*, *for_statement*, *switch_statement*, *using_statement*, *foreach_statement*, or *specific_catch_clause* with which it is associated, until execution of that *block*, *for_statement*, *switch_statement*, *using_statement*, *foreach_statement*, or *specific_catch_clause* ends in any way. (Entering an enclosed *block* or calling a method suspends, but does not end, execution of the current *block*, *for_statement*, *switch_statement*, *using_statement*, *foreach_statement*, or *specific_catch_clause*.) If the local variable is captured by an anonymous function (Captured outer variables), its lifetime extends at least until the delegate or expression tree created from the anonymous function, along with any other objects that come to reference the captured variable, are eligible for garbage collection.

If the parent *block*, *for_statement*, *switch_statement*, *using_statement*, *foreach_statement*, or *specific_catch_clause* is entered recursively, a new instance of the local variable is created each time, and its *local_variable_initializer*, if any, is evaluated each time.

A local variable introduced by a *local_variable_declaration* is not automatically initialized and thus has no default value. For the purpose of definite assignment checking, a local variable introduced by a *local_variable_declaration* is considered initially unassigned. A *local_variable_declaration* may include a *local_variable_initializer*, in which case the variable is considered definitely assigned only after the initializing expression (Declaration statements).

Within the scope of a local variable introduced by a *local_variable_declaration*, it is a compile-time error to refer to that local variable in a textual position that precedes its *local_variable_declarator*. If the local variable declaration is implicit (Local variable declarations), it is also an error to refer to the variable within its *local_variable_declarator*.

A local variable introduced by a *foreach_statement* or a *specific_catch_clause* is considered definitely assigned in its entire scope.

The actual lifetime of a local variable is implementation-dependent. For example, a compiler might statically determine that a local variable in a block is only used for a small portion of that block. Using this analysis, the compiler could generate code that results in the variable's storage having a shorter lifetime than its containing block.

The storage referred to by a local reference variable is reclaimed independently of the lifetime of that local reference variable (Automatic memory management).

Default values

The following categories of variables are automatically initialized to their default values:

- Static variables.
- Instance variables of class instances.
- Array elements.

The default value of a variable depends on the type of the variable and is determined as follows:

- For a variable of a *value_type*, the default value is the same as the value computed by the *value_type*'s default constructor (Default constructors).
- For a variable of a *reference_type*, the default value is `null`.

Initialization to default values is typically done by having the memory manager or garbage collector initialize memory to all-bits-zero before it is allocated for use. For this reason, it is convenient to use all-bits-zero to represent the null reference.

Definite assignment

At a given location in the executable code of a function member, a variable is said to be ***definitely assigned*** if the compiler can prove, by a particular static flow analysis (Precise rules for determining definite assignment), that the variable has been automatically initialized or has been the target of at least one assignment. Informally stated, the rules of definite assignment are:

- An initially assigned variable (Initially assigned variables) is always considered definitely assigned.
- An initially unassigned variable (Initially unassigned variables) is considered definitely assigned at a given location if all possible execution paths leading to that location contain at least one of the following:
  - A simple assignment (Simple assignment) in which the variable is the left operand.
  - An invocation expression (Invocation expressions) or object creation expression (Object creation expressions) that passes the variable as an output parameter.
  - For a local variable, a local variable declaration (Local variable declarations) that includes a variable initializer.

The formal specification underlying the above informal rules is described in Initially assigned variables, Initially unassigned variables, and Precise rules for determining definite assignment.

The definite assignment states of instance variables of a *struct_type* variable are tracked individually as well as collectively. In addition to the rules above, the following rules apply to *struct_type* variables and their instance variables:

- An instance variable is considered definitely assigned if its containing *struct_type* variable is considered definitely assigned.
- A *struct_type* variable is considered definitely assigned if each of its instance variables is considered definitely assigned.

Definite assignment is a requirement in the following contexts:

- A variable must be definitely assigned at each location where its value is obtained. This ensures that undefined values never occur. The occurrence of a variable in an expression is considered to obtain the value of the variable, except when
  - the variable is the left operand of a simple assignment,
  - the variable is passed as an output parameter, or
  - the variable is a *struct_type* variable and occurs as the left operand of a member access.
- A variable must be definitely assigned at each location where it is passed as a reference parameter. This ensures that the function member being invoked can consider the reference parameter initially assigned.
- All output parameters of a function member must be definitely assigned at each location where the function

member returns (through a `return` statement or through execution reaching the end of the function member body). This ensures that function members do not return undefined values in output parameters, thus enabling the compiler to consider a function member invocation that takes a variable as an output parameter equivalent to an assignment to the variable.

- The `this` variable of a *struct_type* instance constructor must be definitely assigned at each location where that instance constructor returns.

Initially assigned variables

The following categories of variables are classified as initially assigned:

- Static variables.
- Instance variables of class instances.
- Instance variables of initially assigned struct variables.
- Array elements.
- Value parameters.
- Reference parameters.
- Variables declared in a `catch` clause or a `foreach` statement.

Initially unassigned variables

The following categories of variables are classified as initially unassigned:

- Instance variables of initially unassigned struct variables.
- Output parameters, including the `this` variable of struct instance constructors.
- Local variables, except those declared in a `catch` clause or a `foreach` statement.

Precise rules for determining definite assignment

In order to determine that each used variable is definitely assigned, the compiler must use a process that is equivalent to the one described in this section.

The compiler processes the body of each function member that has one or more initially unassigned variables. For each initially unassigned variable *v*, the compiler determines a ***definite assignment state*** for *v* at each of the following points in the function member:

- At the beginning of each statement
- At the end point (End points and reachability) of each statement
- On each arc which transfers control to another statement or to the end point of a statement
- At the beginning of each expression
- At the end of each expression

The definite assignment state of *v* can be either:

- Definitely assigned. This indicates that on all possible control flows to this point, *v* has been assigned a value.
- Not definitely assigned. For the state of a variable at the end of an expression of type `bool`, the state of a variable that isn't definitely assigned may (but doesn't necessarily) fall into one of the following sub-states:
  - Definitely assigned after true expression. This state indicates that *v* is definitely assigned if the boolean expression evaluated as true, but is not necessarily assigned if the boolean expression evaluated as false.
  - Definitely assigned after false expression. This state indicates that *v* is definitely assigned if the boolean expression evaluated as false, but is not necessarily assigned if the boolean expression evaluated as true.

The following rules govern how the state of a variable *v* is determined at each location.

General rules for statements

- *v* is not definitely assigned at the beginning of a function member body.
- *v* is definitely assigned at the beginning of any unreachable statement.
- The definite assignment state of *v* at the beginning of any other statement is determined by checking the definite assignment state of *v* on all control flow transfers that target the beginning of that statement. If (and only if) *v* is definitely assigned on all such control flow transfers, then *v* is definitely assigned at the beginning of the statement. The set of possible control flow transfers is determined in the same way as for checking statement reachability (End points and reachability).
- The definite assignment state of *v* at the end point of a block, `checked`, `unchecked`, `if`, `while`, `do`, `for`, `foreach`, `lock`, `using`, or `switch` statement is determined by checking the definite assignment state of *v* on all control flow transfers that target the end point of that statement. If *v* is definitely assigned on all such control flow transfers, then *v* is definitely assigned at the end point of the statement. Otherwise; *v* is not definitely assigned at the end point of the statement. The set of possible control flow transfers is determined in the same way as for checking statement reachability (End points and reachability).

Block statements, checked, and unchecked statements

The definite assignment state of *v* on the control transfer to the first statement of the statement list in the block (or to the end point of the block, if the statement list is empty) is the same as the definite assignment statement of *v* before the block, `checked`, or `unchecked` statement.

Expression statements

For an expression statement *stmt* that consists of the expression *expr*:

- *v* has the same definite assignment state at the beginning of *expr* as at the beginning of *stmt*.
- If *v* if definitely assigned at the end of *expr*, it is definitely assigned at the end point of *stmt*; otherwise; it is not definitely assigned at the end point of *stmt*.

Declaration statements

- If *stmt* is a declaration statement without initializers, then *v* has the same definite assignment state at the end point of *stmt* as at the beginning of *stmt*.
- If *stmt* is a declaration statement with initializers, then the definite assignment state for *v* is determined as if *stmt* were a statement list, with one assignment statement for each declaration with an initializer (in the order of declaration).

If statements

For an `if` statement *stmt* of the form:

```
if ( expr ) then_stmt else else_stmt
```

- *v* has the same definite assignment state at the beginning of *expr* as at the beginning of *stmt*.
- If *v* is definitely assigned at the end of *expr*, then it is definitely assigned on the control flow transfer to *then_stmt* and to either *else_stmt* or to the end-point of *stmt* if there is no else clause.
- If *v* has the state "definitely assigned after true expression" at the end of *expr*, then it is definitely assigned on the control flow transfer to *then_stmt*, and not definitely assigned on the control flow transfer to either *else_stmt* or to the end-point of *stmt* if there is no else clause.
- If *v* has the state "definitely assigned after false expression" at the end of *expr*, then it is definitely assigned on the control flow transfer to *else_stmt*, and not definitely assigned on the control flow transfer to *then_stmt*. It is definitely assigned at the end-point of *stmt* if and only if it is definitely assigned at the end-point of *then_stmt*.
- Otherwise, *v* is considered not definitely assigned on the control flow transfer to either the *then_stmt* or *else_stmt*, or to the end-point of *stmt* if there is no else clause.

Switch statements

In a `switch` statement *stmt* with a controlling expression *expr*:

- The definite assignment state of *v* at the beginning of *expr* is the same as the state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* on the control flow transfer to a reachable switch block statement list is the same as the definite assignment state of *v* at the end of *expr*.

While statements

For a `while` statement *stmt* of the form:

```
while ( expr ) while_body
```

- *v* has the same definite assignment state at the beginning of *expr* as at the beginning of *stmt*.
- If *v* is definitely assigned at the end of *expr*, then it is definitely assigned on the control flow transfer to *while_body* and to the end point of *stmt*.
- If *v* has the state "definitely assigned after true expression" at the end of *expr*, then it is definitely assigned on the control flow transfer to *while_body*, but not definitely assigned at the end-point of *stmt*.
- If *v* has the state "definitely assigned after false expression" at the end of *expr*, then it is definitely assigned on the control flow transfer to the end point of *stmt*, but not definitely assigned on the control flow transfer to *while_body*.

Do statements

For a `do` statement *stmt* of the form:

```
do do_body while ( expr ) ;
```

- *v* has the same definite assignment state on the control flow transfer from the beginning of *stmt* to *do_body* as at the beginning of *stmt*.
- *v* has the same definite assignment state at the beginning of *expr* as at the end point of *do_body*.
- If *v* is definitely assigned at the end of *expr*, then it is definitely assigned on the control flow transfer to the end point of *stmt*.
- If *v* has the state "definitely assigned after false expression" at the end of *expr*, then it is definitely assigned on the control flow transfer to the end point of *stmt*.

For statements

Definite assignment checking for a `for` statement of the form:

```
for ( for_initializer ; for_condition ; for_iterator ) embedded_statement
```

is done as if the statement were written:

```
{
    for_initializer ;
    while ( for_condition ) {
        embedded_statement ;
        for_iterator ;
    }
}
```

If the *for_condition* is omitted from the `for` statement, then evaluation of definite assignment proceeds as if *for_condition* were replaced with `true` in the above expansion.

Break, continue, and goto statements

The definite assignment state of *v* on the control flow transfer caused by a `break`, `continue`, or `goto` statement is the same as the definite assignment state of *v* at the beginning of the statement.

Throw statements

For a statement *stmt* of the form

```
throw expr ;
```

The definite assignment state of *v* at the beginning of *expr* is the same as the definite assignment state of *v* at the beginning of *stmt*.

Return statements

For a statement *stmt* of the form

```
return expr ;
```

- The definite assignment state of *v* at the beginning of *expr* is the same as the definite assignment state of *v* at the beginning of *stmt*.
- If *v* is an output parameter, then it must be definitely assigned either:
  - after *expr*
  - or at the end of the `finally` block of a `try` - `finally` or `try` - `catch` - `finally` that encloses the `return` statement.

For a statement stmt of the form:

```
return ;
```

- If *v* is an output parameter, then it must be definitely assigned either:
  - before *stmt*
  - or at the end of the `finally` block of a `try` - `finally` or `try` - `catch` - `finally` that encloses the `return` statement.

Try-catch statements

For a statement *stmt* of the form:

```
try try_block
catch(...) catch_block_1
...
catch(...) catch_block_n
```

- The definite assignment state of *v* at the beginning of *try_block* is the same as the definite assignment state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* at the beginning of *catch_block_i* (for any *i*) is the same as the definite assignment state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* at the end-point of *stmt* is definitely assigned if (and only if) *v* is definitely assigned at the end-point of *try_block* and every *catch_block_i* (for every *i* from 1 to *n*).

Try-finally statements

For a `try` statement *stmt* of the form:

```
try try_block finally finally_block
```

- The definite assignment state of *v* at the beginning of *try_block* is the same as the definite assignment state of *v* at the beginning of *stmt*.

- The definite assignment state of *v* at the beginning of *finally_block* is the same as the definite assignment state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* at the end-point of *stmt* is definitely assigned if (and only if) at least one of the following is true:
  - *v* is definitely assigned at the end-point of *try_block*
  - *v* is definitely assigned at the end-point of *finally_block*

If a control flow transfer (for example, a `goto` statement) is made that begins within *try_block*, and ends outside of *try_block*, then *v* is also considered definitely assigned on that control flow transfer if *v* is definitely assigned at the end-point of *finally_block*. (This is not an only if—if *v* is definitely assigned for another reason on this control flow transfer, then it is still considered definitely assigned.)

Try-catch-finally statements

Definite assignment analysis for a `try` - `catch` - `finally` statement of the form:

```
try try_block
catch(...) catch_block_1
...
catch(...) catch_block_n
finally *finally_block*
```

is done as if the statement were a `try` - `finally` statement enclosing a `try` - `catch` statement:

```
try {
    try try_block
    catch(...) catch_block_1
    ...
    catch(...) catch_block_n
}
finally finally_block
```

The following example demonstrates how the different blocks of a `try` statement (The try statement) affect definite assignment.

```
class A
{
    static void F() {
        int i, j;
        try {
            goto LABEL;
            // neither i nor j definitely assigned
            i = 1;
            // i definitely assigned
        }

        catch {
            // neither i nor j definitely assigned
            i = 3;
            // i definitely assigned
        }

        finally {
            // neither i nor j definitely assigned
            j = 5;
            // j definitely assigned
            }
        // i and j definitely assigned
        LABEL:;
        // j definitely assigned

    }
}
```

Foreach statements

For a `foreach` statement *stmt* of the form:

```
foreach ( type identifier in expr ) embedded_statement
```

- The definite assignment state of *v* at the beginning of *expr* is the same as the state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* on the control flow transfer to *embedded_statement* or to the end point of *stmt* is the same as the state of *v* at the end of *expr*.

Using statements

For a `using` statement *stmt* of the form:

```
using ( resource_acquisition ) embedded_statement
```

- The definite assignment state of *v* at the beginning of *resource_acquisition* is the same as the state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* on the control flow transfer to *embedded_statement* is the same as the state of *v* at the end of *resource_acquisition*.

Lock statements

For a `lock` statement *stmt* of the form:

```
lock ( expr ) embedded_statement
```

- The definite assignment state of *v* at the beginning of *expr* is the same as the state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* on the control flow transfer to *embedded_statement* is the same as the

state of *v* at the end of *expr*.

Yield statements

For a `yield return` statement *stmt* of the form:

```
yield return expr ;
```

- The definite assignment state of *v* at the beginning of *expr* is the same as the state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* at the end of *stmt* is the same as the state of *v* at the end of *expr*.
- A `yield break` statement has no effect on the definite assignment state.

General rules for simple expressions

The following rule applies to these kinds of expressions: literals (Literals), simple names (Simple names), member access expressions (Member access), non-indexed base access expressions (Base access), `typeof` expressions (The typeof operator), default value expressions (Default value expressions) and `nameof` expressions (Nameof expressions).

- The definite assignment state of *v* at the end of such an expression is the same as the definite assignment state of *v* at the beginning of the expression.

General rules for expressions with embedded expressions

The following rules apply to these kinds of expressions: parenthesized expressions (Parenthesized expressions), element access expressions (Element access), base access expressions with indexing (Base access), increment and decrement expressions (Postfix increment and decrement operators, Prefix increment and decrement operators), cast expressions (Cast expressions), unary `+`, `-`, `~`, `*` expressions, binary `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `<`, `<=`, `>`, `>=`, `==`, `!=`, `is`, `as`, `&`, `|`, `^` expressions (Arithmetic operators, Shift operators, Relational and type-testing operators, Logical operators), compound assignment expressions (Compound assignment), `checked` and `unchecked` expressions (The checked and unchecked operators), plus array and delegate creation expressions (The new operator).

Each of these expressions has one or more sub-expressions that are unconditionally evaluated in a fixed order. For example, the binary `%` operator evaluates the left hand side of the operator, then the right hand side. An indexing operation evaluates the indexed expression, and then evaluates each of the index expressions, in order from left to right. For an expression *expr*, which has sub-expressions *e1, e2, ..., eN*, evaluated in that order:

- The definite assignment state of *v* at the beginning of *e1* is the same as the definite assignment state at the beginning of *expr*.
- The definite assignment state of *v* at the beginning of *ei* (*i* greater than one) is the same as the definite assignment state at the end of the previous sub-expression.
- The definite assignment state of *v* at the end of *expr* is the same as the definite assignment state at the end of *eN*

Invocation expressions and object creation expressions

For an invocation expression *expr* of the form:

```
primary_expression ( arg1 , arg2 , ... , argN )
```

or an object creation expression of the form:

```
new type ( arg1 , arg2 , ... , argN )
```

- For an invocation expression, the definite assignment state of *v* before *primary_expression* is the same as the

state of *v* before *expr*.

- For an invocation expression, the definite assignment state of *v* before *arg1* is the same as the state of *v* after *primary_expression*.
- For an object creation expression, the definite assignment state of *v* before *arg1* is the same as the state of *v* before *expr*.
- For each argument *argi*, the definite assignment state of *v* after *argi* is determined by the normal expression rules, ignoring any `ref` or `out` modifiers.
- For each argument *argi* for any *i* greater than one, the definite assignment state of *v* before *argi* is the same as the state of *v* after the previous *arg*.
- If the variable *v* is passed as an `out` argument (i.e., an argument of the form `out v`) in any of the arguments, then the state of *v* after *expr* is definitely assigned. Otherwise; the state of *v* after *expr* is the same as the state of *v* after *argN*.
- For array initializers (Array creation expressions), object initializers (Object initializers), collection initializers (Collection initializers) and anonymous object initializers (Anonymous object creation expressions), the definite assignment state is determined by the expansion that these constructs are defined in terms of.

Simple assignment expressions

For an expression *expr* of the form `w = expr_rhs`:

- The definite assignment state of *v* before *expr_rhs* is the same as the definite assignment state of *v* before *expr*.
- The definite assignment state of *v* after *expr* is determined by:
  - If *w* is the same variable as *v*, then the definite assignment state of *v* after *expr* is definitely assigned.
  - Otherwise, if the assignment occurs within the instance constructor of a struct type, if *w* is a property access designating an automatically implemented property *P* on the instance being constructed and *v* is the hidden backing field of *P*, then the definite assignment state of *v* after *expr* is definitely assigned.
  - Otherwise, the definite assignment state of *v* after *expr* is the same as the definite assignment state of *v* after *expr_rhs*.

&& (conditional AND) expressions

For an expression *expr* of the form `expr_first && expr_second`:

- The definite assignment state of *v* before *expr_first* is the same as the definite assignment state of *v* before *expr*.
- The definite assignment state of *v* before *expr_second* is definitely assigned if the state of *v* after *expr_first* is either definitely assigned or "definitely assigned after true expression". Otherwise, it is not definitely assigned.
- The definite assignment state of *v* after *expr* is determined by:
  - If *expr_first* is a constant expression with the value `false`, then the definite assignment state of *v* after *expr* is the same as the definite assignment state of *v* after *expr_first*.
  - Otherwise, if the state of *v* after *expr_first* is definitely assigned, then the state of *v* after *expr* is definitely assigned.
  - Otherwise, if the state of *v* after *expr_second* is definitely assigned, and the state of *v* after *expr_first* is "definitely assigned after false expression", then the state of *v* after *expr* is definitely assigned.
  - Otherwise, if the state of *v* after *expr_second* is definitely assigned or "definitely assigned after true expression", then the state of *v* after *expr* is "definitely assigned after true expression".
  - Otherwise, if the state of *v* after *expr_first* is "definitely assigned after false expression", and the state of *v* after *expr_second* is "definitely assigned after false expression", then the state of *v* after *expr* is "definitely assigned after false expression".
  - Otherwise, the state of *v* after *expr* is not definitely assigned.

In the example

```
class A
{
    static void F(int x, int y) {
        int i;
        if (x >= 0 && (i = y) >= 0) {
            // i definitely assigned
        }
        else {
            // i not definitely assigned
        }
        // i not definitely assigned
    }
}
```

the variable `i` is considered definitely assigned in one of the embedded statements of an `if` statement but not in the other. In the `if` statement in method `F`, the variable `i` is definitely assigned in the first embedded statement because execution of the expression `(i = y)` always precedes execution of this embedded statement. In contrast, the variable `i` is not definitely assigned in the second embedded statement, since `x >= 0` might have tested false, resulting in the variable `i` being unassigned.

|| (conditional OR) expressions

For an expression *expr* of the form `expr_first || expr_second`:

- The definite assignment state of *v* before *expr_first* is the same as the definite assignment state of *v* before *expr*.
- The definite assignment state of *v* before *expr_second* is definitely assigned if the state of *v* after *expr_first* is either definitely assigned or "definitely assigned after false expression". Otherwise, it is not definitely assigned.
- The definite assignment statement of *v* after *expr* is determined by:
  - If *expr_first* is a constant expression with the value `true`, then the definite assignment state of *v* after *expr* is the same as the definite assignment state of *v* after *expr_first*.
  - Otherwise, if the state of *v* after *expr_first* is definitely assigned, then the state of *v* after *expr* is definitely assigned.
  - Otherwise, if the state of *v* after *expr_second* is definitely assigned, and the state of *v* after *expr_first* is "definitely assigned after true expression", then the state of *v* after *expr* is definitely assigned.
  - Otherwise, if the state of *v* after *expr_second* is definitely assigned or "definitely assigned after false expression", then the state of *v* after *expr* is "definitely assigned after false expression".
  - Otherwise, if the state of *v* after *expr_first* is "definitely assigned after true expression", and the state of *v* after *expr_second* is "definitely assigned after true expression", then the state of *v* after *expr* is "definitely assigned after true expression".
  - Otherwise, the state of *v* after *expr* is not definitely assigned.

In the example

```
class A
{
    static void G(int x, int y) {
        int i;
        if (x >= 0 || (i = y) >= 0) {
            // i not definitely assigned
        }
        else {
            // i definitely assigned
        }
        // i not definitely assigned
    }
}
```

the variable `i` is considered definitely assigned in one of the embedded statements of an `if` statement but not in the other. In the `if` statement in method `G`, the variable `i` is definitely assigned in the second embedded statement because execution of the expression `(i = y)` always precedes execution of this embedded statement. In contrast, the variable `i` is not definitely assigned in the first embedded statement, since `x >= 0` might have tested true, resulting in the variable `i` being unassigned.

! (logical negation) expressions

For an expression *expr* of the form `! expr_operand`:

- The definite assignment state of *v* before *expr_operand* is the same as the definite assignment state of *v* before *expr*.
- The definite assignment state of *v* after *expr* is determined by:
  - If the state of *v* after *expr_operand *is definitely assigned, then the state of *v* after *expr* is definitely assigned.
  - If the state of *v* after *expr_operand *is not definitely assigned, then the state of *v* after *expr* is not definitely assigned.
  - If the state of *v* after *expr_operand *is "definitely assigned after false expression", then the state of *v* after *expr* is "definitely assigned after true expression".
  - If the state of *v* after *expr_operand *is "definitely assigned after true expression", then the state of *v* after *expr* is "definitely assigned after false expression".

?? (null coalescing) expressions

For an expression *expr* of the form `expr_first ?? expr_second`:

- The definite assignment state of *v* before *expr_first* is the same as the definite assignment state of *v* before *expr*.
- The definite assignment state of *v* before *expr_second* is the same as the definite assignment state of *v* after *expr_first*.
- The definite assignment statement of *v* after *expr* is determined by:
  - If *expr_first* is a constant expression (Constant expressions) with value null, then the state of *v* after *expr* is the same as the state of *v* after *expr_second*.
- Otherwise, the state of *v* after *expr* is the same as the definite assignment state of *v* after *expr_first*.

?: (conditional) expressions

For an expression *expr* of the form `expr_cond ? expr_true : expr_false`:

- The definite assignment state of *v* before *expr_cond* is the same as the state of *v* before *expr*.
- The definite assignment state of *v* before *expr_true* is definitely assigned if and only if one of the following holds:
  - *expr_cond* is a constant expression with the value `false`
  - the state of *v* after *expr_cond* is definitely assigned or "definitely assigned after true expression".

- The definite assignment state of *v* before *expr_false* is definitely assigned if and only if one of the following holds:
  - *expr_cond* is a constant expression with the value `true`
  - the state of *v* after *expr_cond* is definitely assigned or "definitely assigned after false expression".
- The definite assignment state of *v* after *expr* is determined by:
  - If *expr_cond* is a constant expression (Constant expressions) with value `true` then the state of *v* after *expr* is the same as the state of *v* after *expr_true*.
  - Otherwise, if *expr_cond* is a constant expression (Constant expressions) with value `false` then the state of *v* after *expr* is the same as the state of *v* after *expr_false*.
  - Otherwise, if the state of *v* after *expr_true* is definitely assigned and the state of *v* after *expr_false* is definitely assigned, then the state of *v* after *expr* is definitely assigned.
  - Otherwise, the state of *v* after *expr* is not definitely assigned.

Anonymous functions

For a *lambda_expression* or *anonymous_method_expression expr* with a body (either *block* or *expression*) *body*:

- The definite assignment state of an outer variable *v* before *body* is the same as the state of *v* before *expr*. That is, definite assignment state of outer variables is inherited from the context of the anonymous function.
- The definite assignment state of an outer variable *v* after *expr* is the same as the state of *v* before *expr*.

The example

```
delegate bool Filter(int i);

void F() {
    int max;

    // Error, max is not definitely assigned
    Filter f = (int n) => n < max;

    max = 5;
    DoWork(f);
}
```

generates a compile-time error since `max` is not definitely assigned where the anonymous function is declared. The example

```
delegate void D();

void F() {
    int n;
    D d = () => { n = 1; };

    d();

    // Error, n is not definitely assigned
    Console.WriteLine(n);
}
```

also generates a compile-time error since the assignment to `n` in the anonymous function has no affect on the definite assignment state of `n` outside the anonymous function.

Variable references

A *variable_reference* is an *expression* that is classified as a variable. A *variable_reference* denotes a storage location that can be accessed both to fetch the current value and to store a new value.

```
variable_reference
    : expression
    ;
```

In C and C++, a *variable_reference* is known as an *lvalue*.

Atomicity of variable references

Reads and writes of the following data types are atomic: `bool`, `char`, `byte`, `sbyte`, `short`, `ushort`, `uint`, `int`, `float`, and reference types. In addition, reads and writes of enum types with an underlying type in the previous list are also atomic. Reads and writes of other types, including `long`, `ulong`, `double`, and `decimal`, as well as user-defined types, are not guaranteed to be atomic. Aside from the library functions designed for that purpose, there is no guarantee of atomic read-modify-write, such as in the case of increment or decrement.

Conversions

18.12.2021 • 47 minutes to read

A ***conversion*** enables an expression to be treated as being of a particular type. A conversion may cause an expression of a given type to be treated as having a different type, or it may cause an expression without a type to get a type. Conversions can be ***implicit*** or ***explicit***, and this determines whether an explicit cast is required. For instance, the conversion from type `int` to type `long` is implicit, so expressions of type `int` can implicitly be treated as type `long`. The opposite conversion, from type `long` to type `int`, is explicit and so an explicit cast is required.

```
int a = 123;
long b = a;         // implicit conversion from int to long
int c = (int) b;    // explicit conversion from long to int
```

Some conversions are defined by the language. Programs may also define their own conversions (User-defined conversions).

Implicit conversions

The following conversions are classified as implicit conversions:

- Identity conversions
- Implicit numeric conversions
- Implicit enumeration conversions
- Implicit interpolated string conversions
- Implicit nullable conversions
- Null literal conversions
- Implicit reference conversions
- Boxing conversions
- Implicit dynamic conversions
- Implicit constant expression conversions
- User-defined implicit conversions
- Anonymous function conversions
- Method group conversions

Implicit conversions can occur in a variety of situations, including function member invocations (Compile-time checking of dynamic overload resolution), cast expressions (Cast expressions), and assignments (Assignment operators).

The pre-defined implicit conversions always succeed and never cause exceptions to be thrown. Properly designed user-defined implicit conversions should exhibit these characteristics as well.

For the purposes of conversion, the types `object` and `dynamic` are considered equivalent.

However, dynamic conversions (Implicit dynamic conversions and Explicit dynamic conversions) apply only to expressions of type `dynamic` (The dynamic type).

Identity conversion

An identity conversion converts from any type to the same type. This conversion exists such that an entity that

already has a required type can be said to be convertible to that type.

- Because `object` and `dynamic` are considered equivalent there is an identity conversion between `object` and `dynamic`, and between constructed types that are the same when replacing all occurrences of `dynamic` with `object`.

Implicit numeric conversions

The implicit numeric conversions are:

- From `sbyte` to `short`, `int`, `long`, `float`, `double`, or `decimal`.
- From `byte` to `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, or `decimal`.
- From `short` to `int`, `long`, `float`, `double`, or `decimal`.
- From `ushort` to `int`, `uint`, `long`, `ulong`, `float`, `double`, or `decimal`.
- From `int` to `long`, `float`, `double`, or `decimal`.
- From `uint` to `long`, `ulong`, `float`, `double`, or `decimal`.
- From `long` to `float`, `double`, or `decimal`.
- From `ulong` to `float`, `double`, or `decimal`.
- From `char` to `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, or `decimal`.
- From `float` to `double`.

Conversions from `int`, `uint`, `long`, or `ulong` to `float` and from `long` or `ulong` to `double` may cause a loss of precision, but will never cause a loss of magnitude. The other implicit numeric conversions never lose any information.

There are no implicit conversions to the `char` type, so values of the other integral types do not automatically convert to the `char` type.

Implicit enumeration conversions

An implicit enumeration conversion permits the *decimal_integer_literal* `0` to be converted to any *enum_type* and to any *nullable_type* whose underlying type is an *enum_type*. In the latter case the conversion is evaluated by converting to the underlying *enum_type* and wrapping the result (Nullable types).

Implicit interpolated string conversions

An implicit interpolated string conversion permits an *interpolated_string_expression* (Interpolated strings) to be converted to `System.IFormattable` or `System.FormattableString` (which implements `System.IFormattable`).

When this conversion is applied a string value is not composed from the interpolated string. Instead an instance of `System.FormattableString` is created, as further described in Interpolated strings.

Implicit nullable conversions

Predefined implicit conversions that operate on non-nullable value types can also be used with nullable forms of those types. For each of the predefined implicit identity and numeric conversions that convert from a non-nullable value type `S` to a non-nullable value type `T`, the following implicit nullable conversions exist:

- An implicit conversion from `S?` to `T?`.
- An implicit conversion from `S` to `T?`.

Evaluation of an implicit nullable conversion based on an underlying conversion from `S` to `T` proceeds as follows:

- If the nullable conversion is from `S?` to `T?`:
  - If the source value is null (`HasValue` property is false), the result is the null value of type `T?`.
  - Otherwise, the conversion is evaluated as an unwrapping from `S?` to `S`, followed by the underlying

conversion from `S` to `T`, followed by a wrapping (Nullable types) from `T` to `T?`.

- If the nullable conversion is from `S` to `T?`, the conversion is evaluated as the underlying conversion from `S` to `T` followed by a wrapping from `T` to `T?`.

Null literal conversions

An implicit conversion exists from the `null` literal to any nullable type. This conversion produces the null value (Nullable types) of the given nullable type.

Implicit reference conversions

The implicit reference conversions are:

- From any *reference_type* to `object` and `dynamic`.
- From any *class_type* `S` to any *class_type* `T`, provided `S` is derived from `T`.
- From any *class_type* `S` to any *interface_type* `T`, provided `S` implements `T`.
- From any *interface_type* `S` to any *interface_type* `T`, provided `S` is derived from `T`.
- From an *array_type* `S` with an element type `SE` to an *array_type* `T` with an element type `TE`, provided all of the following are true:
  - `S` and `T` differ only in element type. In other words, `S` and `T` have the same number of dimensions.
  - Both `SE` and `TE` are *reference_type*s.
  - An implicit reference conversion exists from `SE` to `TE`.
- From any *array_type* to `System.Array` and the interfaces it implements.
- From a single-dimensional array type `S[]` to `System.Collections.Generic.IList<T>` and its base interfaces, provided that there is an implicit identity or reference conversion from `S` to `T`.
- From any *delegate_type* to `System.Delegate` and the interfaces it implements.
- From the null literal to any *reference_type*.
- From any *reference_type* to a *reference_type* `T` if it has an implicit identity or reference conversion to a *reference_type* `T0` and `T0` has an identity conversion to `T`.
- From any *reference_type* to an interface or delegate type `T` if it has an implicit identity or reference conversion to an interface or delegate type `T0` and `T0` is variance-convertible (Variance conversion) to `T`.
- Implicit conversions involving type parameters that are known to be reference types. See Implicit conversions involving type parameters for more details on implicit conversions involving type parameters.

The implicit reference conversions are those conversions between *reference_type*s that can be proven to always succeed, and therefore require no checks at run-time.

Reference conversions, implicit or explicit, never change the referential identity of the object being converted. In other words, while a reference conversion may change the type of the reference, it never changes the type or value of the object being referred to.

Boxing conversions

A boxing conversion permits a *value_type* to be implicitly converted to a reference type. A boxing conversion exists from any *non_nullable_value_type* to `object` and `dynamic`, to `System.ValueType` and to any *interface_type* implemented by the *non_nullable_value_type*. Furthermore an *enum_type* can be converted to the type `System.Enum`.

A boxing conversion exists from a *nullable_type* to a reference type, if and only if a boxing conversion exists from the underlying *non_nullable_value_type* to the reference type.

A value type has a boxing conversion to an interface type `I` if it has a boxing conversion to an interface type `I0` and `I0` has an identity conversion to `I`.

A value type has a boxing conversion to an interface type `I` if it has a boxing conversion to an interface or delegate type `I0` and `I0` is variance-convertible (Variance conversion) to `I`.

Boxing a value of a *non_nullable_value_type* consists of allocating an object instance and copying the *value_type* value into that instance. A struct can be boxed to the type `System.ValueType`, since that is a base class for all structs (Inheritance).

Boxing a value of a *nullable_type* proceeds as follows:

- If the source value is null (`HasValue` property is false), the result is a null reference of the target type.
- Otherwise, the result is a reference to a boxed `T` produced by unwrapping and boxing the source value.

Boxing conversions are described further in Boxing conversions.

Implicit dynamic conversions

An implicit dynamic conversion exists from an expression of type `dynamic` to any type `T`. The conversion is dynamically bound (Dynamic binding), which means that an implicit conversion will be sought at run-time from the run-time type of the expression to `T`. If no conversion is found, a run-time exception is thrown.

Note that this implicit conversion seemingly violates the advice in the beginning of Implicit conversions that an implicit conversion should never cause an exception. However it is not the conversion itself, but the *finding* of the conversion that causes the exception. The risk of run-time exceptions is inherent in the use of dynamic binding. If dynamic binding of the conversion is not desired, the expression can be first converted to `object`, and then to the desired type.

The following example illustrates implicit dynamic conversions:

```
object o  = "object";
dynamic d = "dynamic";

string s1 = o; // Fails at compile-time -- no conversion exists
string s2 = d; // Compiles and succeeds at run-time
int i     = d; // Compiles but fails at run-time -- no conversion exists
```

The assignments to `s2` and `i` both employ implicit dynamic conversions, where the binding of the operations is suspended until run-time. At run-time, implicit conversions are sought from the run-time type of `d` -- `string` -- to the target type. A conversion is found to `string` but not to `int`.

Implicit constant expression conversions

An implicit constant expression conversion permits the following conversions:

- A *constant_expression* (Constant expressions) of type `int` can be converted to type `sbyte`, `byte`, `short`, `ushort`, `uint`, or `ulong`, provided the value of the *constant_expression* is within the range of the destination type.
- A *constant_expression* of type `long` can be converted to type `ulong`, provided the value of the *constant_expression* is not negative.

Implicit conversions involving type parameters

The following implicit conversions exist for a given type parameter `T`:

- From `T` to its effective base class `C`, from `T` to any base class of `C`, and from `T` to any interface implemented by `C`. At run-time, if `T` is a value type, the conversion is executed as a boxing conversion. Otherwise, the conversion is executed as an implicit reference conversion or identity conversion.
- From `T` to an interface type `I` in `T`'s effective interface set and from `T` to any base interface of `I`. At run-time, if `T` is a value type, the conversion is executed as a boxing conversion. Otherwise, the conversion is executed as an implicit reference conversion or identity conversion.

- From `T` to a type parameter `U`, provided `T` depends on `U` (Type parameter constraints). At run-time, if `U` is a value type, then `T` and `U` are necessarily the same type and no conversion is performed. Otherwise, if `T` is a value type, the conversion is executed as a boxing conversion. Otherwise, the conversion is executed as an implicit reference conversion or identity conversion.
- From the null literal to `T`, provided `T` is known to be a reference type.
- From `T` to a reference type `I` if it has an implicit conversion to a reference type `S0` and `S0` has an identity conversion to `S`. At run-time the conversion is executed the same way as the conversion to `S0`.
- From `T` to an interface type `I` if it has an implicit conversion to an interface or delegate type `I0` and `I0` is variance-convertible to `I` (Variance conversion). At run-time, if `T` is a value type, the conversion is executed as a boxing conversion. Otherwise, the conversion is executed as an implicit reference conversion or identity conversion.

If `T` is known to be a reference type (Type parameter constraints), the conversions above are all classified as implicit reference conversions (Implicit reference conversions). If `T` is not known to be a reference type, the conversions above are classified as boxing conversions (Boxing conversions).

User-defined implicit conversions

A user-defined implicit conversion consists of an optional standard implicit conversion, followed by execution of a user-defined implicit conversion operator, followed by another optional standard implicit conversion. The exact rules for evaluating user-defined implicit conversions are described in Processing of user-defined implicit conversions.

Anonymous function conversions and method group conversions

Anonymous functions and method groups do not have types in and of themselves, but may be implicitly converted to delegate types or expression tree types. Anonymous function conversions are described in more detail in Anonymous function conversions and method group conversions in Method group conversions.

Explicit conversions

The following conversions are classified as explicit conversions:

- All implicit conversions.
- Explicit numeric conversions.
- Explicit enumeration conversions.
- Explicit nullable conversions.
- Explicit reference conversions.
- Explicit interface conversions.
- Unboxing conversions.
- Explicit dynamic conversions
- User-defined explicit conversions.

Explicit conversions can occur in cast expressions (Cast expressions).

The set of explicit conversions includes all implicit conversions. This means that redundant cast expressions are allowed.

The explicit conversions that are not implicit conversions are conversions that cannot be proven to always succeed, conversions that are known to possibly lose information, and conversions across domains of types sufficiently different to merit explicit notation.

Explicit numeric conversions

The explicit numeric conversions are the conversions from a *numeric_type* to another *numeric_type* for which an implicit numeric conversion (Implicit numeric conversions) does not already exist:

- From `sbyte` to `byte`, `ushort`, `uint`, `ulong`, or `char`.
- From `byte` to `sbyte` and `char`.
- From `short` to `sbyte`, `byte`, `ushort`, `uint`, `ulong`, or `char`.
- From `ushort` to `sbyte`, `byte`, `short`, or `char`.
- From `int` to `sbyte`, `byte`, `short`, `ushort`, `uint`, `ulong`, or `char`.
- From `uint` to `sbyte`, `byte`, `short`, `ushort`, `int`, or `char`.
- From `long` to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `ulong`, or `char`.
- From `ulong` to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, or `char`.
- From `char` to `sbyte`, `byte`, or `short`.
- From `float` to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, or `decimal`.
- From `double` to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, or `decimal`.
- From `decimal` to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, or `double`.

Because the explicit conversions include all implicit and explicit numeric conversions, it is always possible to convert from any *numeric_type* to any other *numeric_type* using a cast expression (Cast expressions).

The explicit numeric conversions possibly lose information or possibly cause exceptions to be thrown. An explicit numeric conversion is processed as follows:

- For a conversion from an integral type to another integral type, the processing depends on the overflow checking context (The checked and unchecked operators) in which the conversion takes place:
  - In a `checked` context, the conversion succeeds if the value of the source operand is within the range of the destination type, but throws a `System.OverflowException` if the value of the source operand is outside the range of the destination type.
  - In an `unchecked` context, the conversion always succeeds, and proceeds as follows.
    - If the source type is larger than the destination type, then the source value is truncated by discarding its "extra" most significant bits. The result is then treated as a value of the destination type.
    - If the source type is smaller than the destination type, then the source value is either sign-extended or zero-extended so that it is the same size as the destination type. Sign-extension is used if the source type is signed; zero-extension is used if the source type is unsigned. The result is then treated as a value of the destination type.
    - If the source type is the same size as the destination type, then the source value is treated as a value of the destination type.
- For a conversion from `decimal` to an integral type, the source value is rounded towards zero to the nearest integral value, and this integral value becomes the result of the conversion. If the resulting integral value is outside the range of the destination type, a `System.OverflowException` is thrown.
- For a conversion from `float` or `double` to an integral type, the processing depends on the overflow checking context (The checked and unchecked operators) in which the conversion takes place:
  - In a `checked` context, the conversion proceeds as follows:
    - If the value of the operand is NaN or infinite, a `System.OverflowException` is thrown.
    - Otherwise, the source operand is rounded towards zero to the nearest integral value. If this integral value is within the range of the destination type then this value is the result of the conversion.
    - Otherwise, a `System.OverflowException` is thrown.
  - In an `unchecked` context, the conversion always succeeds, and proceeds as follows.
    - If the value of the operand is NaN or infinite, the result of the conversion is an unspecified value of the destination type.
    - Otherwise, the source operand is rounded towards zero to the nearest integral value. If this

integral value is within the range of the destination type then this value is the result of the conversion.
    - Otherwise, the result of the conversion is an unspecified value of the destination type.
- For a conversion from `double` to `float`, the `double` value is rounded to the nearest `float` value. If the `double` value is too small to represent as a `float`, the result becomes positive zero or negative zero. If the `double` value is too large to represent as a `float`, the result becomes positive infinity or negative infinity. If the `double` value is NaN, the result is also NaN.
- For a conversion from `float` or `double` to `decimal`, the source value is converted to `decimal` representation and rounded to the nearest number after the 28th decimal place if required (The decimal type). If the source value is too small to represent as a `decimal`, the result becomes zero. If the source value is NaN, infinity, or too large to represent as a `decimal`, a `System.OverflowException` is thrown.
- For a conversion from `decimal` to `float` or `double`, the `decimal` value is rounded to the nearest `double` or `float` value. While this conversion may lose precision, it never causes an exception to be thrown.

Explicit enumeration conversions

The explicit enumeration conversions are:

- From `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, or `decimal` to any *enum_type*.
- From any *enum_type* to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, or `decimal`.
- From any *enum_type* to any other *enum_type*.

An explicit enumeration conversion between two types is processed by treating any participating *enum_type* as the underlying type of that *enum_type*, and then performing an implicit or explicit numeric conversion between the resulting types. For example, given an *enum_type* `E` with and underlying type of `int`, a conversion from `E` to `byte` is processed as an explicit numeric conversion (Explicit numeric conversions) from `int` to `byte`, and a conversion from `byte` to `E` is processed as an implicit numeric conversion (Implicit numeric conversions) from `byte` to `int`.

Explicit nullable conversions

Explicit nullable conversions permit predefined explicit conversions that operate on non-nullable value types to also be used with nullable forms of those types. For each of the predefined explicit conversions that convert from a non-nullable value type `S` to a non-nullable value type `T` (Identity conversion, Implicit numeric conversions, Implicit enumeration conversions, Explicit numeric conversions, and Explicit enumeration conversions), the following nullable conversions exist:

- An explicit conversion from `S?` to `T?`.
- An explicit conversion from `S` to `T?`.
- An explicit conversion from `S?` to `T`.

Evaluation of a nullable conversion based on an underlying conversion from `S` to `T` proceeds as follows:

- If the nullable conversion is from `S?` to `T?`:
    - If the source value is null (`HasValue` property is false), the result is the null value of type `T?`.
    - Otherwise, the conversion is evaluated as an unwrapping from `S?` to `S`, followed by the underlying conversion from `S` to `T`, followed by a wrapping from `T` to `T?`.
- If the nullable conversion is from `S` to `T?`, the conversion is evaluated as the underlying conversion from `S` to `T` followed by a wrapping from `T` to `T?`.
- If the nullable conversion is from `S?` to `T`, the conversion is evaluated as an unwrapping from `S?` to `S` followed by the underlying conversion from `S` to `T`.

Note that an attempt to unwrap a nullable value will throw an exception if the value is `null`.

Explicit reference conversions

The explicit reference conversions are:

- From `object` and `dynamic` to any other *reference_type*.
- From any *class_type* `S` to any *class_type* `T`, provided `S` is a base class of `T`.
- From any *class_type* `S` to any *interface_type* `T`, provided `S` is not sealed and provided `S` does not implement `T`.
- From any *interface_type* `S` to any *class_type* `T`, provided `T` is not sealed or provided `T` implements `S`.
- From any *interface_type* `S` to any *interface_type* `T`, provided `S` is not derived from `T`.
- From an *array_type* `S` with an element type `SE` to an *array_type* `T` with an element type `TE`, provided all of the following are true:
  - `S` and `T` differ only in element type. In other words, `S` and `T` have the same number of dimensions.
  - Both `SE` and `TE` are *reference_type*s.
  - An explicit reference conversion exists from `SE` to `TE`.
- From `System.Array` and the interfaces it implements to any *array_type*.
- From a single-dimensional array type `S[]` to `System.Collections.Generic.IList<T>` and its base interfaces, provided that there is an explicit reference conversion from `S` to `T`.
- From `System.Collections.Generic.IList<S>` and its base interfaces to a single-dimensional array type `T[]`, provided that there is an explicit identity or reference conversion from `S` to `T`.
- From `System.Delegate` and the interfaces it implements to any *delegate_type*.
- From a reference type to a reference type `T` if it has an explicit reference conversion to a reference type `T0` and `T0` has an identity conversion `T`.
- From a reference type to an interface or delegate type `T` if it has an explicit reference conversion to an interface or delegate type `T0` and either `T0` is variance-convertible to `T` or `T` is variance-convertible to `T0` (Variance conversion).
- From `D<S1...Sn>` to `D<T1...Tn>` where `D<X1...Xn>` is a generic delegate type, `D<S1...Sn>` is not compatible with or identical to `D<T1...Tn>`, and for each type parameter `Xi` of `D` the following holds:
  - If `Xi` is invariant, then `Si` is identical to `Ti`.
  - If `Xi` is covariant, then there is an implicit or explicit identity or reference conversion from `Si` to `Ti`.
  - If `Xi` is contravariant, then `Si` and `Ti` are either identical or both reference types.
- Explicit conversions involving type parameters that are known to be reference types. For more details on explicit conversions involving type parameters, see Explicit conversions involving type parameters.

The explicit reference conversions are those conversions between reference-types that require run-time checks to ensure they are correct.

For an explicit reference conversion to succeed at run-time, the value of the source operand must be `null`, or the actual type of the object referenced by the source operand must be a type that can be converted to the destination type by an implicit reference conversion (Implicit reference conversions) or boxing conversion (Boxing conversions). If an explicit reference conversion fails, a `System.InvalidCastException` is thrown.

Reference conversions, implicit or explicit, never change the referential identity of the object being converted. In other words, while a reference conversion may change the type of the reference, it never changes the type or value of the object being referred to.

Unboxing conversions

An unboxing conversion permits a reference type to be explicitly converted to a *value_type*. An unboxing conversion exists from the types `object`, `dynamic` and `System.ValueType` to any *non_nullable_value_type*, and

from any *interface_type* to any *non_nullable_value_type* that implements the *interface_type*. Furthermore type `System.Enum` can be unboxed to any *enum_type*.

An unboxing conversion exists from a reference type to a *nullable_type* if an unboxing conversion exists from the reference type to the underlying *non_nullable_value_type* of the *nullable_type*.

A value type `S` has an unboxing conversion from an interface type `I` if it has an unboxing conversion from an interface type `I0` and `I0` has an identity conversion to `I`.

A value type `S` has an unboxing conversion from an interface type `I` if it has an unboxing conversion from an interface or delegate type `I0` and either `I0` is variance-convertible to `I` or `I` is variance-convertible to `I0` (Variance conversion).

An unboxing operation consists of first checking that the object instance is a boxed value of the given *value_type*, and then copying the value out of the instance. Unboxing a null reference to a *nullable_type* produces the null value of the *nullable_type*. A struct can be unboxed from the type `System.ValueType`, since that is a base class for all structs (Inheritance).

Unboxing conversions are described further in Unboxing conversions.

Explicit dynamic conversions

An explicit dynamic conversion exists from an expression of type `dynamic` to any type `T`. The conversion is dynamically bound (Dynamic binding), which means that an explicit conversion will be sought at run-time from the run-time type of the expression to `T`. If no conversion is found, a run-time exception is thrown.

If dynamic binding of the conversion is not desired, the expression can be first converted to `object`, and then to the desired type.

Assume the following class is defined:

```
class C
{
    int i;

    public C(int i) { this.i = i; }

    public static explicit operator C(string s)
    {
        return new C(int.Parse(s));
    }
}
```

The following example illustrates explicit dynamic conversions:

```
object o  = "1";
dynamic d = "2";

var c1 = (C)o; // Compiles, but explicit reference conversion fails
var c2 = (C)d; // Compiles and user defined conversion succeeds
```

The best conversion of `o` to `C` is found at compile-time to be an explicit reference conversion. This fails at run-time, because `"1"` is not in fact a `C`. The conversion of `d` to `C` however, as an explicit dynamic conversion, is suspended to run-time, where a user defined conversion from the run-time type of `d` -- `string` -- to `C` is found, and succeeds.

Explicit conversions involving type parameters

The following explicit conversions exist for a given type parameter `T`:

- From the effective base class `C` of `T` to `T` and from any base class of `C` to `T`. At run-time, if `T` is a value type, the conversion is executed as an unboxing conversion. Otherwise, the conversion is executed as an explicit reference conversion or identity conversion.
- From any interface type to `T`. At run-time, if `T` is a value type, the conversion is executed as an unboxing conversion. Otherwise, the conversion is executed as an explicit reference conversion or identity conversion.
- From `T` to any *interface_type* `I` provided there is not already an implicit conversion from `T` to `I`. At run-time, if `T` is a value type, the conversion is executed as a boxing conversion followed by an explicit reference conversion. Otherwise, the conversion is executed as an explicit reference conversion or identity conversion.
- From a type parameter `U` to `T`, provided `T` depends on `U` (Type parameter constraints). At run-time, if `U` is a value type, then `T` and `U` are necessarily the same type and no conversion is performed. Otherwise, if `T` is a value type, the conversion is executed as an unboxing conversion. Otherwise, the conversion is executed as an explicit reference conversion or identity conversion.

If `T` is known to be a reference type, the conversions above are all classified as explicit reference conversions (Explicit reference conversions). If `T` is not known to be a reference type, the conversions above are classified as unboxing conversions (Unboxing conversions).

The above rules do not permit a direct explicit conversion from an unconstrained type parameter to a non-interface type, which might be surprising. The reason for this rule is to prevent confusion and make the semantics of such conversions clear. For example, consider the following declaration:

```
class X<T>
{
    public static long F(T t) {
        return (long)t;               // Error
    }
}
```

If the direct explicit conversion of `t` to `int` were permitted, one might easily expect that `X<int>.F(7)` would return `7L`. However, it would not, because the standard numeric conversions are only considered when the types are known to be numeric at binding-time. In order to make the semantics clear, the above example must instead be written:

```
class X<T>
{
    public static long F(T t) {
        return (long)(object)t;       // Ok, but will only work when T is long
    }
}
```

This code will now compile but executing `X<int>.F(7)` would then throw an exception at run-time, since a boxed `int` cannot be converted directly to a `long`.

User-defined explicit conversions

A user-defined explicit conversion consists of an optional standard explicit conversion, followed by execution of a user-defined implicit or explicit conversion operator, followed by another optional standard explicit conversion. The exact rules for evaluating user-defined explicit conversions are described in Processing of user-defined explicit conversions.

Standard conversions

The standard conversions are those pre-defined conversions that can occur as part of a user-defined conversion.

Standard implicit conversions

The following implicit conversions are classified as standard implicit conversions:

- Identity conversions (Identity conversion)
- Implicit numeric conversions (Implicit numeric conversions)
- Implicit nullable conversions (Implicit nullable conversions)
- Implicit reference conversions (Implicit reference conversions)
- Boxing conversions (Boxing conversions)
- Implicit constant expression conversions (Implicit constant expression conversions)
- Implicit conversions involving type parameters (Implicit conversions involving type parameters)

The standard implicit conversions specifically exclude user-defined implicit conversions.

Standard explicit conversions

The standard explicit conversions are all standard implicit conversions plus the subset of the explicit conversions for which an opposite standard implicit conversion exists. In other words, if a standard implicit conversion exists from a type `A` to a type `B`, then a standard explicit conversion exists from type `A` to type `B` and from type `B` to type `A`.

User-defined conversions

C# allows the pre-defined implicit and explicit conversions to be augmented by ***user-defined conversions***. User-defined conversions are introduced by declaring conversion operators (Conversion operators) in class and struct types.

Permitted user-defined conversions

C# permits only certain user-defined conversions to be declared. In particular, it is not possible to redefine an already existing implicit or explicit conversion.

For a given source type `S` and target type `T`, if `S` or `T` are nullable types, let `S0` and `T0` refer to their underlying types, otherwise `S0` and `T0` are equal to `S` and `T` respectively. A class or struct is permitted to declare a conversion from a source type `S` to a target type `T` only if all of the following are true:

- `S0` and `T0` are different types.
- Either `S0` or `T0` is the class or struct type in which the operator declaration takes place.
- Neither `S0` nor `T0` is an *interface_type*.
- Excluding user-defined conversions, a conversion does not exist from `S` to `T` or from `T` to `S`.

The restrictions that apply to user-defined conversions are discussed further in Conversion operators.

Lifted conversion operators

Given a user-defined conversion operator that converts from a non-nullable value type `S` to a non-nullable value type `T`, a ***lifted conversion operator*** exists that converts from `S?` to `T?`. This lifted conversion operator performs an unwrapping from `S?` to `S` followed by the user-defined conversion from `S` to `T` followed by a wrapping from `T` to `T?`, except that a null valued `S?` converts directly to a null valued `T?`.

A lifted conversion operator has the same implicit or explicit classification as its underlying user-defined conversion operator. The term "user-defined conversion" applies to the use of both user-defined and lifted conversion operators.

Evaluation of user-defined conversions

A user-defined conversion converts a value from its type, called the ***source type***, to another type, called the ***target type***. Evaluation of a user-defined conversion centers on finding the ***most specific*** user-defined conversion operator for the particular source and target types. This determination is broken into several steps:

- Finding the set of classes and structs from which user-defined conversion operators will be considered. This set consists of the source type and its base classes and the target type and its base classes (with the implicit assumptions that only classes and structs can declare user-defined operators, and that non-class types have no base classes). For the purposes of this step, if either the source or target type is a *nullable_type*, their underlying type is used instead.
- From that set of types, determining which user-defined and lifted conversion operators are applicable. For a conversion operator to be applicable, it must be possible to perform a standard conversion (Standard conversions) from the source type to the operand type of the operator, and it must be possible to perform a standard conversion from the result type of the operator to the target type.
- From the set of applicable user-defined operators, determining which operator is unambiguously the most specific. In general terms, the most specific operator is the operator whose operand type is "closest" to the source type and whose result type is "closest" to the target type. User-defined conversion operators are preferred over lifted conversion operators. The exact rules for establishing the most specific user-defined conversion operator are defined in the following sections.

Once a most specific user-defined conversion operator has been identified, the actual execution of the user-defined conversion involves up to three steps:

- First, if required, performing a standard conversion from the source type to the operand type of the user-defined or lifted conversion operator.
- Next, invoking the user-defined or lifted conversion operator to perform the conversion.
- Finally, if required, performing a standard conversion from the result type of the user-defined or lifted conversion operator to the target type.

Evaluation of a user-defined conversion never involves more than one user-defined or lifted conversion operator. In other words, a conversion from type `S` to type `T` will never first execute a user-defined conversion from `S` to `X` and then execute a user-defined conversion from `X` to `T`.

Exact definitions of evaluation of user-defined implicit or explicit conversions are given in the following sections. The definitions make use of the following terms:

- If a standard implicit conversion (Standard implicit conversions) exists from a type `A` to a type `B`, and if neither `A` nor `B` are *interface_type*s, then `A` is said to be ***encompassed by*** `B`, and `B` is said to ***encompass*** `A`.
- The ***most encompassing type*** in a set of types is the one type that encompasses all other types in the set. If no single type encompasses all other types, then the set has no most encompassing type. In more intuitive terms, the most encompassing type is the "largest" type in the set—the one type to which each of the other types can be implicitly converted.
- The ***most encompassed type*** in a set of types is the one type that is encompassed by all other types in the set. If no single type is encompassed by all other types, then the set has no most encompassed type. In more intuitive terms, the most encompassed type is the "smallest" type in the set—the one type that can be implicitly converted to each of the other types.

Processing of user-defined implicit conversions

A user-defined implicit conversion from type `S` to type `T` is processed as follows:

- Determine the types `S0` and `T0`. If `S` or `T` are nullable types, `S0` and `T0` are their underlying types, otherwise `S0` and `T0` are equal to `S` and `T` respectively.
- Find the set of types, `D`, from which user-defined conversion operators will be considered. This set consists of `S0` (if `S0` is a class or struct), the base classes of `S0` (if `S0` is a class), and `T0` (if `T0` is a class or struct).
- Find the set of applicable user-defined and lifted conversion operators, `U`. This set consists of the user-defined and lifted implicit conversion operators declared by the classes or structs in `D` that convert from a type encompassing `S` to a type encompassed by `T`. If `U` is empty, the conversion is undefined and a

compile-time error occurs.

- Find the most specific source type, `SX`, of the operators in `U`:
  - If any of the operators in `U` convert from `S`, then `SX` is `S`.
  - Otherwise, `SX` is the most encompassed type in the combined set of source types of the operators in `U`. If exactly one most encompassed type cannot be found, then the conversion is ambiguous and a compile-time error occurs.
- Find the most specific target type, `TX`, of the operators in `U`:
  - If any of the operators in `U` convert to `T`, then `TX` is `T`.
  - Otherwise, `TX` is the most encompassing type in the combined set of target types of the operators in `U`. If exactly one most encompassing type cannot be found, then the conversion is ambiguous and a compile-time error occurs.
- Find the most specific conversion operator:
  - If `U` contains exactly one user-defined conversion operator that converts from `SX` to `TX`, then this is the most specific conversion operator.
  - Otherwise, if `U` contains exactly one lifted conversion operator that converts from `SX` to `TX`, then this is the most specific conversion operator.
  - Otherwise, the conversion is ambiguous and a compile-time error occurs.
- Finally, apply the conversion:
  - If `S` is not `SX`, then a standard implicit conversion from `S` to `SX` is performed.
  - The most specific conversion operator is invoked to convert from `SX` to `TX`.
  - If `TX` is not `T`, then a standard implicit conversion from `TX` to `T` is performed.

Processing of user-defined explicit conversions

A user-defined explicit conversion from type `S` to type `T` is processed as follows:

- Determine the types `S0` and `T0`. If `S` or `T` are nullable types, `S0` and `T0` are their underlying types, otherwise `S0` and `T0` are equal to `S` and `T` respectively.
- Find the set of types, `D`, from which user-defined conversion operators will be considered. This set consists of `S0` (if `S0` is a class or struct), the base classes of `S0` (if `S0` is a class), `T0` (if `T0` is a class or struct), and the base classes of `T0` (if `T0` is a class).
- Find the set of applicable user-defined and lifted conversion operators, `U`. This set consists of the user-defined and lifted implicit or explicit conversion operators declared by the classes or structs in `D` that convert from a type encompassing or encompassed by `S` to a type encompassing or encompassed by `T`. If `U` is empty, the conversion is undefined and a compile-time error occurs.
- Find the most specific source type, `SX`, of the operators in `U`:
  - If any of the operators in `U` convert from `S`, then `SX` is `S`.
  - Otherwise, if any of the operators in `U` convert from types that encompass `S`, then `SX` is the most encompassed type in the combined set of source types of those operators. If no most encompassed type can be found, then the conversion is ambiguous and a compile-time error occurs.
  - Otherwise, `SX` is the most encompassing type in the combined set of source types of the operators in `U`. If exactly one most encompassing type cannot be found, then the conversion is ambiguous and a compile-time error occurs.
- Find the most specific target type, `TX`, of the operators in `U`:
  - If any of the operators in `U` convert to `T`, then `TX` is `T`.
  - Otherwise, if any of the operators in `U` convert to types that are encompassed by `T`, then `TX` is the most encompassing type in the combined set of target types of those operators. If exactly one most encompassing type cannot be found, then the conversion is ambiguous and a compile-time error occurs.

    - Otherwise, `TX` is the most encompassed type in the combined set of target types of the operators in `U`. If no most encompassed type can be found, then the conversion is ambiguous and a compile-time error occurs.
- Find the most specific conversion operator:
    - If `U` contains exactly one user-defined conversion operator that converts from `SX` to `TX`, then this is the most specific conversion operator.
    - Otherwise, if `U` contains exactly one lifted conversion operator that converts from `SX` to `TX`, then this is the most specific conversion operator.
    - Otherwise, the conversion is ambiguous and a compile-time error occurs.
- Finally, apply the conversion:
    - If `S` is not `SX`, then a standard explicit conversion from `S` to `SX` is performed.
    - The most specific user-defined conversion operator is invoked to convert from `SX` to `TX`.
    - If `TX` is not `T`, then a standard explicit conversion from `TX` to `T` is performed.

Anonymous function conversions

An *anonymous_method_expression* or *lambda_expression* is classified as an anonymous function (Anonymous function expressions). The expression does not have a type but can be implicitly converted to a compatible delegate type or expression tree type. Specifically, an anonymous function `F` is compatible with a delegate type `D` provided:

- If `F` contains an *anonymous_function_signature*, then `D` and `F` have the same number of parameters.
- If `F` does not contain an *anonymous_function_signature*, then `D` may have zero or more parameters of any type, as long as no parameter of `D` has the `out` parameter modifier.
- If `F` has an explicitly typed parameter list, each parameter in `D` has the same type and modifiers as the corresponding parameter in `F`.
- If `F` has an implicitly typed parameter list, `D` has no `ref` or `out` parameters.
- If the body of `F` is an expression, and either `D` has a `void` return type or `F` is async and `D` has the return type `Task`, then when each parameter of `F` is given the type of the corresponding parameter in `D`, the body of `F` is a valid expression (wrt Expressions) that would be permitted as a *statement_expression* (Expression statements).
- If the body of `F` is a statement block, and either `D` has a `void` return type or `F` is async and `D` has the return type `Task`, then when each parameter of `F` is given the type of the corresponding parameter in `D`, the body of `F` is a valid statement block (wrt Blocks) in which no `return` statement specifies an expression.
- If the body of `F` is an expression, and *either* `F` is non-async and `D` has a non-void return type `T`, *or* `F` is async and `D` has a return type `Task<T>`, then when each parameter of `F` is given the type of the corresponding parameter in `D`, the body of `F` is a valid expression (wrt Expressions) that is implicitly convertible to `T`.
- If the body of `F` is a statement block, and *either* `F` is non-async and `D` has a non-void return type `T`, *or* `F` is async and `D` has a return type `Task<T>`, then when each parameter of `F` is given the type of the corresponding parameter in `D`, the body of `F` is a valid statement block (wrt Blocks) with a non-reachable end point in which each `return` statement specifies an expression that is implicitly convertible to `T`.

For the purpose of brevity, this section uses the short form for the task types `Task` and `Task<T>` (Async functions).

A lambda expression `F` is compatible with an expression tree type `Expression<D>` if `F` is compatible with the delegate type `D`. Note that this does not apply to anonymous methods, only lambda expressions.

Certain lambda expressions cannot be converted to expression tree types: Even though the conversion *exists*, it fails at compile-time. This is the case if the lambda expression:

- Has a *block* body
- Contains simple or compound assignment operators
- Contains a dynamically bound expression
- Is async

The examples that follow use a generic delegate type `Func<A,R>` which represents a function that takes an argument of type `A` and returns a value of type `R`:

```
delegate R Func<A,R>(A arg);
```

In the assignments

```
Func<int,int> f1 = x => x + 1;               // Ok

Func<int,double> f2 = x => x + 1;            // Ok

Func<double,int> f3 = x => x + 1;            // Error

Func<int, Task<int>> f4 = async x => x + 1;  // Ok
```

the parameter and return types of each anonymous function are determined from the type of the variable to which the anonymous function is assigned.

The first assignment successfully converts the anonymous function to the delegate type `Func<int,int>` because, when `x` is given type `int`, `x+1` is a valid expression that is implicitly convertible to type `int`.

Likewise, the second assignment successfully converts the anonymous function to the delegate type `Func<int,double>` because the result of `x+1` (of type `int`) is implicitly convertible to type `double`.

However, the third assignment is a compile-time error because, when `x` is given type `double`, the result of `x+1` (of type `double`) is not implicitly convertible to type `int`.

The fourth assignment successfully converts the anonymous async function to the delegate type `Func<int, Task<int>>` because the result of `x+1` (of type `int`) is implicitly convertible to the result type `int` of the task type `Task<int>`.

Anonymous functions may influence overload resolution, and participate in type inference. See Function members for further details.

Evaluation of anonymous function conversions to delegate types

Conversion of an anonymous function to a delegate type produces a delegate instance which references the anonymous function and the (possibly empty) set of captured outer variables that are active at the time of the evaluation. When the delegate is invoked, the body of the anonymous function is executed. The code in the body is executed using the set of captured outer variables referenced by the delegate.

The invocation list of a delegate produced from an anonymous function contains a single entry. The exact target object and target method of the delegate are unspecified. In particular, it is unspecified whether the target object of the delegate is `null`, the `this` value of the enclosing function member, or some other object.

Conversions of semantically identical anonymous functions with the same (possibly empty) set of captured outer variable instances to the same delegate types are permitted (but not required) to return the same delegate instance. The term semantically identical is used here to mean that execution of the anonymous functions will, in all cases, produce the same effects given the same arguments. This rule permits code such as the following to be optimized.

```
delegate double Function(double x);

class Test
{
    static double[] Apply(double[] a, Function f) {
        double[] result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }

    static void F(double[] a, double[] b) {
        a = Apply(a, (double x) => Math.Sin(x));
        b = Apply(b, (double y) => Math.Sin(y));
        ...
    }
}
```

Since the two anonymous function delegates have the same (empty) set of captured outer variables, and since the anonymous functions are semantically identical, the compiler is permitted to have the delegates refer to the same target method. Indeed, the compiler is permitted to return the very same delegate instance from both anonymous function expressions.

Evaluation of anonymous function conversions to expression tree types

Conversion of an anonymous function to an expression tree type produces an expression tree (Expression tree types). More precisely, evaluation of the anonymous function conversion leads to the construction of an object structure that represents the structure of the anonymous function itself. The precise structure of the expression tree, as well as the exact process for creating it, are implementation defined.

Implementation example

This section describes a possible implementation of anonymous function conversions in terms of other C# constructs. The implementation described here is based on the same principles used by the Microsoft C# compiler, but it is by no means a mandated implementation, nor is it the only one possible. It only briefly mentions conversions to expression trees, as their exact semantics are outside the scope of this specification.

The remainder of this section gives several examples of code that contains anonymous functions with different characteristics. For each example, a corresponding translation to code that uses only other C# constructs is provided. In the examples, the identifier `D` is assumed by represent the following delegate type:

```
public delegate void D();
```

The simplest form of an anonymous function is one that captures no outer variables:

```
class Test
{
    static void F() {
        D d = () => { Console.WriteLine("test"); };
    }
}
```

This can be translated to a delegate instantiation that references a compiler generated static method in which the code of the anonymous function is placed:

```
class Test
{
    static void F() {
        D d = new D(__Method1);
    }

    static void __Method1() {
        Console.WriteLine("test");
    }
}
```

In the following example, the anonymous function references instance members of `this`:

```
class Test
{
    int x;

    void F() {
        D d = () => { Console.WriteLine(x); };
    }
}
```

This can be translated to a compiler generated instance method containing the code of the anonymous function:

```
class Test
{
    int x;

    void F() {
        D d = new D(__Method1);
    }

    void __Method1() {
        Console.WriteLine(x);
    }
}
```

In this example, the anonymous function captures a local variable:

```
class Test
{
    void F() {
        int y = 123;
        D d = () => { Console.WriteLine(y); };
    }
}
```

The lifetime of the local variable must now be extended to at least the lifetime of the anonymous function delegate. This can be achieved by "hoisting" the local variable into a field of a compiler generated class. Instantiation of the local variable (Instantiation of local variables) then corresponds to creating an instance of the compiler generated class, and accessing the local variable corresponds to accessing a field in the instance of the compiler generated class. Furthermore, the anonymous function becomes an instance method of the compiler generated class:

```
class Test
{
    void F() {
        __Locals1 __locals1 = new __Locals1();
        __locals1.y = 123;
        D d = new D(__locals1.__Method1);
    }

    class __Locals1
    {
        public int y;

        public void __Method1() {
            Console.WriteLine(y);
        }
    }
}
```

Finally, the following anonymous function captures `this` as well as two local variables with different lifetimes:

```
class Test
{
    int x;

    void F() {
        int y = 123;
        for (int i = 0; i < 10; i++) {
            int z = i * 2;
            D d = () => { Console.WriteLine(x + y + z); };
        }
    }
}
```

Here, a compiler generated class is created for each statement block in which locals are captured such that the locals in the different blocks can have independent lifetimes. An instance of `__Locals2`, the compiler generated class for the inner statement block, contains the local variable `z` and a field that references an instance of `__Locals1`. An instance of `__Locals1`, the compiler generated class for the outer statement block, contains the local variable `y` and a field that references `this` of the enclosing function member. With these data structures it is possible to reach all captured outer variables through an instance of `__Local2`, and the code of the anonymous function can thus be implemented as an instance method of that class.

```
class Test
{
    void F() {
        __Locals1 __locals1 = new __Locals1();
        __locals1.__this = this;
        __locals1.y = 123;
        for (int i = 0; i < 10; i++) {
            __Locals2 __locals2 = new __Locals2();
            __locals2.__locals1 = __locals1;
            __locals2.z = i * 2;
            D d = new D(__locals2.__Method1);
        }
    }

    class __Locals1
    {
        public Test __this;
        public int y;
    }

    class __Locals2
    {
        public __Locals1 __locals1;
        public int z;

        public void __Method1() {
            Console.WriteLine(__locals1.__this.x + __locals1.y + z);
        }
    }
}
```

The same technique applied here to capture local variables can also be used when converting anonymous functions to expression trees: References to the compiler generated objects can be stored in the expression tree, and access to the local variables can be represented as field accesses on these objects. The advantage of this approach is that it allows the "lifted" local variables to be shared between delegates and expression trees.

Method group conversions

An implicit conversion (Implicit conversions) exists from a method group (Expression classifications) to a compatible delegate type. Given a delegate type `D` and an expression `E` that is classified as a method group, an implicit conversion exists from `E` to `D` if `E` contains at least one method that is applicable in its normal form (Applicable function member) to an argument list constructed by use of the parameter types and modifiers of `D`, as described in the following.

The compile-time application of a conversion from a method group `E` to a delegate type `D` is described in the following. Note that the existence of an implicit conversion from `E` to `D` does not guarantee that the compile-time application of the conversion will succeed without error.

- A single method `M` is selected corresponding to a method invocation (Method invocations) of the form `E(A)`, with the following modifications:
  - The argument list `A` is a list of expressions, each classified as a variable and with the type and modifier (`ref` or `out`) of the corresponding parameter in the *formal_parameter_list* of `D`.
  - The candidate methods considered are only those methods that are applicable in their normal form (Applicable function member), not those applicable only in their expanded form.
- If the algorithm of Method invocations produces an error, then a compile-time error occurs. Otherwise the algorithm produces a single best method `M` having the same number of parameters as `D` and the conversion is considered to exist.
- The selected method `M` must be compatible (Delegate compatibility) with the delegate type `D`, or

otherwise, a compile-time error occurs.

- If the selected method `M` is an instance method, the instance expression associated with `E` determines the target object of the delegate.
- If the selected method M is an extension method which is denoted by means of a member access on an instance expression, that instance expression determines the target object of the delegate.
- The result of the conversion is a value of type `D`, namely a newly created delegate that refers to the selected method and target object.
- Note that this process can lead to the creation of a delegate to an extension method, if the algorithm of Method invocations fails to find an instance method but succeeds in processing the invocation of `E(A)` as an extension method invocation (Extension method invocations). A delegate thus created captures the extension method as well as its first argument.

The following example demonstrates method group conversions:

```
delegate string D1(object o);

delegate object D2(string s);

delegate object D3();

delegate string D4(object o, params object[] a);

delegate string D5(int i);

class Test
{
    static string F(object o) {...}

    static void G() {
        D1 d1 = F;            // Ok
        D2 d2 = F;            // Ok
        D3 d3 = F;            // Error -- not applicable
        D4 d4 = F;            // Error -- not applicable in normal form
        D5 d5 = F;            // Error -- applicable but not compatible

    }
}
```

The assignment to `d1` implicitly converts the method group `F` to a value of type `D1`.

The assignment to `d2` shows how it is possible to create a delegate to a method that has less derived (contravariant) parameter types and a more derived (covariant) return type.

The assignment to `d3` shows how no conversion exists if the method is not applicable.

The assignment to `d4` shows how the method must be applicable in its normal form.

The assignment to `d5` shows how parameter and return types of the delegate and method are allowed to differ only for reference types.

As with all other implicit and explicit conversions, the cast operator can be used to explicitly perform a method group conversion. Thus, the example

```
object obj = new EventHandler(myDialog.OkClick);
```

could instead be written

```
object obj = (EventHandler)myDialog.OkClick;
```

Method groups may influence overload resolution, and participate in type inference. See Function members for further details.

The run-time evaluation of a method group conversion proceeds as follows:

- If the method selected at compile-time is an instance method, or it is an extension method which is accessed as an instance method, the target object of the delegate is determined from the instance expression associated with `E`:
  - The instance expression is evaluated. If this evaluation causes an exception, no further steps are executed.
  - If the instance expression is of a *reference_type*, the value computed by the instance expression becomes the target object. If the selected method is an instance method and the target object is `null`, a `System.NullReferenceException` is thrown and no further steps are executed.
  - If the instance expression is of a *value_type*, a boxing operation (Boxing conversions) is performed to convert the value to an object, and this object becomes the target object.
- Otherwise the selected method is part of a static method call, and the target object of the delegate is `null`.
- A new instance of the delegate type `D` is allocated. If there is not enough memory available to allocate the new instance, a `System.OutOfMemoryException` is thrown and no further steps are executed.
- The new delegate instance is initialized with a reference to the method that was determined at compile-time and a reference to the target object computed above.

Expressions

18.12.2021 • 227 minutes to read

An expression is a sequence of operators and operands. This chapter defines the syntax, order of evaluation of operands and operators, and meaning of expressions.

Expression classifications

An expression is classified as one of the following:

- A value. Every value has an associated type.
- A variable. Every variable has an associated type, namely the declared type of the variable.
- A namespace. An expression with this classification can only appear as the left hand side of a *member_access* (Member access). In any other context, an expression classified as a namespace causes a compile-time error.
- A type. An expression with this classification can only appear as the left hand side of a *member_access* (Member access), or as an operand for the `as` operator (The as operator), the `is` operator (The is operator), or the `typeof` operator (The typeof operator). In any other context, an expression classified as a type causes a compile-time error.
- A method group, which is a set of overloaded methods resulting from a member lookup (Member lookup). A method group may have an associated instance expression and an associated type argument list. When an instance method is invoked, the result of evaluating the instance expression becomes the instance represented by `this` (This access). A method group is permitted in an *invocation_expression* (Invocation expressions) , a *delegate_creation_expression* (Delegate creation expressions) and as the left hand side of an is operator, and can be implicitly converted to a compatible delegate type (Method group conversions). In any other context, an expression classified as a method group causes a compile-time error.
- A null literal. An expression with this classification can be implicitly converted to a reference type or nullable type.
- An anonymous function. An expression with this classification can be implicitly converted to a compatible delegate type or expression tree type.
- A property access. Every property access has an associated type, namely the type of the property. Furthermore, a property access may have an associated instance expression. When an accessor (the `get` or `set` block) of an instance property access is invoked, the result of evaluating the instance expression becomes the instance represented by `this` (This access).
- An event access. Every event access has an associated type, namely the type of the event. Furthermore, an event access may have an associated instance expression. An event access may appear as the left hand operand of the `+=` and `-=` operators (Event assignment). In any other context, an expression classified as an event access causes a compile-time error.
- An indexer access. Every indexer access has an associated type, namely the element type of the indexer. Furthermore, an indexer access has an associated instance expression and an associated argument list. When an accessor (the `get` or `set` block) of an indexer access is invoked, the result of evaluating the instance expression becomes the instance represented by `this` (This access), and the result of evaluating the argument list becomes the parameter list of the invocation.
- Nothing. This occurs when the expression is an invocation of a method with a return type of `void`. An expression classified as nothing is only valid in the context of a *statement_expression* (Expression statements).

The final result of an expression is never a namespace, type, method group, or event access. Rather, as noted above, these categories of expressions are intermediate constructs that are only permitted in certain contexts.

A property access or indexer access is always reclassified as a value by performing an invocation of the *get accessor* or the *set accessor*. The particular accessor is determined by the context of the property or indexer access: If the access is the target of an assignment, the *set accessor* is invoked to assign a new value (Simple assignment). Otherwise, the *get accessor* is invoked to obtain the current value (Values of expressions).

Values of expressions

Most of the constructs that involve an expression ultimately require the expression to denote a ***value***. In such cases, if the actual expression denotes a namespace, a type, a method group, or nothing, a compile-time error occurs. However, if the expression denotes a property access, an indexer access, or a variable, the value of the property, indexer, or variable is implicitly substituted:

- The value of a variable is simply the value currently stored in the storage location identified by the variable. A variable must be considered definitely assigned (Definite assignment) before its value can be obtained, or otherwise a compile-time error occurs.
- The value of a property access expression is obtained by invoking the *get accessor* of the property. If the property has no *get accessor*, a compile-time error occurs. Otherwise, a function member invocation (Compile-time checking of dynamic overload resolution) is performed, and the result of the invocation becomes the value of the property access expression.
- The value of an indexer access expression is obtained by invoking the *get accessor* of the indexer. If the indexer has no *get accessor*, a compile-time error occurs. Otherwise, a function member invocation (Compile-time checking of dynamic overload resolution) is performed with the argument list associated with the indexer access expression, and the result of the invocation becomes the value of the indexer access expression.

Static and Dynamic Binding

The process of determining the meaning of an operation based on the type or value of constituent expressions (arguments, operands, receivers) is often referred to as ***binding***. For instance the meaning of a method call is determined based on the type of the receiver and arguments. The meaning of an operator is determined based on the type of its operands.

In C# the meaning of an operation is usually determined at compile-time, based on the compile-time type of its constituent expressions. Likewise, if an expression contains an error, the error is detected and reported by the compiler. This approach is known as ***static binding***.

However, if an expression is a dynamic expression (i.e. has the type `dynamic`) this indicates that any binding that it participates in should be based on its run-time type (i.e. the actual type of the object it denotes at run-time) rather than the type it has at compile-time. The binding of such an operation is therefore deferred until the time where the operation is to be executed during the running of the program. This is referred to as ***dynamic binding***.

When an operation is dynamically bound, little or no checking is performed by the compiler. Instead if the run-time binding fails, errors are reported as exceptions at run-time.

The following operations in C# are subject to binding:

- Member access: `e.M`
- Method invocation: `e.M(e1, ..., eN)`
- Delegate invocation: `e(e1, ..., eN)`
- Element access: `e[e1, ..., eN]`
- Object creation: `new C(e1, ..., eN)`
- Overloaded unary operators: `+`, `-`, `!`, `~`, `++`, `--`, `true`, `false`
- Overloaded binary operators: `+`, `-`, `*`, `/`, `%`, `&`, `&&`, `|`, `||`, `??`, `^`, `<<`, `>>`, `==`, `!=`, `>`, `<`, `>=`, `<=`

- Assignment operators: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`
- Implicit and explicit conversions

When no dynamic expressions are involved, C# defaults to static binding, which means that the compile-time types of constituent expressions are used in the selection process. However, when one of the constituent expressions in the operations listed above is a dynamic expression, the operation is instead dynamically bound.

Binding-time

Static binding takes place at compile-time, whereas dynamic binding takes place at run-time. In the following sections, the term ***binding-time*** refers to either compile-time or run-time, depending on when the binding takes place.

The following example illustrates the notions of static and dynamic binding and of binding-time:

```
object  o = 5;
dynamic d = 5;

Console.WriteLine(5);  // static  binding to Console.WriteLine(int)
Console.WriteLine(o);  // static  binding to Console.WriteLine(object)
Console.WriteLine(d);  // dynamic binding to Console.WriteLine(int)
```

The first two calls are statically bound: the overload of `Console.WriteLine` is picked based on the compile-time type of their argument. Thus, the binding-time is compile-time.

The third call is dynamically bound: the overload of `Console.WriteLine` is picked based on the run-time type of its argument. This happens because the argument is a dynamic expression -- its compile-time type is `dynamic`. Thus, the binding-time for the third call is run-time.

Dynamic binding

The purpose of dynamic binding is to allow C# programs to interact with ***dynamic objects***, i.e. objects that do not follow the normal rules of the C# type system. Dynamic objects may be objects from other programming languages with different types systems, or they may be objects that are programmatically setup to implement their own binding semantics for different operations.

The mechanism by which a dynamic object implements its own semantics is implementation defined. A given interface -- again implementation defined -- is implemented by dynamic objects to signal to the C# run-time that they have special semantics. Thus, whenever operations on a dynamic object are dynamically bound, their own binding semantics, rather than those of C# as specified in this document, take over.

While the purpose of dynamic binding is to allow interoperation with dynamic objects, C# allows dynamic binding on all objects, whether they are dynamic or not. This allows for a smoother integration of dynamic objects, as the results of operations on them may not themselves be dynamic objects, but are still of a type unknown to the programmer at compile-time. Also dynamic binding can help eliminate error-prone reflection-based code even when no objects involved are dynamic objects.

The following sections describe for each construct in the language exactly when dynamic binding is applied, what compile time checking -- if any -- is applied, and what the compile-time result and expression classification is.

Types of constituent expressions

When an operation is statically bound, the type of a constituent expression (e.g. a receiver, an argument, an index or an operand) is always considered to be the compile-time type of that expression.

When an operation is dynamically bound, the type of a constituent expression is determined in different ways depending on the compile-time type of the constituent expression:

- A constituent expression of compile-time type `dynamic` is considered to have the type of the actual value that the expression evaluates to at runtime
- A constituent expression whose compile-time type is a type parameter is considered to have the type which the type parameter is bound to at runtime
- Otherwise the constituent expression is considered to have its compile-time type.

Operators

Expressions are constructed from ***operands*** and ***operators***. The operators of an expression indicate which operations to apply to the operands. Examples of operators include `+`, `-`, `*`, `/`, and `new`. Examples of operands include literals, fields, local variables, and expressions.

There are three kinds of operators:

- Unary operators. The unary operators take one operand and use either prefix notation (such as `--x`) or postfix notation (such as `x++`).
- Binary operators. The binary operators take two operands and all use infix notation (such as `x + y`).
- Ternary operator. Only one ternary operator, `?:`, exists; it takes three operands and uses infix notation (`c ? x : y`).

The order of evaluation of operators in an expression is determined by the ***precedence*** and ***associativity*** of the operators (Operator precedence and associativity).

Operands in an expression are evaluated from left to right. For example, in `F(i) + G(i++) * H(i)`, method `F` is called using the old value of `i`, then method `G` is called with the old value of `i`, and, finally, method `H` is called with the new value of `i`. This is separate from and unrelated to operator precedence.

Certain operators can be ***overloaded***. Operator overloading permits user-defined operator implementations to be specified for operations where one or both of the operands are of a user-defined class or struct type (Operator overloading).

Operator precedence and associativity

When an expression contains multiple operators, the ***precedence*** of the operators controls the order in which the individual operators are evaluated. For example, the expression `x + y * z` is evaluated as `x + (y * z)` because the `*` operator has higher precedence than the binary `+` operator. The precedence of an operator is established by the definition of its associated grammar production. For example, an *additive_expression* consists of a sequence of *multiplicative_expression*s separated by `+` or `-` operators, thus giving the `+` and `-` operators lower precedence than the `*`, `/`, and `%` operators.

The following table summarizes all operators in order of precedence from highest to lowest:

SECTION	CATEGORY	OPERATORS
Primary expressions	Primary	`x.y` `f(x)` `a[x]` `x++` `x--` `new` `typeof` `default` `checked` `unchecked` `delegate`
Unary operators	Unary	`+` `-` `!` `~` `++x` `--x` `(T)x`
Arithmetic operators	Multiplicative	`*` `/` `%`
Arithmetic operators	Additive	`+` `-`
Shift operators	Shift	`<<` `>>`

SECTION	CATEGORY	OPERATORS
Relational and type-testing operators	Relational and type testing	`<` `>` `<=` `>=` `is` `as`
Relational and type-testing operators	Equality	`==` `!=`
Logical operators	Logical AND	`&`
Logical operators	Logical XOR	`^`
Logical operators	Logical OR	`|`
Conditional logical operators	Conditional AND	`&&`
Conditional logical operators	Conditional OR	`||`
The null coalescing operator	Null coalescing	`??`
Conditional operator	Conditional	`?:`
Assignment operators, Anonymous function expressions	Assignment and lambda expression	`=` `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=` `=>`

When an operand occurs between two operators with the same precedence, the associativity of the operators controls the order in which the operations are performed:

- Except for the assignment operators and the null coalescing operator, all binary operators are ***left-associative***, meaning that operations are performed from left to right. For example, `x + y + z` is evaluated as `(x + y) + z`.
- The assignment operators, the null coalescing operator and the conditional operator (`?:`) are ***right-associative***, meaning that operations are performed from right to left. For example, `x = y = z` is evaluated as `x = (y = z)`.

Precedence and associativity can be controlled using parentheses. For example, `x + y * z` first multiplies `y` by `z` and then adds the result to `x`, but `(x + y) * z` first adds `x` and `y` and then multiplies the result by `z`.

Operator overloading

All unary and binary operators have predefined implementations that are automatically available in any expression. In addition to the predefined implementations, user-defined implementations can be introduced by including `operator` declarations in classes and structs (Operators). User-defined operator implementations always take precedence over predefined operator implementations: Only when no applicable user-defined operator implementations exist will the predefined operator implementations be considered, as described in Unary operator overload resolution and Binary operator overload resolution.

The *overloadable unary operators* are:

```
+   -   !   ~   ++   --   true   false
```

Although `true` and `false` are not used explicitly in expressions (and therefore are not included in the precedence table in Operator precedence and associativity), they are considered operators because they are

invoked in several expression contexts: boolean expressions (Boolean expressions) and expressions involving the conditional (Conditional operator), and conditional logical operators (Conditional logical operators).

The *overloadable binary operators* are:

```
+   -   *   /   %   &   |   ^   <<   >>   ==   !=   >   <   >=   <=
```

Only the operators listed above can be overloaded. In particular, it is not possible to overload member access, method invocation, or the `=`, `&&`, `||`, `??`, `?:`, `=>`, `checked`, `unchecked`, `new`, `typeof`, `default`, `as`, and `is` operators.

When a binary operator is overloaded, the corresponding assignment operator, if any, is also implicitly overloaded. For example, an overload of operator `*` is also an overload of operator `*=`. This is described further in Compound assignment. Note that the assignment operator itself (`=`) cannot be overloaded. An assignment always performs a simple bit-wise copy of a value into a variable.

Cast operations, such as `(T)x`, are overloaded by providing user-defined conversions (User-defined conversions).

Element access, such as `a[x]`, is not considered an overloadable operator. Instead, user-defined indexing is supported through indexers (Indexers).

In expressions, operators are referenced using operator notation, and in declarations, operators are referenced using functional notation. The following table shows the relationship between operator and functional notations for unary and binary operators. In the first entry, *op* denotes any overloadable unary prefix operator. In the second entry, *op* denotes the unary postfix `++` and `--` operators. In the third entry, *op* denotes any overloadable binary operator.

OPERATOR NOTATION	FUNCTIONAL NOTATION
`op x`	`operator op(x)`
`x op`	`operator op(x)`
`x op y`	`operator op(x,y)`

User-defined operator declarations always require at least one of the parameters to be of the class or struct type that contains the operator declaration. Thus, it is not possible for a user-defined operator to have the same signature as a predefined operator.

User-defined operator declarations cannot modify the syntax, precedence, or associativity of an operator. For example, the `/` operator is always a binary operator, always has the precedence level specified in Operator precedence and associativity, and is always left-associative.

While it is possible for a user-defined operator to perform any computation it pleases, implementations that produce results other than those that are intuitively expected are strongly discouraged. For example, an implementation of `operator ==` should compare the two operands for equality and return an appropriate `bool` result.

The descriptions of individual operators in Primary expressions through Conditional logical operators specify the predefined implementations of the operators and any additional rules that apply to each operator. The descriptions make use of the terms *unary operator overload resolution*, *binary operator overload resolution*, and *numeric promotion*, definitions of which are found in the following sections.

Unary operator overload resolution

An operation of the form `op x` or `x op`, where `op` is an overloadable unary operator, and `x` is an expression of type `X`, is processed as follows:

- The set of candidate user-defined operators provided by `X` for the operation `operator op(x)` is determined using the rules of Candidate user-defined operators.
- If the set of candidate user-defined operators is not empty, then this becomes the set of candidate operators for the operation. Otherwise, the predefined unary `operator op` implementations, including their lifted forms, become the set of candidate operators for the operation. The predefined implementations of a given operator are specified in the description of the operator (Primary expressions and Unary operators).
- The overload resolution rules of Overload resolution are applied to the set of candidate operators to select the best operator with respect to the argument list `(x)`, and this operator becomes the result of the overload resolution process. If overload resolution fails to select a single best operator, a binding-time error occurs.

Binary operator overload resolution

An operation of the form `x op y`, where `op` is an overloadable binary operator, `x` is an expression of type `X`, and `y` is an expression of type `Y`, is processed as follows:

- The set of candidate user-defined operators provided by `X` and `Y` for the operation `operator op(x,y)` is determined. The set consists of the union of the candidate operators provided by `X` and the candidate operators provided by `Y`, each determined using the rules of Candidate user-defined operators. If `X` and `Y` are the same type, or if `X` and `Y` are derived from a common base type, then shared candidate operators only occur in the combined set once.
- If the set of candidate user-defined operators is not empty, then this becomes the set of candidate operators for the operation. Otherwise, the predefined binary `operator op` implementations, including their lifted forms, become the set of candidate operators for the operation. The predefined implementations of a given operator are specified in the description of the operator (Arithmetic operators through Conditional logical operators). For predefined enum and delegate operators, the only operators considered are those defined by an enum or delegate type that is the binding-time type of one of the operands.
- The overload resolution rules of Overload resolution are applied to the set of candidate operators to select the best operator with respect to the argument list `(x,y)`, and this operator becomes the result of the overload resolution process. If overload resolution fails to select a single best operator, a binding-time error occurs.

Candidate user-defined operators

Given a type `T` and an operation `operator op(A)`, where `op` is an overloadable operator and `A` is an argument list, the set of candidate user-defined operators provided by `T` for `operator op(A)` is determined as follows:

- Determine the type `T0`. If `T` is a nullable type, `T0` is its underlying type, otherwise `T0` is equal to `T`.
- For all `operator op` declarations in `T0` and all lifted forms of such operators, if at least one operator is applicable (Applicable function member) with respect to the argument list `A`, then the set of candidate operators consists of all such applicable operators in `T0`.
- Otherwise, if `T0` is `object`, the set of candidate operators is empty.
- Otherwise, the set of candidate operators provided by `T0` is the set of candidate operators provided by the direct base class of `T0`, or the effective base class of `T0` if `T0` is a type parameter.

Numeric promotions

Numeric promotion consists of automatically performing certain implicit conversions of the operands of the predefined unary and binary numeric operators. Numeric promotion is not a distinct mechanism, but rather an effect of applying overload resolution to the predefined operators. Numeric promotion specifically does not affect evaluation of user-defined operators, although user-defined operators can be implemented to exhibit

similar effects.

As an example of numeric promotion, consider the predefined implementations of the binary `*` operator:

```
int operator *(int x, int y);
uint operator *(uint x, uint y);
long operator *(long x, long y);
ulong operator *(ulong x, ulong y);
float operator *(float x, float y);
double operator *(double x, double y);
decimal operator *(decimal x, decimal y);
```

When overload resolution rules (Overload resolution) are applied to this set of operators, the effect is to select the first of the operators for which implicit conversions exist from the operand types. For example, for the operation `b * s`, where `b` is a `byte` and `s` is a `short`, overload resolution selects `operator *(int,int)` as the best operator. Thus, the effect is that `b` and `s` are converted to `int`, and the type of the result is `int`. Likewise, for the operation `i * d`, where `i` is an `int` and `d` is a `double`, overload resolution selects `operator *(double,double)` as the best operator.

Unary numeric promotions

Unary numeric promotion occurs for the operands of the predefined `+`, `-`, and `~` unary operators. Unary numeric promotion simply consists of converting operands of type `sbyte`, `byte`, `short`, `ushort`, or `char` to type `int`. Additionally, for the unary `-` operator, unary numeric promotion converts operands of type `uint` to type `long`.

Binary numeric promotions

Binary numeric promotion occurs for the operands of the predefined `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `==`, `!=`, `>`, `<`, `>=`, and `<=` binary operators. Binary numeric promotion implicitly converts both operands to a common type which, in case of the non-relational operators, also becomes the result type of the operation. Binary numeric promotion consists of applying the following rules, in the order they appear here:

- If either operand is of type `decimal`, the other operand is converted to type `decimal`, or a binding-time error occurs if the other operand is of type `float` or `double`.
- Otherwise, if either operand is of type `double`, the other operand is converted to type `double`.
- Otherwise, if either operand is of type `float`, the other operand is converted to type `float`.
- Otherwise, if either operand is of type `ulong`, the other operand is converted to type `ulong`, or a binding-time error occurs if the other operand is of type `sbyte`, `short`, `int`, or `long`.
- Otherwise, if either operand is of type `long`, the other operand is converted to type `long`.
- Otherwise, if either operand is of type `uint` and the other operand is of type `sbyte`, `short`, or `int`, both operands are converted to type `long`.
- Otherwise, if either operand is of type `uint`, the other operand is converted to type `uint`.
- Otherwise, both operands are converted to type `int`.

Note that the first rule disallows any operations that mix the `decimal` type with the `double` and `float` types. The rule follows from the fact that there are no implicit conversions between the `decimal` type and the `double` and `float` types.

Also note that it is not possible for an operand to be of type `ulong` when the other operand is of a signed integral type. The reason is that no integral type exists that can represent the full range of `ulong` as well as the signed integral types.

In both of the above cases, a cast expression can be used to explicitly convert one operand to a type that is compatible with the other operand.

In the example

```
decimal AddPercent(decimal x, double percent) {
    return x * (1.0 + percent / 100.0);
}
```

a binding-time error occurs because a `decimal` cannot be multiplied by a `double`. The error is resolved by explicitly converting the second operand to `decimal`, as follows:

```
decimal AddPercent(decimal x, double percent) {
    return x * (decimal)(1.0 + percent / 100.0);
}
```

Lifted operators

Lifted operators permit predefined and user-defined operators that operate on non-nullable value types to also be used with nullable forms of those types. Lifted operators are constructed from predefined and user-defined operators that meet certain requirements, as described in the following:

- For the unary operators

  ```
  +  ++  -  --  !  ~
  ```

  a lifted form of an operator exists if the operand and result types are both non-nullable value types. The lifted form is constructed by adding a single `?` modifier to the operand and result types. The lifted operator produces a null value if the operand is null. Otherwise, the lifted operator unwraps the operand, applies the underlying operator, and wraps the result.

- For the binary operators

  ```
  +  -  *  /  %  &  |  ^  <<  >>
  ```

  a lifted form of an operator exists if the operand and result types are all non-nullable value types. The lifted form is constructed by adding a single `?` modifier to each operand and result type. The lifted operator produces a null value if one or both operands are null (an exception being the `&` and `|` operators of the `bool?` type, as described in Boolean logical operators). Otherwise, the lifted operator unwraps the operands, applies the underlying operator, and wraps the result.

- For the equality operators

  ```
  ==  !=
  ```

  a lifted form of an operator exists if the operand types are both non-nullable value types and if the result type is `bool`. The lifted form is constructed by adding a single `?` modifier to each operand type. The lifted operator considers two null values equal, and a null value unequal to any non-null value. If both operands are non-null, the lifted operator unwraps the operands and applies the underlying operator to produce the `bool` result.

- For the relational operators

  ```
  <  >  <=  >=
  ```

  a lifted form of an operator exists if the operand types are both non-nullable value types and if the result type is `bool`. The lifted form is constructed by adding a single `?` modifier to each operand type. The

lifted operator produces the value `false` if one or both operands are null. Otherwise, the lifted operator unwraps the operands and applies the underlying operator to produce the `bool` result.

Member lookup

A member lookup is the process whereby the meaning of a name in the context of a type is determined. A member lookup can occur as part of evaluating a *simple_name* (Simple names) or a *member_access* (Member access) in an expression. If the *simple_name* or *member_access* occurs as the *primary_expression* of an *invocation_expression* (Method invocations), the member is said to be invoked.

If a member is a method or event, or if it is a constant, field or property of either a delegate type (Delegates) or the type `dynamic` (The dynamic type), then the member is said to be *invocable*.

Member lookup considers not only the name of a member but also the number of type parameters the member has and whether the member is accessible. For the purposes of member lookup, generic methods and nested generic types have the number of type parameters indicated in their respective declarations and all other members have zero type parameters.

A member lookup of a name `N` with `K` type parameters in a type `T` is processed as follows:

- First, a set of accessible members named `N` is determined:
  - If `T` is a type parameter, then the set is the union of the sets of accessible members named `N` in each of the types specified as a primary constraint or secondary constraint (Type parameter constraints) for `T`, along with the set of accessible members named `N` in `object`.
  - Otherwise, the set consists of all accessible (Member access) members named `N` in `T`, including inherited members and the accessible members named `N` in `object`. If `T` is a constructed type, the set of members is obtained by substituting type arguments as described in Members of constructed types. Members that include an `override` modifier are excluded from the set.
- Next, if `K` is zero, all nested types whose declarations include type parameters are removed. If `K` is not zero, all members with a different number of type parameters are removed. Note that when `K` is zero, methods having type parameters are not removed, since the type inference process (Type inference) might be able to infer the type arguments.
- Next, if the member is *invoked*, all non-*invocable* members are removed from the set.
- Next, members that are hidden by other members are removed from the set. For every member `S.M` in the set, where `S` is the type in which the member `M` is declared, the following rules are applied:
  - If `M` is a constant, field, property, event, or enumeration member, then all members declared in a base type of `S` are removed from the set.
  - If `M` is a type declaration, then all non-types declared in a base type of `S` are removed from the set, and all type declarations with the same number of type parameters as `M` declared in a base type of `S` are removed from the set.
  - If `M` is a method, then all non-method members declared in a base type of `S` are removed from the set.
- Next, interface members that are hidden by class members are removed from the set. This step only has an effect if `T` is a type parameter and `T` has both an effective base class other than `object` and a non-empty effective interface set (Type parameter constraints). For every member `S.M` in the set, where `S` is the type in which the member `M` is declared, the following rules are applied if `S` is a class declaration other than `object`:
  - If `M` is a constant, field, property, event, enumeration member, or type declaration, then all members declared in an interface declaration are removed from the set.
  - If `M` is a method, then all non-method members declared in an interface declaration are removed from the set, and all methods with the same signature as `M` declared in an interface declaration are removed from the set.

- Finally, having removed hidden members, the result of the lookup is determined:
  - If the set consists of a single member that is not a method, then this member is the result of the lookup.
  - Otherwise, if the set contains only methods, then this group of methods is the result of the lookup.
  - Otherwise, the lookup is ambiguous, and a binding-time error occurs.

For member lookups in types other than type parameters and interfaces, and member lookups in interfaces that are strictly single-inheritance (each interface in the inheritance chain has exactly zero or one direct base interface), the effect of the lookup rules is simply that derived members hide base members with the same name or signature. Such single-inheritance lookups are never ambiguous. The ambiguities that can possibly arise from member lookups in multiple-inheritance interfaces are described in Interface member access.

Base types

For purposes of member lookup, a type `T` is considered to have the following base types:

- If `T` is `object`, then `T` has no base type.
- If `T` is an *enum_type*, the base types of `T` are the class types `System.Enum`, `System.ValueType`, and `object`.
- If `T` is a *struct_type*, the base types of `T` are the class types `System.ValueType` and `object`.
- If `T` is a *class_type*, the base types of `T` are the base classes of `T`, including the class type `object`.
- If `T` is an *interface_type*, the base types of `T` are the base interfaces of `T` and the class type `object`.
- If `T` is an *array_type*, the base types of `T` are the class types `System.Array` and `object`.
- If `T` is a *delegate_type*, the base types of `T` are the class types `System.Delegate` and `object`.

Function members

Function members are members that contain executable statements. Function members are always members of types and cannot be members of namespaces. C# defines the following categories of function members:

- Methods
- Properties
- Events
- Indexers
- User-defined operators
- Instance constructors
- Static constructors
- Destructors

Except for destructors and static constructors (which cannot be invoked explicitly), the statements contained in function members are executed through function member invocations. The actual syntax for writing a function member invocation depends on the particular function member category.

The argument list (Argument lists) of a function member invocation provides actual values or variable references for the parameters of the function member.

Invocations of generic methods may employ type inference to determine the set of type arguments to pass to the method. This process is described in Type inference.

Invocations of methods, indexers, operators and instance constructors employ overload resolution to determine which of a candidate set of function members to invoke. This process is described in Overload resolution.

Once a particular function member has been identified at binding-time, possibly through overload resolution, the actual run-time process of invoking the function member is described in Compile-time checking of dynamic overload resolution.

The following table summarizes the processing that takes place in constructs involving the six categories of function members that can be explicitly invoked. In the table, `e`, `x`, `y`, and `value` indicate expressions classified as variables or values, `T` indicates an expression classified as a type, `F` is the simple name of a method, and `P` is the simple name of a property.

CONSTRUCT	EXAMPLE	DESCRIPTION
Method invocation	`F(x,y)`	Overload resolution is applied to select the best method `F` in the containing class or struct. The method is invoked with the argument list `(x,y)`. If the method is not `static`, the instance expression is `this`.
	`T.F(x,y)`	Overload resolution is applied to select the best method `F` in the class or struct `T`. A binding-time error occurs if the method is not `static`. The method is invoked with the argument list `(x,y)`.
	`e.F(x,y)`	Overload resolution is applied to select the best method F in the class, struct, or interface given by the type of `e`. A binding-time error occurs if the method is `static`. The method is invoked with the instance expression `e` and the argument list `(x,y)`.
Property access	`P`	The `get` accessor of the property `P` in the containing class or struct is invoked. A compile-time error occurs if `P` is write-only. If `P` is not `static`, the instance expression is `this`.
	`P = value`	The `set` accessor of the property `P` in the containing class or struct is invoked with the argument list `(value)`. A compile-time error occurs if `P` is read-only. If `P` is not `static`, the instance expression is `this`.
	`T.P`	The `get` accessor of the property `P` in the class or struct `T` is invoked. A compile-time error occurs if `P` is not `static` or if `P` is write-only.
	`T.P = value`	The `set` accessor of the property `P` in the class or struct `T` is invoked with the argument list `(value)`. A compile-time error occurs if `P` is not `static` or if `P` is read-only.

CONSTRUCT	EXAMPLE	DESCRIPTION
	`e.P`	The `get` accessor of the property `P` in the class, struct, or interface given by the type of `e` is invoked with the instance expression `e`. A binding-time error occurs if `P` is `static` or if `P` is write-only.
	`e.P = value`	The `set` accessor of the property `P` in the class, struct, or interface given by the type of `e` is invoked with the instance expression `e` and the argument list `(value)`. A binding-time error occurs if `P` is `static` or if `P` is read-only.
Event access	`E += value`	The `add` accessor of the event `E` in the containing class or struct is invoked. If `E` is not static, the instance expression is `this`.
	`E -= value`	The `remove` accessor of the event `E` in the containing class or struct is invoked. If `E` is not static, the instance expression is `this`.
	`T.E += value`	The `add` accessor of the event `E` in the class or struct `T` is invoked. A binding-time error occurs if `E` is not static.
	`T.E -= value`	The `remove` accessor of the event `E` in the class or struct `T` is invoked. A binding-time error occurs if `E` is not static.
	`e.E += value`	The `add` accessor of the event `E` in the class, struct, or interface given by the type of `e` is invoked with the instance expression `e`. A binding-time error occurs if `E` is static.
	`e.E -= value`	The `remove` accessor of the event `E` in the class, struct, or interface given by the type of `e` is invoked with the instance expression `e`. A binding-time error occurs if `E` is static.

CONSTRUCT	EXAMPLE	DESCRIPTION
Indexer access	`e[x,y]`	Overload resolution is applied to select the best indexer in the class, struct, or interface given by the type of e. The `get` accessor of the indexer is invoked with the instance expression `e` and the argument list `(x,y)`. A binding-time error occurs if the indexer is write-only.
	`e[x,y] = value`	Overload resolution is applied to select the best indexer in the class, struct, or interface given by the type of `e`. The `set` accessor of the indexer is invoked with the instance expression `e` and the argument list `(x,y,value)`. A binding-time error occurs if the indexer is read-only.
Operator invocation	`-x`	Overload resolution is applied to select the best unary operator in the class or struct given by the type of `x`. The selected operator is invoked with the argument list `(x)`.
	`x + y`	Overload resolution is applied to select the best binary operator in the classes or structs given by the types of `x` and `y`. The selected operator is invoked with the argument list `(x,y)` .
Instance constructor invocation	`new T(x,y)`	Overload resolution is applied to select the best instance constructor in the class or struct `T`. The instance constructor is invoked with the argument list `(x,y)`.

Argument lists

Every function member and delegate invocation includes an argument list which provides actual values or variable references for the parameters of the function member. The syntax for specifying the argument list of a function member invocation depends on the function member category:

- For instance constructors, methods, indexers and delegates, the arguments are specified as an *argument_list*, as described below. For indexers, when invoking the `set` accessor, the argument list additionally includes the expression specified as the right operand of the assignment operator.
- For properties, the argument list is empty when invoking the `get` accessor, and consists of the expression specified as the right operand of the assignment operator when invoking the `set` accessor.
- For events, the argument list consists of the expression specified as the right operand of the `+=` or `-=` operator.
- For user-defined operators, the argument list consists of the single operand of the unary operator or the two operands of the binary operator.

The arguments of properties (Properties), events (Events), and user-defined operators (Operators) are always passed as value parameters (Value parameters). The arguments of indexers (Indexers) are always passed as

value parameters (Value parameters) or parameter arrays (Parameter arrays). Reference and output parameters are not supported for these categories of function members.

The arguments of an instance constructor, method, indexer or delegate invocation are specified as an *argument_list*.

```
argument_list
    : argument (',' argument)*
    ;

argument
    : argument_name? argument_value
    ;

argument_name
    : identifier ':'
    ;

argument_value
    : expression
    | 'ref' variable_reference
    | 'out' variable_reference
    ;
```

An *argument_list* consists of one or more *argument*s, separated by commas. Each argument consists of an optional *argument_name* followed by an *argument_value*. An *argument* with an *argument_name* is referred to as a ***named argument***, whereas an *argument* without an *argument_name* is a ***positional argument***. It is an error for a positional argument to appear after a named argument in an *argument_list*.

The *argument_value* can take one of the following forms:

- An *expression*, indicating that the argument is passed as a value parameter (Value parameters).
- The keyword `ref` followed by a *variable_reference* (Variable references), indicating that the argument is passed as a reference parameter (Reference parameters). A variable must be definitely assigned (Definite assignment) before it can be passed as a reference parameter. The keyword `out` followed by a *variable_reference* (Variable references), indicating that the argument is passed as an output parameter (Output parameters). A variable is considered definitely assigned (Definite assignment) following a function member invocation in which the variable is passed as an output parameter.

Corresponding parameters

For each argument in an argument list there has to be a corresponding parameter in the function member or delegate being invoked.

The parameter list used in the following is determined as follows:

- For virtual methods and indexers defined in classes, the parameter list is picked from the most specific declaration or override of the function member, starting with the static type of the receiver, and searching through its base classes.
- For interface methods and indexers, the parameter list is picked form the most specific definition of the member, starting with the interface type and searching through the base interfaces. If no unique parameter list is found, a parameter list with inaccessible names and no optional parameters is constructed, so that invocations cannot use named parameters or omit optional arguments.
- For partial methods, the parameter list of the defining partial method declaration is used.
- For all other function members and delegates there is only a single parameter list, which is the one used.

The position of an argument or parameter is defined as the number of arguments or parameters preceding it in the argument list or parameter list.

The corresponding parameters for function member arguments are established as follows:

- Arguments in the *argument_list* of instance constructors, methods, indexers and delegates:
  - A positional argument where a fixed parameter occurs at the same position in the parameter list corresponds to that parameter.
  - A positional argument of a function member with a parameter array invoked in its normal form corresponds to the parameter array, which must occur at the same position in the parameter list.
  - A positional argument of a function member with a parameter array invoked in its expanded form, where no fixed parameter occurs at the same position in the parameter list, corresponds to an element in the parameter array.
  - A named argument corresponds to the parameter of the same name in the parameter list.
  - For indexers, when invoking the `set` accessor, the expression specified as the right operand of the assignment operator corresponds to the implicit `value` parameter of the `set` accessor declaration.
- For properties, when invoking the `get` accessor there are no arguments. When invoking the `set` accessor, the expression specified as the right operand of the assignment operator corresponds to the implicit `value` parameter of the `set` accessor declaration.
- For user-defined unary operators (including conversions), the single operand corresponds to the single parameter of the operator declaration.
- For user-defined binary operators, the left operand corresponds to the first parameter, and the right operand corresponds to the second parameter of the operator declaration.

Run-time evaluation of argument lists

During the run-time processing of a function member invocation (Compile-time checking of dynamic overload resolution), the expressions or variable references of an argument list are evaluated in order, from left to right, as follows:

- For a value parameter, the argument expression is evaluated and an implicit conversion (Implicit conversions) to the corresponding parameter type is performed. The resulting value becomes the initial value of the value parameter in the function member invocation.
- For a reference or output parameter, the variable reference is evaluated and the resulting storage location becomes the storage location represented by the parameter in the function member invocation. If the variable reference given as a reference or output parameter is an array element of a *reference_type*, a run-time check is performed to ensure that the element type of the array is identical to the type of the parameter. If this check fails, a `System.ArrayTypeMismatchException` is thrown.

Methods, indexers, and instance constructors may declare their right-most parameter to be a parameter array (Parameter arrays). Such function members are invoked either in their normal form or in their expanded form depending on which is applicable (Applicable function member):

- When a function member with a parameter array is invoked in its normal form, the argument given for the parameter array must be a single expression that is implicitly convertible (Implicit conversions) to the parameter array type. In this case, the parameter array acts precisely like a value parameter.
- When a function member with a parameter array is invoked in its expanded form, the invocation must specify zero or more positional arguments for the parameter array, where each argument is an expression that is implicitly convertible (Implicit conversions) to the element type of the parameter array. In this case, the invocation creates an instance of the parameter array type with a length corresponding to the number of arguments, initializes the elements of the array instance with the given argument values, and uses the newly created array instance as the actual argument.

The expressions of an argument list are always evaluated in the order they are written. Thus, the example

```
class Test
{
    static void F(int x, int y = -1, int z = -2) {
        System.Console.WriteLine("x = {0}, y = {1}, z = {2}", x, y, z);
    }

    static void Main() {
        int i = 0;
        F(i++, i++, i++);
        F(z: i++, x: i++);
    }
}
```

produces the output

```
x = 0, y = 1, z = 2
x = 4, y = -1, z = 3
```

The array co-variance rules (Array covariance) permit a value of an array type `A[]` to be a reference to an instance of an array type `B[]`, provided an implicit reference conversion exists from `B` to `A`. Because of these rules, when an array element of a *reference_type* is passed as a reference or output parameter, a run-time check is required to ensure that the actual element type of the array is identical to that of the parameter. In the example

```
class Test
{
    static void F(ref object x) {...}

    static void Main() {
        object[] a = new object[10];
        object[] b = new string[10];
        F(ref a[0]);        // Ok
        F(ref b[1]);        // ArrayTypeMismatchException
    }
}
```

the second invocation of `F` causes a `System.ArrayTypeMismatchException` to be thrown because the actual element type of `b` is `string` and not `object`.

When a function member with a parameter array is invoked in its expanded form, the invocation is processed exactly as if an array creation expression with an array initializer (Array creation expressions) was inserted around the expanded parameters. For example, given the declaration

```
void F(int x, int y, params object[] args);
```

the following invocations of the expanded form of the method

```
F(10, 20);
F(10, 20, 30, 40);
F(10, 20, 1, "hello", 3.0);
```

correspond exactly to

```
F(10, 20, new object[] {});
F(10, 20, new object[] {30, 40});
F(10, 20, new object[] {1, "hello", 3.0});
```

In particular, note that an empty array is created when there are zero arguments given for the parameter array.

When arguments are omitted from a function member with corresponding optional parameters, the default arguments of the function member declaration are implicitly passed. Because these are always constant, their evaluation will not impact the evaluation order of the remaining arguments.

Type inference

When a generic method is called without specifying type arguments, a ***type inference*** process attempts to infer type arguments for the call. The presence of type inference allows a more convenient syntax to be used for calling a generic method, and allows the programmer to avoid specifying redundant type information. For example, given the method declaration:

```
class Chooser
{
    static Random rand = new Random();

    public static T Choose<T>(T first, T second) {
        return (rand.Next(2) == 0)? first: second;
    }
}
```

it is possible to invoke the `Choose` method without explicitly specifying a type argument:

```
int i = Chooser.Choose(5, 213);                // Calls Choose<int>

string s = Chooser.Choose("foo", "bar");       // Calls Choose<string>
```

Through type inference, the type arguments `int` and `string` are determined from the arguments to the method.

Type inference occurs as part of the binding-time processing of a method invocation (Method invocations) and takes place before the overload resolution step of the invocation. When a particular method group is specified in a method invocation, and no type arguments are specified as part of the method invocation, type inference is applied to each generic method in the method group. If type inference succeeds, then the inferred type arguments are used to determine the types of arguments for subsequent overload resolution. If overload resolution chooses a generic method as the one to invoke, then the inferred type arguments are used as the actual type arguments for the invocation. If type inference for a particular method fails, that method does not participate in overload resolution. The failure of type inference, in and of itself, does not cause a binding-time error. However, it often leads to a binding-time error when overload resolution then fails to find any applicable methods.

If the supplied number of arguments is different than the number of parameters in the method, then inference immediately fails. Otherwise, assume that the generic method has the following signature:

```
Tr M<X1,...,Xn>(T1 x1, ..., Tm xm)
```

With a method call of the form `M(E1...Em)` the task of type inference is to find unique type arguments `S1...Sn` for each of the type parameters `X1...Xn` so that the call `M<S1...Sn>(E1...Em)` becomes valid.

During the process of inference each type parameter `Xi` is either *fixed* to a particular type `Si` or *unfixed* with an associated set of *bounds*. Each of the bounds is some type `T`. Initially each type variable `Xi` is unfixed with an empty set of bounds.

Type inference takes place in phases. Each phase will try to infer type arguments for more type variables based on the findings of the previous phase. The first phase makes some initial inferences of bounds, whereas the

second phase fixes type variables to specific types and infers further bounds. The second phase may have to be repeated a number of times.

Note: Type inference takes place not only when a generic method is called. Type inference for conversion of method groups is described in Type inference for conversion of method groups and finding the best common type of a set of expressions is described in Finding the best common type of a set of expressions.

The first phase

For each of the method arguments `Ei`:

- If `Ei` is an anonymous function, an *explicit parameter type inference* (Explicit parameter type inferences) is made from `Ei` to `Ti`
- Otherwise, if `Ei` has a type `U` and `xi` is a value parameter then a *lower-bound inference* is made *from* `U` *to* `Ti`.
- Otherwise, if `Ei` has a type `U` and `xi` is a `ref` or `out` parameter then an *exact inference* is made *from* `U` *to* `Ti`.
- Otherwise, no inference is made for this argument.

The second phase

The second phase proceeds as follows:

- All *unfixed* type variables `Xi` which do not *depend on* (Dependence) any `Xj` are fixed (Fixing).
- If no such type variables exist, all *unfixed* type variables `Xi` are *fixed* for which all of the following hold:
  - There is at least one type variable `Xj` that depends on `Xi`
  - `Xi` has a non-empty set of bounds
- If no such type variables exist and there are still *unfixed* type variables, type inference fails.
- Otherwise, if no further *unfixed* type variables exist, type inference succeeds.
- Otherwise, for all arguments `Ei` with corresponding parameter type `Ti` where the *output types* (Output types) contain *unfixed* type variables `Xj` but the *input types* (Input types) do not, an *output type inference* (Output type inferences) is made *from* `Ei` *to* `Ti`. Then the second phase is repeated.

Input types

If `E` is a method group or implicitly typed anonymous function and `T` is a delegate type or expression tree type then all the parameter types of `T` are *input types* of `E` *with type* `T`.

Output types

If `E` is a method group or an anonymous function and `T` is a delegate type or expression tree type then the return type of `T` is an *output type of* `E` *with type* `T`.

Dependence

An *unfixed* type variable `Xi` *depends directly on* an unfixed type variable `Xj` if for some argument `Ek` with type `Tk` `Xj` occurs in an *input type* of `Ek` with type `Tk` and `Xi` occurs in an *output type* of `Ek` with type `Tk`.

`Xj` *depends on* `Xi` if `Xj` *depends directly on* `Xi` or if `Xi` *depends directly on* `Xk` and `Xk` *depends on* `Xj`. Thus "depends on" is the transitive but not reflexive closure of "depends directly on".

Output type inferences

An *output type inference* is made *from* an expression `E` *to* a type `T` in the following way:

- If `E` is an anonymous function with inferred return type `U` (Inferred return type) and `T` is a delegate type or expression tree type with return type `Tb`, then a *lower-bound inference* (Lower-bound inferences) is made *from* `U` *to* `Tb`.
- Otherwise, if `E` is a method group and `T` is a delegate type or expression tree type with parameter types `T1...Tk` and return type `Tb`, and overload resolution of `E` with the types `T1...Tk` yields a single method

with return type `U`, then a *lower-bound inference* is made *from* `U` *to* `Tb`.

- Otherwise, if `E` is an expression with type `U`, then a *lower-bound inference* is made *from* `U` *to* `T`.
- Otherwise, no inferences are made.

Explicit parameter type inferences

An *explicit parameter type inference* is made *from* an expression `E` *to* a type `T` in the following way:

- If `E` is an explicitly typed anonymous function with parameter types `U1...Uk` and `T` is a delegate type or expression tree type with parameter types `V1...Vk` then for each `Ui` an *exact inference* (Exact inferences) is made *from* `Ui` *to* the corresponding `Vi`.

Exact inferences

An *exact inference from* a type `U` *to* a type `V` is made as follows:

- If `V` is one of the *unfixed* `Xi` then `U` is added to the set of exact bounds for `Xi`.
- Otherwise, sets `V1...Vk` and `U1...Uk` are determined by checking if any of the following cases apply:
  - `V` is an array type `V1[...]` and `U` is an array type `U1[...]` of the same rank
  - `V` is the type `V1?` and `U` is the type `U1?`
  - `V` is a constructed type `C<V1...Vk>` and `U` is a constructed type `C<U1...Uk>`

  If any of these cases apply then an *exact inference* is made *from* each `Ui` *to* the corresponding `Vi`.
- Otherwise no inferences are made.

Lower-bound inferences

A *lower-bound inference from* a type `U` *to* a type `V` is made as follows:

- If `V` is one of the *unfixed* `Xi` then `U` is added to the set of lower bounds for `Xi`.
- Otherwise, if `V` is the type `V1?` and `U` is the type `U1?` then a lower bound inference is made from `U1` to `V1`.
- Otherwise, sets `U1...Uk` and `V1...Vk` are determined by checking if any of the following cases apply:
  - `V` is an array type `V1[...]` and `U` is an array type `U1[...]` (or a type parameter whose effective base type is `U1[...]`) of the same rank
  - `V` is one of `IEnumerable<V1>`, `ICollection<V1>` or `IList<V1>` and `U` is a one-dimensional array type `U1[]` (or a type parameter whose effective base type is `U1[]`)
  - `V` is a constructed class, struct, interface or delegate type `C<V1...Vk>` and there is a unique type `C<U1...Uk>` such that `U` (or, if `U` is a type parameter, its effective base class or any member of its effective interface set) is identical to, inherits from (directly or indirectly), or implements (directly or indirectly) `C<U1...Uk>`.

    (The "uniqueness" restriction means that in the case interface `C<T> {} class U: C<X>, C<Y> {}`, then no inference is made when inferring from `U` to `C<T>` because `U1` could be `X` or `Y`.)

  If any of these cases apply then an inference is made *from* each `Ui` *to* the corresponding `Vi` as follows:

  - If `Ui` is not known to be a reference type then an *exact inference* is made
  - Otherwise, if `U` is an array type then a *lower-bound inference* is made
  - Otherwise, if `V` is `C<V1...Vk>` then inference depends on the i-th type parameter of `C`:
    - If it is covariant then a *lower-bound inference* is made.
    - If it is contravariant then an *upper-bound inference* is made.
    - If it is invariant then an *exact inference* is made.

- Otherwise, no inferences are made.

Upper-bound inferences

An *upper-bound inference from* a type `U` *to* a type `V` is made as follows:

- If `V` is one of the *unfixed* `Xi` then `U` is added to the set of upper bounds for `Xi`.
- Otherwise, sets `V1...Vk` and `U1...Uk` are determined by checking if any of the following cases apply:
  - `U` is an array type `U1[...]` and `V` is an array type `V1[...]` of the same rank
  - `U` is one of `IEnumerable<Ue>`, `ICollection<Ue>` or `IList<Ue>` and `V` is a one-dimensional array type `Ve[]`
  - `U` is the type `U1?` and `V` is the type `V1?`
  - `U` is constructed class, struct, interface or delegate type `C<U1...Uk>` and `V` is a class, struct, interface or delegate type which is identical to, inherits from (directly or indirectly), or implements (directly or indirectly) a unique type `C<V1...Vk>`

    (The "uniqueness" restriction means that if we have `interface C<T>{} class V<Z>: C<X<Z>>, C<Y<Z>>{}`, then no inference is made when inferring from `C<U1>` to `V<Q>`. Inferences are not made from `U1` to either `X<Q>` or `Y<Q>`.)

  If any of these cases apply then an inference is made *from* each `Ui` *to* the corresponding `Vi` as follows:

  - If `Ui` is not known to be a reference type then an *exact inference* is made
  - Otherwise, if `V` is an array type then an *upper-bound inference* is made
  - Otherwise, if `U` is `C<U1...Uk>` then inference depends on the i-th type parameter of `C`:
    - If it is covariant then an *upper-bound inference* is made.
    - If it is contravariant then a *lower-bound inference* is made.
    - If it is invariant then an *exact inference* is made.
- Otherwise, no inferences are made.

Fixing

An *unfixed* type variable `Xi` with a set of bounds is *fixed* as follows:

- The set of *candidate types* `Uj` starts out as the set of all types in the set of bounds for `Xi`.
- We then examine each bound for `Xi` in turn: For each exact bound `U` of `Xi` all types `Uj` which are not identical to `U` are removed from the candidate set. For each lower bound `U` of `Xi` all types `Uj` to which there is *not* an implicit conversion from `U` are removed from the candidate set. For each upper bound `U` of `Xi` all types `Uj` from which there is *not* an implicit conversion to `U` are removed from the candidate set.
- If among the remaining candidate types `Uj` there is a unique type `V` from which there is an implicit conversion to all the other candidate types, then `Xi` is fixed to `V`.
- Otherwise, type inference fails.

Inferred return type

The inferred return type of an anonymous function `F` is used during type inference and overload resolution. The inferred return type can only be determined for an anonymous function where all parameter types are known, either because they are explicitly given, provided through an anonymous function conversion or inferred during type inference on an enclosing generic method invocation.

The ***inferred result type*** is determined as follows:

- If the body of `F` is an *expression* that has a type, then the inferred result type of `F` is the type of that expression.

- If the body of `F` is a *block* and the set of expressions in the block's `return` statements has a best common type `T` (Finding the best common type of a set of expressions), then the inferred result type of `F` is `T`.
- Otherwise, a result type cannot be inferred for `F`.

The ***inferred return type*** is determined as follows:

- If `F` is async and the body of `F` is either an expression classified as nothing (Expression classifications), or a statement block where no return statements have expressions, the inferred return type is `System.Threading.Tasks.Task`
- If `F` is async and has an inferred result type `T`, the inferred return type is `System.Threading.Tasks.Task<T>`.
- If `F` is non-async and has an inferred result type `T`, the inferred return type is `T`.
- Otherwise a return type cannot be inferred for `F`.

As an example of type inference involving anonymous functions, consider the `Select` extension method declared in the `System.Linq.Enumerable` class:

```
namespace System.Linq
{
    public static class Enumerable
    {
        public static IEnumerable<TResult> Select<TSource,TResult>(
            this IEnumerable<TSource> source,
            Func<TSource,TResult> selector)
        {
            foreach (TSource element in source) yield return selector(element);
        }
    }
}
```

Assuming the `System.Linq` namespace was imported with a `using` clause, and given a class `Customer` with a `Name` property of type `string`, the `Select` method can be used to select the names of a list of customers:

```
List<Customer> customers = GetCustomerList();
IEnumerable<string> names = customers.Select(c => c.Name);
```

The extension method invocation (Extension method invocations) of `Select` is processed by rewriting the invocation to a static method invocation:

```
IEnumerable<string> names = Enumerable.Select(customers, c => c.Name);
```

Since type arguments were not explicitly specified, type inference is used to infer the type arguments. First, the `customers` argument is related to the `source` parameter, inferring `T` to be `Customer`. Then, using the anonymous function type inference process described above, `c` is given type `Customer`, and the expression `c.Name` is related to the return type of the `selector` parameter, inferring `S` to be `string`. Thus, the invocation is equivalent to

```
Sequence.Select<Customer,string>(customers, (Customer c) => c.Name)
```

and the result is of type `IEnumerable<string>`.

The following example demonstrates how anonymous function type inference allows type information to "flow" between arguments in a generic method invocation. Given the method:

```
static Z F<X,Y,Z>(X value, Func<X,Y> f1, Func<Y,Z> f2) {
    return f2(f1(value));
}
```

Type inference for the invocation:

```
double seconds = F("1:15:30", s => TimeSpan.Parse(s), t => t.TotalSeconds);
```

proceeds as follows: First, the argument `"1:15:30"` is related to the `value` parameter, inferring `X` to be `string`. Then, the parameter of the first anonymous function, `s`, is given the inferred type `string`, and the expression `TimeSpan.Parse(s)` is related to the return type of `f1`, inferring `Y` to be `System.TimeSpan`. Finally, the parameter of the second anonymous function, `t`, is given the inferred type `System.TimeSpan`, and the expression `t.TotalSeconds` is related to the return type of `f2`, inferring `Z` to be `double`. Thus, the result of the invocation is of type `double`.

Type inference for conversion of method groups

Similar to calls of generic methods, type inference must also be applied when a method group `M` containing a generic method is converted to a given delegate type `D` (Method group conversions). Given a method

```
Tr M<X1...Xn>(T1 x1 ... Tm xm)
```

and the method group `M` being assigned to the delegate type `D` the task of type inference is to find type arguments `S1...Sn` so that the expression:

```
M<S1...Sn>
```

becomes compatible (Delegate declarations) with `D`.

Unlike the type inference algorithm for generic method calls, in this case there are only argument *types*, no argument *expressions*. In particular, there are no anonymous functions and hence no need for multiple phases of inference.

Instead, all `Xi` are considered *unfixed*, and a *lower-bound inference* is made *from* each argument type `Uj` of `D` *to* the corresponding parameter type `Tj` of `M`. If for any of the `Xi` no bounds were found, type inference fails. Otherwise, all `Xi` are *fixed* to corresponding `Si`, which are the result of type inference.

Finding the best common type of a set of expressions

In some cases, a common type needs to be inferred for a set of expressions. In particular, the element types of implicitly typed arrays and the return types of anonymous functions with *block* bodies are found in this way.

Intuitively, given a set of expressions `E1...Em` this inference should be equivalent to calling a method

```
Tr M<X>(X x1 ... X xm)
```

with the `Ei` as arguments.

More precisely, the inference starts out with an *unfixed* type variable `X`. *Output type inferences* are then made *from* each `Ei` *to* `X`. Finally, `X` is *fixed* and, if successful, the resulting type `S` is the resulting best common type for the expressions. If no such `S` exists, the expressions have no best common type.

Overload resolution

Overload resolution is a binding-time mechanism for selecting the best function member to invoke given an

argument list and a set of candidate function members. Overload resolution selects the function member to invoke in the following distinct contexts within C#:

- Invocation of a method named in an *invocation_expression* (Method invocations).
- Invocation of an instance constructor named in an *object_creation_expression* (Object creation expressions).
- Invocation of an indexer accessor through an *element_access* (Element access).
- Invocation of a predefined or user-defined operator referenced in an expression (Unary operator overload resolution and Binary operator overload resolution).

Each of these contexts defines the set of candidate function members and the list of arguments in its own unique way, as described in detail in the sections listed above. For example, the set of candidates for a method invocation does not include methods marked `override` (Member lookup), and methods in a base class are not candidates if any method in a derived class is applicable (Method invocations).

Once the candidate function members and the argument list have been identified, the selection of the best function member is the same in all cases:

- Given the set of applicable candidate function members, the best function member in that set is located. If the set contains only one function member, then that function member is the best function member. Otherwise, the best function member is the one function member that is better than all other function members with respect to the given argument list, provided that each function member is compared to all other function members using the rules in Better function member. If there is not exactly one function member that is better than all other function members, then the function member invocation is ambiguous and a binding-time error occurs.

The following sections define the exact meanings of the terms ***applicable function member*** and ***better function member***.

Applicable function member

A function member is said to be an ***applicable function member*** with respect to an argument list `A` when all of the following are true:

- Each argument in `A` corresponds to a parameter in the function member declaration as described in Corresponding parameters, and any parameter to which no argument corresponds is an optional parameter.
- For each argument in `A`, the parameter passing mode of the argument (i.e., value, `ref`, or `out`) is identical to the parameter passing mode of the corresponding parameter, and
  - for a value parameter or a parameter array, an implicit conversion (Implicit conversions) exists from the argument to the type of the corresponding parameter, or
  - for a `ref` or `out` parameter, the type of the argument is identical to the type of the corresponding parameter. After all, a `ref` or `out` parameter is an alias for the argument passed.

For a function member that includes a parameter array, if the function member is applicable by the above rules, it is said to be applicable in its ***normal form***. If a function member that includes a parameter array is not applicable in its normal form, the function member may instead be applicable in its ***expanded form***:

- The expanded form is constructed by replacing the parameter array in the function member declaration with zero or more value parameters of the element type of the parameter array such that the number of arguments in the argument list `A` matches the total number of parameters. If `A` has fewer arguments than the number of fixed parameters in the function member declaration, the expanded form of the function member cannot be constructed and is thus not applicable.
- Otherwise, the expanded form is applicable if for each argument in `A` the parameter passing mode of the argument is identical to the parameter passing mode of the corresponding parameter, and
  - for a fixed value parameter or a value parameter created by the expansion, an implicit conversion (Implicit conversions) exists from the type of the argument to the type of the corresponding parameter, or

- for a `ref` or `out` parameter, the type of the argument is identical to the type of the corresponding parameter.

Better function member

For the purposes of determining the better function member, a stripped-down argument list A is constructed containing just the argument expressions themselves in the order they appear in the original argument list.

Parameter lists for each of the candidate function members are constructed in the following way:

- The expanded form is used if the function member was applicable only in the expanded form.
- Optional parameters with no corresponding arguments are removed from the parameter list
- The parameters are reordered so that they occur at the same position as the corresponding argument in the argument list.

Given an argument list `A` with a set of argument expressions `{E1, E2, ..., En}` and two applicable function members `Mp` and `Mq` with parameter types `{P1, P2, ..., Pn}` and `{Q1, Q2, ..., Qn}`, `Mp` is defined to be a ***better function member*** than `Mq` if

- for each argument, the implicit conversion from `Ex` to `Qx` is not better than the implicit conversion from `Ex` to `Px`, and
- for at least one argument, the conversion from `Ex` to `Px` is better than the conversion from `Ex` to `Qx`.

When performing this evaluation, if `Mp` or `Mq` is applicable in its expanded form, then `Px` or `Qx` refers to a parameter in the expanded form of the parameter list.

In case the parameter type sequences `{P1, P2, ..., Pn}` and `{Q1, Q2, ..., Qn}` are equivalent (i.e. each `Pi` has an identity conversion to the corresponding `Qi`), the following tie-breaking rules are applied, in order, to determine the better function member.

- If `Mp` is a non-generic method and `Mq` is a generic method, then `Mp` is better than `Mq`.
- Otherwise, if `Mp` is applicable in its normal form and `Mq` has a `params` array and is applicable only in its expanded form, then `Mp` is better than `Mq`.
- Otherwise, if `Mp` has more declared parameters than `Mq`, then `Mp` is better than `Mq`. This can occur if both methods have `params` arrays and are applicable only in their expanded forms.
- Otherwise if all parameters of `Mp` have a corresponding argument whereas default arguments need to be substituted for at least one optional parameter in `Mq` then `Mp` is better than `Mq`.
- Otherwise, if `Mp` has more specific parameter types than `Mq`, then `Mp` is better than `Mq`. Let `{R1, R2, ..., Rn}` and `{S1, S2, ..., Sn}` represent the uninstantiated and unexpanded parameter types of `Mp` and `Mq`. `Mp`'s parameter types are more specific than `Mq`'s if, for each parameter, `Rx` is not less specific than `Sx`, and, for at least one parameter, `Rx` is more specific than `Sx`:
  - A type parameter is less specific than a non-type parameter.
  - Recursively, a constructed type is more specific than another constructed type (with the same number of type arguments) if at least one type argument is more specific and no type argument is less specific than the corresponding type argument in the other.
  - An array type is more specific than another array type (with the same number of dimensions) if the element type of the first is more specific than the element type of the second.
- Otherwise if one member is a non-lifted operator and the other is a lifted operator, the non-lifted one is better.
- Otherwise, neither function member is better.

Better conversion from expression

Given an implicit conversion `C1` that converts from an expression `E` to a type `T1`, and an implicit conversion `C2` that converts from an expression `E` to a type `T2`, `C1` is a ***better conversion*** than `C2` if `E` does not

exactly match `T2` and at least one of the following holds:

- `E` exactly matches `T1` (Exactly matching Expression)
- `T1` is a better conversion target than `T2` (Better conversion target)

Exactly matching Expression

Given an expression `E` and a type `T`, `E` exactly matches `T` if one of the following holds:

- `E` has a type `S`, and an identity conversion exists from `S` to `T`
- `E` is an anonymous function, `T` is either a delegate type `D` or an expression tree type `Expression<D>` and one of the following holds:
  - An inferred return type `X` exists for `E` in the context of the parameter list of `D` (Inferred return type), and an identity conversion exists from `X` to the return type of `D`
  - Either `E` is non-async and `D` has a return type `Y` or `E` is async and `D` has a return type `Task<Y>`, and one of the following holds:
    - The body of `E` is an expression that exactly matches `Y`
    - The body of `E` is a statement block where every return statement returns an expression that exactly matches `Y`

Better conversion target

Given two different types `T1` and `T2`, `T1` is a better conversion target than `T2` if no implicit conversion from `T2` to `T1` exists, and at least one of the following holds:

- An implicit conversion from `T1` to `T2` exists
- `T1` is either a delegate type `D1` or an expression tree type `Expression<D1>`, `T2` is either a delegate type `D2` or an expression tree type `Expression<D2>`, `D1` has a return type `S1` and one of the following holds:
  - `D2` is void returning
  - `D2` has a return type `S2`, and `S1` is a better conversion target than `S2`
- `T1` is `Task<S1>`, `T2` is `Task<S2>`, and `S1` is a better conversion target than `S2`
- `T1` is `S1` or `S1?` where `S1` is a signed integral type, and `T2` is `S2` or `S2?` where `S2` is an unsigned integral type. Specifically:
  - `S1` is `sbyte` and `S2` is `byte`, `ushort`, `uint`, or `ulong`
  - `S1` is `short` and `S2` is `ushort`, `uint`, or `ulong`
  - `S1` is `int` and `S2` is `uint`, or `ulong`
  - `S1` is `long` and `S2` is `ulong`

Overloading in generic classes

While signatures as declared must be unique, it is possible that substitution of type arguments results in identical signatures. In such cases, the tie-breaking rules of overload resolution above will pick the most specific member.

The following examples show overloads that are valid and invalid according to this rule:

```
interface I1<T> {...}

interface I2<T> {...}

class G1<U>
{
    int F1(U u);                  // Overload resolution for G<int>.F1
    int F1(int i);                // will pick non-generic

    void F2(I1<U> a);             // Valid overload
    void F2(I2<U> a);
}

class G2<U,V>
{
    void F3(U u, V v);            // Valid, but overload resolution for
    void F3(V v, U u);            // G2<int,int>.F3 will fail

    void F4(U u, I1<V> v);        // Valid, but overload resolution for
    void F4(I1<V> v, U u);        // G2<I1<int>,int>.F4 will fail

    void F5(U u1, I1<V> v2);      // Valid overload
    void F5(V v1, U u2);

    void F6(ref U u);             // valid overload
    void F6(out V v);
}
```

Compile-time checking of dynamic overload resolution

For most dynamically bound operations the set of possible candidates for resolution is unknown at compile-time. In certain cases, however the candidate set is known at compile-time:

- Static method calls with dynamic arguments
- Instance method calls where the receiver is not a dynamic expression
- Indexer calls where the receiver is not a dynamic expression
- Constructor calls with dynamic arguments

In these cases a limited compile-time check is performed for each candidate to see if any of them could possibly apply at run-time.This check consists of the following steps:

- Partial type inference: Any type argument that does not depend directly or indirectly on an argument of type `dynamic` is inferred using the rules of Type inference. The remaining type arguments are unknown.
- Partial applicability check: Applicability is checked according to Applicable function member, but ignoring parameters whose types are unknown.
- If no candidate passes this test, a compile-time error occurs.

Function member invocation

This section describes the process that takes place at run-time to invoke a particular function member. It is assumed that a binding-time process has already determined the particular member to invoke, possibly by applying overload resolution to a set of candidate function members.

For purposes of describing the invocation process, function members are divided into two categories:

- Static function members. These are instance constructors, static methods, static property accessors, and user-defined operators. Static function members are always non-virtual.
- Instance function members. These are instance methods, instance property accessors, and indexer accessors. Instance function members are either non-virtual or virtual, and are always invoked on a particular instance. The instance is computed by an instance expression, and it becomes accessible within the function member as `this` (This access).

The run-time processing of a function member invocation consists of the following steps, where `M` is the function member and, if `M` is an instance member, `E` is the instance expression:

- If `M` is a static function member:
  - The argument list is evaluated as described in Argument lists.
  - `M` is invoked.
- If `M` is an instance function member declared in a *value_type*:
  - `E` is evaluated. If this evaluation causes an exception, then no further steps are executed.
  - If `E` is not classified as a variable, then a temporary local variable of `E`'s type is created and the value of `E` is assigned to that variable. `E` is then reclassified as a reference to that temporary local variable. The temporary variable is accessible as `this` within `M`, but not in any other way. Thus, only when `E` is a true variable is it possible for the caller to observe the changes that `M` makes to `this`.
  - The argument list is evaluated as described in Argument lists.
  - `M` is invoked. The variable referenced by `E` becomes the variable referenced by `this`.
- If `M` is an instance function member declared in a *reference_type*:
  - `E` is evaluated. If this evaluation causes an exception, then no further steps are executed.
  - The argument list is evaluated as described in Argument lists.
  - If the type of `E` is a *value_type*, a boxing conversion (Boxing conversions) is performed to convert `E` to type `object`, and `E` is considered to be of type `object` in the following steps. In this case, `M` could only be a member of `System.Object`.
  - The value of `E` is checked to be valid. If the value of `E` is `null`, a `System.NullReferenceException` is thrown and no further steps are executed.
  - The function member implementation to invoke is determined:
    - If the binding-time type of `E` is an interface, the function member to invoke is the implementation of `M` provided by the run-time type of the instance referenced by `E`. This function member is determined by applying the interface mapping rules (Interface mapping) to determine the implementation of `M` provided by the run-time type of the instance referenced by `E`.
    - Otherwise, if `M` is a virtual function member, the function member to invoke is the implementation of `M` provided by the run-time type of the instance referenced by `E`. This function member is determined by applying the rules for determining the most derived implementation (Virtual methods) of `M` with respect to the run-time type of the instance referenced by `E`.
    - Otherwise, `M` is a non-virtual function member, and the function member to invoke is `M` itself.
  - The function member implementation determined in the step above is invoked. The object referenced by `E` becomes the object referenced by `this`.

Invocations on boxed instances

A function member implemented in a *value_type* can be invoked through a boxed instance of that *value_type* in the following situations:

- When the function member is an `override` of a method inherited from type `object` and is invoked through an instance expression of type `object`.
- When the function member is an implementation of an interface function member and is invoked through an instance expression of an *interface_type*.
- When the function member is invoked through a delegate.

In these situations, the boxed instance is considered to contain a variable of the *value_type*, and this variable becomes the variable referenced by `this` within the function member invocation. In particular, this means that

when a function member is invoked on a boxed instance, it is possible for the function member to modify the value contained in the boxed instance.

Primary expressions

Primary expressions include the simplest forms of expressions.

```
primary_expression
    : primary_no_array_creation_expression
    | array_creation_expression
    ;

primary_no_array_creation_expression
    : literal
    | interpolated_string_expression
    | simple_name
    | parenthesized_expression
    | member_access
    | invocation_expression
    | element_access
    | this_access
    | base_access
    | post_increment_expression
    | post_decrement_expression
    | object_creation_expression
    | delegate_creation_expression
    | anonymous_object_creation_expression
    | typeof_expression
    | checked_expression
    | unchecked_expression
    | default_value_expression
    | nameof_expression
    | anonymous_method_expression
    | primary_no_array_creation_expression_unsafe
    ;
```

Primary expressions are divided between *array_creation_expression*s and *primary_no_array_creation_expression*s. Treating array-creation-expression in this way, rather than listing it along with the other simple expression forms, enables the grammar to disallow potentially confusing code such as

```
object o = new int[3][1];
```

which would otherwise be interpreted as

```
object o = (new int[3])[1];
```

Literals

A *primary_expression* that consists of a *literal* (Literals) is classified as a value.

Interpolated strings

An *interpolated_string_expression* consists of a `$` sign followed by a regular or verbatim string literal, wherein holes, delimited by `{` and `}`, enclose expressions and formatting specifications. An interpolated string expression is the result of an *interpolated_string_literal* that has been broken up into individual tokens, as described in Interpolated string literals.

```
interpolated_string_expression
    : '$' interpolated_regular_string
    | '$' interpolated_verbatim_string
    ;

interpolated_regular_string
    : interpolated_regular_string_whole
    | interpolated_regular_string_start interpolated_regular_string_body interpolated_regular_string_end
    ;

interpolated_regular_string_body
    : interpolation (interpolated_regular_string_mid interpolation)*
    ;

interpolation
    : expression
    | expression ',' constant_expression
    ;

interpolated_verbatim_string
    : interpolated_verbatim_string_whole
    | interpolated_verbatim_string_start interpolated_verbatim_string_body interpolated_verbatim_string_end
    ;

interpolated_verbatim_string_body
    : interpolation (interpolated_verbatim_string_mid interpolation)+
    ;
```

The *constant_expression* in an interpolation must have an implicit conversion to `int`.

An *interpolated_string_expression* is classified as a value. If it is immediately converted to `System.IFormattable` or `System.FormattableString` with an implicit interpolated string conversion (Implicit interpolated string conversions), the interpolated string expression has that type. Otherwise, it has the type `string`.

If the type of an interpolated string is `System.IFormattable` or `System.FormattableString`, the meaning is a call to `System.Runtime.CompilerServices.FormattableStringFactory.Create`. If the type is `string`, the meaning of the expression is a call to `string.Format`. In both cases, the argument list of the call consists of a format string literal with placeholders for each interpolation, and an argument for each expression corresponding to the place holders.

The format string literal is constructed as follows, where `N` is the number of interpolations in the *interpolated_string_expression*:

- If an *interpolated_regular_string_whole* or an *interpolated_verbatim_string_whole* follows the `$` sign, then the format string literal is that token.
- Otherwise, the format string literal consists of:
  - First the *interpolated_regular_string_start* or *interpolated_verbatim_string_start*
  - Then for each number `I` from `0` to `N-1`:
    - The decimal representation of `I`
    - Then, if the corresponding *interpolation* has a *constant_expression*, a `,` (comma) followed by the decimal representation of the value of the *constant_expression*
    - Then the *interpolated_regular_string_mid*, *interpolated_regular_string_end*, *interpolated_verbatim_string_mid* or *interpolated_verbatim_string_end* immediately following the corresponding interpolation.

The subsequent arguments are simply the *expressions* from the *interpolations* (if any), in order.

TODO: examples.

Simple names

A *simple_name* consists of an identifier, optionally followed by a type argument list:

```
simple_name
    : identifier type_argument_list?
    ;
```

A *simple_name* is either of the form `I` or of the form `I<A1,...,Ak>`, where `I` is a single identifier and `<A1,...,Ak>` is an optional *type_argument_list*. When no *type_argument_list* is specified, consider `K` to be zero. The *simple_name* is evaluated and classified as follows:

- If `K` is zero and the *simple_name* appears within a *block* and if the *block*'s (or an enclosing *block*'s) local variable declaration space (Declarations) contains a local variable, parameter or constant with name `I`, then the *simple_name* refers to that local variable, parameter or constant and is classified as a variable or value.
- If `K` is zero and the *simple_name* appears within the body of a generic method declaration and if that declaration includes a type parameter with name `I`, then the *simple_name* refers to that type parameter.
- Otherwise, for each instance type `T` (The instance type), starting with the instance type of the immediately enclosing type declaration and continuing with the instance type of each enclosing class or struct declaration (if any):
  - If `K` is zero and the declaration of `T` includes a type parameter with name `I`, then the *simple_name* refers to that type parameter.
  - Otherwise, if a member lookup (Member lookup) of `I` in `T` with `K` type arguments produces a match:
    - If `T` is the instance type of the immediately enclosing class or struct type and the lookup identifies one or more methods, the result is a method group with an associated instance expression of `this`. If a type argument list was specified, it is used in calling a generic method (Method invocations).
    - Otherwise, if `T` is the instance type of the immediately enclosing class or struct type, if the lookup identifies an instance member, and if the reference occurs within the body of an instance constructor, an instance method, or an instance accessor, the result is the same as a member access (Member access) of the form `this.I`. This can only happen when `K` is zero.
    - Otherwise, the result is the same as a member access (Member access) of the form `T.I` or `T.I<A1,...,Ak>`. In this case, it is a binding-time error for the *simple_name* to refer to an instance member.
- Otherwise, for each namespace `N`, starting with the namespace in which the *simple_name* occurs, continuing with each enclosing namespace (if any), and ending with the global namespace, the following steps are evaluated until an entity is located:
  - If `K` is zero and `I` is the name of a namespace in `N`, then:
    - If the location where the *simple_name* occurs is enclosed by a namespace declaration for `N` and the namespace declaration contains an *extern_alias_directive* or *using_alias_directive* that associates the name `I` with a namespace or type, then the *simple_name* is ambiguous and a compile-time error occurs.
    - Otherwise, the *simple_name* refers to the namespace named `I` in `N`.
  - Otherwise, if `N` contains an accessible type having name `I` and `K` type parameters, then:
    - If `K` is zero and the location where the *simple_name* occurs is enclosed by a namespace declaration for `N` and the namespace declaration contains an *extern_alias_directive* or *using_alias_directive* that associates the name `I` with a namespace or type, then the

simple_name is ambiguous and a compile-time error occurs.
    - Otherwise, the *namespace_or_type_name* refers to the type constructed with the given type arguments.
- Otherwise, if the location where the *simple_name* occurs is enclosed by a namespace declaration for `N`:
    - If `K` is zero and the namespace declaration contains an *extern_alias_directive* or *using_alias_directive* that associates the name `I` with an imported namespace or type, then the *simple_name* refers to that namespace or type.
    - Otherwise, if the namespaces and type declarations imported by the *using_namespace_directive*s and *using_static_directive*s of the namespace declaration contain exactly one accessible type or non-extension static member having name `I` and `K` type parameters, then the *simple_name* refers to that type or member constructed with the given type arguments.
    - Otherwise, if the namespaces and types imported by the *using_namespace_directive*s of the namespace declaration contain more than one accessible type or non-extension-method static member having name `I` and `K` type parameters, then the *simple_name* is ambiguous and an error occurs.

  Note that this entire step is exactly parallel to the corresponding step in the processing of a *namespace_or_type_name* (Namespace and type names).

- Otherwise, the *simple_name* is undefined and a compile-time error occurs.

Parenthesized expressions

A *parenthesized_expression* consists of an *expression* enclosed in parentheses.

```
parenthesized_expression
    : '(' expression ')'
    ;
```

A *parenthesized_expression* is evaluated by evaluating the *expression* within the parentheses. If the *expression* within the parentheses denotes a namespace or type, a compile-time error occurs. Otherwise, the result of the *parenthesized_expression* is the result of the evaluation of the contained *expression*.

Member access

A *member_access* consists of a *primary_expression*, a *predefined_type*, or a *qualified_alias_member*, followed by a "`.`" token, followed by an *identifier*, optionally followed by a *type_argument_list*.

```
member_access
    : primary_expression '.' identifier type_argument_list?
    | predefined_type '.' identifier type_argument_list?
    | qualified_alias_member '.' identifier
    ;

predefined_type
    : 'bool'   | 'byte'  | 'char'  | 'decimal' | 'double' | 'float' | 'int' | 'long'
    | 'object' | 'sbyte' | 'short' | 'string'  | 'uint'   | 'ulong' | 'ushort'
    ;
```

The *qualified_alias_member* production is defined in Namespace alias qualifiers.

A *member_access* is either of the form `E.I` or of the form `E.I<A1, ..., Ak>`, where `E` is a primary-expression, `I` is a single identifier and `<A1, ..., Ak>` is an optional *type_argument_list*. When no *type_argument_list* is specified, consider `K` to be zero.

A *member_access* with a *primary_expression* of type `dynamic` is dynamically bound (Dynamic binding). In this case the compiler classifies the member access as a property access of type `dynamic`. The rules below to determine the meaning of the *member_access* are then applied at run-time, using the run-time type instead of the compile-time type of the *primary_expression*. If this run-time classification leads to a method group, then the member access must be the *primary_expression* of an *invocation_expression*.

The *member_access* is evaluated and classified as follows:

- If `K` is zero and `E` is a namespace and `E` contains a nested namespace with name `I`, then the result is that namespace.
- Otherwise, if `E` is a namespace and `E` contains an accessible type having name `I` and `K` type parameters, then the result is that type constructed with the given type arguments.
- If `E` is a *predefined_type* or a *primary_expression* classified as a type, if `E` is not a type parameter, and if a member lookup (Member lookup) of `I` in `E` with `K` type parameters produces a match, then `E.I` is evaluated and classified as follows:
  - If `I` identifies a type, then the result is that type constructed with the given type arguments.
  - If `I` identifies one or more methods, then the result is a method group with no associated instance expression. If a type argument list was specified, it is used in calling a generic method (Method invocations).
  - If `I` identifies a `static` property, then the result is a property access with no associated instance expression.
  - If `I` identifies a `static` field:
    - If the field is `readonly` and the reference occurs outside the static constructor of the class or struct in which the field is declared, then the result is a value, namely the value of the static field `I` in `E`.
    - Otherwise, the result is a variable, namely the static field `I` in `E`.
  - If `I` identifies a `static` event:
    - If the reference occurs within the class or struct in which the event is declared, and the event was declared without *event_accessor_declarations* (Events), then `E.I` is processed exactly as if `I` were a static field.
    - Otherwise, the result is an event access with no associated instance expression.
  - If `I` identifies a constant, then the result is a value, namely the value of that constant.
  - If `I` identifies an enumeration member, then the result is a value, namely the value of that enumeration member.
  - Otherwise, `E.I` is an invalid member reference, and a compile-time error occurs.
- If `E` is a property access, indexer access, variable, or value, the type of which is `T`, and a member lookup (Member lookup) of `I` in `T` with `K` type arguments produces a match, then `E.I` is evaluated and classified as follows:
  - First, if `E` is a property or indexer access, then the value of the property or indexer access is obtained (Values of expressions) and `E` is reclassified as a value.
  - If `I` identifies one or more methods, then the result is a method group with an associated instance expression of `E`. If a type argument list was specified, it is used in calling a generic method (Method invocations).
  - If `I` identifies an instance property,
    - If `E` is `this`, `I` identifies an automatically implemented property (Automatically implemented properties) without a setter, and the reference occurs within an instance constructor for a class or struct type `T`, then the result is a variable, namely the hidden backing field for the auto-property given by `I` in the instance of `T` given by `this`.
    - Otherwise, the result is a property access with an associated instance expression of `E`.

- If `T` is a *class_type* and `I` identifies an instance field of that *class_type*:
  - If the value of `E` is `null`, then a `System.NullReferenceException` is thrown.
  - Otherwise, if the field is `readonly` and the reference occurs outside an instance constructor of the class in which the field is declared, then the result is a value, namely the value of the field `I` in the object referenced by `E`.
  - Otherwise, the result is a variable, namely the field `I` in the object referenced by `E`.
- If `T` is a *struct_type* and `I` identifies an instance field of that *struct_type*:
  - If `E` is a value, or if the field is `readonly` and the reference occurs outside an instance constructor of the struct in which the field is declared, then the result is a value, namely the value of the field `I` in the struct instance given by `E`.
  - Otherwise, the result is a variable, namely the field `I` in the struct instance given by `E`.
- If `I` identifies an instance event:
  - If the reference occurs within the class or struct in which the event is declared, and the event was declared without *event_accessor_declarations* (Events), and the reference does not occur as the left-hand side of a `+=` or `-=` operator, then `E.I` is processed exactly as if `I` was an instance field.
  - Otherwise, the result is an event access with an associated instance expression of `E`.

- Otherwise, an attempt is made to process `E.I` as an extension method invocation (Extension method invocations). If this fails, `E.I` is an invalid member reference, and a binding-time error occurs.

Identical simple names and type names

In a member access of the form `E.I`, if `E` is a single identifier, and if the meaning of `E` as a *simple_name* (Simple names) is a constant, field, property, local variable, or parameter with the same type as the meaning of `E` as a *type_name* (Namespace and type names), then both possible meanings of `E` are permitted. The two possible meanings of `E.I` are never ambiguous, since `I` must necessarily be a member of the type `E` in both cases. In other words, the rule simply permits access to the static members and nested types of `E` where a compile-time error would otherwise have occurred. For example:

```
struct Color
{
    public static readonly Color White = new Color(...);
    public static readonly Color Black = new Color(...);

    public Color Complement() {...}
}

class A
{
    public Color Color;                // Field Color of type Color

    void F() {
        Color = Color.Black;           // References Color.Black static member
        Color = Color.Complement();    // Invokes Complement() on Color field
    }

    static void G() {
        Color c = Color.White;         // References Color.White static member
    }
}
```

Grammar ambiguities

The productions for *simple_name* (Simple names) and *member_access* (Member access) can give rise to ambiguities in the grammar for expressions. For example, the statement:

```
F(G<A,B>(7));
```

could be interpreted as a call to `F` with two arguments, `G < A` and `B > (7)`. Alternatively, it could be interpreted as a call to `F` with one argument, which is a call to a generic method `G` with two type arguments and one regular argument.

If a sequence of tokens can be parsed (in context) as a *simple_name* (Simple names), *member_access* (Member access), or *pointer_member_access* (Pointer member access) ending with a *type_argument_list* (Type arguments), the token immediately following the closing `>` token is examined. If it is one of

```
( ) ] } : ; , . ? == != | ^
```

then the *type_argument_list* is retained as part of the *simple_name*, *member_access* or *pointer_member_access* and any other possible parse of the sequence of tokens is discarded. Otherwise, the *type_argument_list* is not considered to be part of the *simple_name*, *member_access* or *pointer_member_access*, even if there is no other possible parse of the sequence of tokens. Note that these rules are not applied when parsing a *type_argument_list* in a *namespace_or_type_name* (Namespace and type names). The statement

```
F(G<A,B>(7));
```

will, according to this rule, be interpreted as a call to `F` with one argument, which is a call to a generic method `G` with two type arguments and one regular argument. The statements

```
F(G < A, B > 7);
F(G < A, B >> 7);
```

will each be interpreted as a call to `F` with two arguments. The statement

```
x = F < A > +y;
```

will be interpreted as a less than operator, greater than operator, and unary plus operator, as if the statement had been written `x = (F < A) > (+y)`, instead of as a *simple_name* with a *type_argument_list* followed by a binary plus operator. In the statement

```
x = y is C<T> + z;
```

the tokens `C<T>` are interpreted as a *namespace_or_type_name* with a *type_argument_list*.

Invocation expressions

An *invocation_expression* is used to invoke a method.

```
invocation_expression
    : primary_expression '(' argument_list? ')'
    ;
```

An *invocation_expression* is dynamically bound (Dynamic binding) if at least one of the following holds:

- The *primary_expression* has compile-time type `dynamic`.
- At least one argument of the optional *argument_list* has compile-time type `dynamic` and the *primary_expression* does not have a delegate type.

In this case the compiler classifies the *invocation_expression* as a value of type `dynamic`. The rules below to determine the meaning of the *invocation_expression* are then applied at run-time, using the run-time type instead of the compile-time type of those of the *primary_expression* and arguments which have the compile-time type `dynamic`. If the *primary_expression* does not have compile-time type `dynamic`, then the method invocation undergoes a limited compile time check as described in Compile-time checking of dynamic overload resolution.

The *primary_expression* of an *invocation_expression* must be a method group or a value of a *delegate_type*. If the *primary_expression* is a method group, the *invocation_expression* is a method invocation (Method invocations). If the *primary_expression* is a value of a *delegate_type*, the *invocation_expression* is a delegate invocation (Delegate invocations). If the *primary_expression* is neither a method group nor a value of a *delegate_type*, a binding-time error occurs.

The optional *argument_list* (Argument lists) provides values or variable references for the parameters of the method.

The result of evaluating an *invocation_expression* is classified as follows:

- If the *invocation_expression* invokes a method or delegate that returns `void`, the result is nothing. An expression that is classified as nothing is permitted only in the context of a *statement_expression* (Expression statements) or as the body of a *lambda_expression* (Anonymous function expressions). Otherwise a binding-time error occurs.
- Otherwise, the result is a value of the type returned by the method or delegate.

Method invocations

For a method invocation, the *primary_expression* of the *invocation_expression* must be a method group. The method group identifies the one method to invoke or the set of overloaded methods from which to choose a specific method to invoke. In the latter case, determination of the specific method to invoke is based on the context provided by the types of the arguments in the *argument_list*.

The binding-time processing of a method invocation of the form `M(A)`, where `M` is a method group (possibly including a *type_argument_list*), and `A` is an optional *argument_list*, consists of the following steps:

- The set of candidate methods for the method invocation is constructed. For each method `F` associated with the method group `M`:
  - If `F` is non-generic, `F` is a candidate when:
    - `M` has no type argument list, and
    - `F` is applicable with respect to `A` (Applicable function member).
  - If `F` is generic and `M` has no type argument list, `F` is a candidate when:
    - Type inference (Type inference) succeeds, inferring a list of type arguments for the call, and
    - Once the inferred type arguments are substituted for the corresponding method type parameters, all constructed types in the parameter list of F satisfy their constraints (Satisfying constraints), and the parameter list of `F` is applicable with respect to `A` (Applicable function member).
  - If `F` is generic and `M` includes a type argument list, `F` is a candidate when:
    - `F` has the same number of method type parameters as were supplied in the type argument list, and
    - Once the type arguments are substituted for the corresponding method type parameters, all constructed types in the parameter list of F satisfy their constraints (Satisfying constraints), and the parameter list of `F` is applicable with respect to `A` (Applicable function member).
- The set of candidate methods is reduced to contain only methods from the most derived types: For each method `C.F` in the set, where `C` is the type in which the method `F` is declared, all methods declared in a base type of `C` are removed from the set. Furthermore, if `C` is a class type other than `object`, all methods

declared in an interface type are removed from the set. (This latter rule only has affect when the method group was the result of a member lookup on a type parameter having an effective base class other than object and a non-empty effective interface set.)

- If the resulting set of candidate methods is empty, then further processing along the following steps are abandoned, and instead an attempt is made to process the invocation as an extension method invocation (Extension method invocations). If this fails, then no applicable methods exist, and a binding-time error occurs.
- The best method of the set of candidate methods is identified using the overload resolution rules of Overload resolution. If a single best method cannot be identified, the method invocation is ambiguous, and a binding-time error occurs. When performing overload resolution, the parameters of a generic method are considered after substituting the type arguments (supplied or inferred) for the corresponding method type parameters.
- Final validation of the chosen best method is performed:
  - The method is validated in the context of the method group: If the best method is a static method, the method group must have resulted from a *simple_name* or a *member_access* through a type. If the best method is an instance method, the method group must have resulted from a *simple_name*, a *member_access* through a variable or value, or a *base_access*. If neither of these requirements is true, a binding-time error occurs.
  - If the best method is a generic method, the type arguments (supplied or inferred) are checked against the constraints (Satisfying constraints) declared on the generic method. If any type argument does not satisfy the corresponding constraint(s) on the type parameter, a binding-time error occurs.

Once a method has been selected and validated at binding-time by the above steps, the actual run-time invocation is processed according to the rules of function member invocation described in Compile-time checking of dynamic overload resolution.

The intuitive effect of the resolution rules described above is as follows: To locate the particular method invoked by a method invocation, start with the type indicated by the method invocation and proceed up the inheritance chain until at least one applicable, accessible, non-override method declaration is found. Then perform type inference and overload resolution on the set of applicable, accessible, non-override methods declared in that type and invoke the method thus selected. If no method was found, try instead to process the invocation as an extension method invocation.

Extension method invocations

In a method invocation (Invocations on boxed instances) of one of the forms

```
expr . identifier ( )

expr . identifier ( args )

expr . identifier < typeargs > ( )

expr . identifier < typeargs > ( args )
```

if the normal processing of the invocation finds no applicable methods, an attempt is made to process the construct as an extension method invocation. If *expr* or any of the *args* has compile-time type `dynamic`, extension methods will not apply.

The objective is to find the best *type_name* `C`, so that the corresponding static method invocation can take place:

```
C . identifier ( expr )

C . identifier ( expr , args )

C . identifier < typeargs > ( expr )

C . identifier < typeargs > ( expr , args )
```

An extension method `Ci.Mj` is ***eligible*** if:

- `Ci` is a non-generic, non-nested class
- The name of `Mj` is *identifier*
- `Mj` is accessible and applicable when applied to the arguments as a static method as shown above
- An implicit identity, reference or boxing conversion exists from *expr* to the type of the first parameter of `Mj`.

The search for `C` proceeds as follows:

- Starting with the closest enclosing namespace declaration, continuing with each enclosing namespace declaration, and ending with the containing compilation unit, successive attempts are made to find a candidate set of extension methods:
  - If the given namespace or compilation unit directly contains non-generic type declarations `Ci` with eligible extension methods `Mj`, then the set of those extension methods is the candidate set.
  - If types `Ci` imported by *using_static_declarations* and directly declared in namespaces imported by *using_namespace_directive*s in the given namespace or compilation unit directly contain eligible extension methods `Mj`, then the set of those extension methods is the candidate set.
- If no candidate set is found in any enclosing namespace declaration or compilation unit, a compile-time error occurs.
- Otherwise, overload resolution is applied to the candidate set as described in (Overload resolution). If no single best method is found, a compile-time error occurs.
- `C` is the type within which the best method is declared as an extension method.

Using `C` as a target, the method call is then processed as a static method invocation (Compile-time checking of dynamic overload resolution).

The preceding rules mean that instance methods take precedence over extension methods, that extension methods available in inner namespace declarations take precedence over extension methods available in outer namespace declarations, and that extension methods declared directly in a namespace take precedence over extension methods imported into that same namespace with a using namespace directive. For example:

```
public static class E
{
    public static void F(this object obj, int i) { }

    public static void F(this object obj, string s) { }
}

class A { }

class B
{
    public void F(int i) { }
}

class C
{
    public void F(object obj) { }
}

class X
{
    static void Test(A a, B b, C c) {
        a.F(1);             // E.F(object, int)
        a.F("hello");       // E.F(object, string)

        b.F(1);             // B.F(int)
        b.F("hello");       // E.F(object, string)

        c.F(1);             // C.F(object)
        c.F("hello");       // C.F(object)
    }
}
```

In the example, `B`'s method takes precedence over the first extension method, and `C`'s method takes precedence over both extension methods.

```
public static class C
{
    public static void F(this int i) { Console.WriteLine("C.F({0})", i); }
    public static void G(this int i) { Console.WriteLine("C.G({0})", i); }
    public static void H(this int i) { Console.WriteLine("C.H({0})", i); }
}

namespace N1
{
    public static class D
    {
        public static void F(this int i) { Console.WriteLine("D.F({0})", i); }
        public static void G(this int i) { Console.WriteLine("D.G({0})", i); }
    }
}

namespace N2
{
    using N1;

    public static class E
    {
        public static void F(this int i) { Console.WriteLine("E.F({0})", i); }
    }

    class Test
    {
        static void Main(string[] args)
        {
            1.F();
            2.G();
            3.H();
        }
    }
}
```

The output of this example is:

```
E.F(1)
D.G(2)
C.H(3)
```

`D.G` takes precedence over `C.G`, and `E.F` takes precedence over both `D.F` and `C.F`.

Delegate invocations

For a delegate invocation, the *primary_expression* of the *invocation_expression* must be a value of a *delegate_type*. Furthermore, considering the *delegate_type* to be a function member with the same parameter list as the *delegate_type*, the *delegate_type* must be applicable (Applicable function member) with respect to the *argument_list* of the *invocation_expression*.

The run-time processing of a delegate invocation of the form `D(A)`, where `D` is a *primary_expression* of a *delegate_type* and `A` is an optional *argument_list*, consists of the following steps:

- `D` is evaluated. If this evaluation causes an exception, no further steps are executed.
- The value of `D` is checked to be valid. If the value of `D` is `null`, a `System.NullReferenceException` is thrown and no further steps are executed.
- Otherwise, `D` is a reference to a delegate instance. Function member invocations (Compile-time checking of dynamic overload resolution) are performed on each of the callable entities in the invocation list of the delegate. For callable entities consisting of an instance and instance method, the instance for the invocation is the instance contained in the callable entity.

Element access

An *element_access* consists of a *primary_no_array_creation_expression*, followed by a "`[`" token, followed by an *argument_list*, followed by a "`]`" token. The *argument_list* consists of one or more *arguments*, separated by commas.

```
element_access
    : primary_no_array_creation_expression '[' expression_list ']'
    ;
```

The *argument_list* of an *element_access* is not allowed to contain `ref` or `out` arguments.

An *element_access* is dynamically bound (Dynamic binding) if at least one of the following holds:

- The *primary_no_array_creation_expression* has compile-time type `dynamic`.
- At least one expression of the *argument_list* has compile-time type `dynamic` and the *primary_no_array_creation_expression* does not have an array type.

In this case the compiler classifies the *element_access* as a value of type `dynamic`. The rules below to determine the meaning of the *element_access* are then applied at run-time, using the run-time type instead of the compile-time type of those of the *primary_no_array_creation_expression* and *argument_list* expressions which have the compile-time type `dynamic`. If the *primary_no_array_creation_expression* does not have compile-time type `dynamic`, then the element access undergoes a limited compile time check as described in Compile-time checking of dynamic overload resolution.

If the *primary_no_array_creation_expression* of an *element_access* is a value of an *array_type*, the *element_access* is an array access (Array access). Otherwise, the *primary_no_array_creation_expression* must be a variable or value of a class, struct, or interface type that has one or more indexer members, in which case the *element_access* is an indexer access (Indexer access).

Array access

For an array access, the *primary_no_array_creation_expression* of the *element_access* must be a value of an *array_type*. Furthermore, the *argument_list* of an array access is not allowed to contain named arguments.The number of expressions in the *argument_list* must be the same as the rank of the *array_type*, and each expression must be of type `int`, `uint`, `long`, `ulong`, or must be implicitly convertible to one or more of these types.

The result of evaluating an array access is a variable of the element type of the array, namely the array element selected by the value(s) of the expression(s) in the *argument_list*.

The run-time processing of an array access of the form `P[A]`, where `P` is a *primary_no_array_creation_expression* of an *array_type* and `A` is an *argument_list*, consists of the following steps:

- `P` is evaluated. If this evaluation causes an exception, no further steps are executed.
- The index expressions of the *argument_list* are evaluated in order, from left to right. Following evaluation of each index expression, an implicit conversion (Implicit conversions) to one of the following types is performed: `int`, `uint`, `long`, `ulong`. The first type in this list for which an implicit conversion exists is chosen. For instance, if the index expression is of type `short` then an implicit conversion to `int` is performed, since implicit conversions from `short` to `int` and from `short` to `long` are possible. If evaluation of an index expression or the subsequent implicit conversion causes an exception, then no further index expressions are evaluated and no further steps are executed.
- The value of `P` is checked to be valid. If the value of `P` is `null`, a `System.NullReferenceException` is thrown and no further steps are executed.
- The value of each expression in the *argument_list* is checked against the actual bounds of each dimension of the array instance referenced by `P`. If one or more values are out of range, a

`System.IndexOutOfRangeException` is thrown and no further steps are executed.

- The location of the array element given by the index expression(s) is computed, and this location becomes the result of the array access.

Indexer access

For an indexer access, the *primary_no_array_creation_expression* of the *element_access* must be a variable or value of a class, struct, or interface type, and this type must implement one or more indexers that are applicable with respect to the *argument_list* of the *element_access*.

The binding-time processing of an indexer access of the form `P[A]`, where `P` is a *primary_no_array_creation_expression* of a class, struct, or interface type `T`, and `A` is an *argument_list*, consists of the following steps:

- The set of indexers provided by `T` is constructed. The set consists of all indexers declared in `T` or a base type of `T` that are not `override` declarations and are accessible in the current context (Member access).
- The set is reduced to those indexers that are applicable and not hidden by other indexers. The following rules are applied to each indexer `S.I` in the set, where `S` is the type in which the indexer `I` is declared:
  - If `I` is not applicable with respect to `A` (Applicable function member), then `I` is removed from the set.
  - If `I` is applicable with respect to `A` (Applicable function member), then all indexers declared in a base type of `S` are removed from the set.
  - If `I` is applicable with respect to `A` (Applicable function member) and `S` is a class type other than `object`, all indexers declared in an interface are removed from the set.
- If the resulting set of candidate indexers is empty, then no applicable indexers exist, and a binding-time error occurs.
- The best indexer of the set of candidate indexers is identified using the overload resolution rules of Overload resolution. If a single best indexer cannot be identified, the indexer access is ambiguous, and a binding-time error occurs.
- The index expressions of the *argument_list* are evaluated in order, from left to right. The result of processing the indexer access is an expression classified as an indexer access. The indexer access expression references the indexer determined in the step above, and has an associated instance expression of `P` and an associated argument list of `A`.

Depending on the context in which it is used, an indexer access causes invocation of either the *get accessor* or the *set accessor* of the indexer. If the indexer access is the target of an assignment, the *set accessor* is invoked to assign a new value (Simple assignment). In all other cases, the *get accessor* is invoked to obtain the current value (Values of expressions).

This access

A *this_access* consists of the reserved word `this`.

```
this_access
    : 'this'
    ;
```

A *this_access* is permitted only in the *block* of an instance constructor, an instance method, or an instance accessor. It has one of the following meanings:

- When `this` is used in a *primary_expression* within an instance constructor of a class, it is classified as a value. The type of the value is the instance type (The instance type) of the class within which the usage occurs, and the value is a reference to the object being constructed.
- When `this` is used in a *primary_expression* within an instance method or instance accessor of a class, it is classified as a value. The type of the value is the instance type (The instance type) of the class within which

the usage occurs, and the value is a reference to the object for which the method or accessor was invoked.

- When `this` is used in a *primary_expression* within an instance constructor of a struct, it is classified as a variable. The type of the variable is the instance type (The instance type) of the struct within which the usage occurs, and the variable represents the struct being constructed. The `this` variable of an instance constructor of a struct behaves exactly the same as an `out` parameter of the struct type—in particular, this means that the variable must be definitely assigned in every execution path of the instance constructor.
- When `this` is used in a *primary_expression* within an instance method or instance accessor of a struct, it is classified as a variable. The type of the variable is the instance type (The instance type) of the struct within which the usage occurs.
  - If the method or accessor is not an iterator (Iterators), the `this` variable represents the struct for which the method or accessor was invoked, and behaves exactly the same as a `ref` parameter of the struct type.
  - If the method or accessor is an iterator, the `this` variable represents a copy of the struct for which the method or accessor was invoked, and behaves exactly the same as a value parameter of the struct type.

Use of `this` in a *primary_expression* in a context other than the ones listed above is a compile-time error. In particular, it is not possible to refer to `this` in a static method, a static property accessor, or in a *variable_initializer* of a field declaration.

Base access

A *base_access* consists of the reserved word `base` followed by either a "`.`" token and an identifier or an *argument_list* enclosed in square brackets:

```
base_access
    : 'base' '.' identifier
    | 'base' '[' expression_list ']'
    ;
```

A *base_access* is used to access base class members that are hidden by similarly named members in the current class or struct. A *base_access* is permitted only in the *block* of an instance constructor, an instance method, or an instance accessor. When `base.I` occurs in a class or struct, `I` must denote a member of the base class of that class or struct. Likewise, when `base[E]` occurs in a class, an applicable indexer must exist in the base class.

At binding-time, *base_access* expressions of the form `base.I` and `base[E]` are evaluated exactly as if they were written `((B)this).I` and `((B)this)[E]`, where `B` is the base class of the class or struct in which the construct occurs. Thus, `base.I` and `base[E]` correspond to `this.I` and `this[E]`, except `this` is viewed as an instance of the base class.

When a *base_access* references a virtual function member (a method, property, or indexer), the determination of which function member to invoke at run-time (Compile-time checking of dynamic overload resolution) is changed. The function member that is invoked is determined by finding the most derived implementation (Virtual methods) of the function member with respect to `B` (instead of with respect to the run-time type of `this`, as would be usual in a non-base access). Thus, within an `override` of a `virtual` function member, a *base_access* can be used to invoke the inherited implementation of the function member. If the function member referenced by a *base_access* is abstract, a binding-time error occurs.

Postfix increment and decrement operators

```
post_increment_expression
    : primary_expression '++'
    ;

post_decrement_expression
    : primary_expression '--'
    ;
```

The operand of a postfix increment or decrement operation must be an expression classified as a variable, a property access, or an indexer access. The result of the operation is a value of the same type as the operand.

If the *primary_expression* has the compile-time type `dynamic` then the operator is dynamically bound (Dynamic binding), the *post_increment_expression* or *post_decrement_expression* has the compile-time type `dynamic` and the following rules are applied at run-time using the run-time type of the *primary_expression*.

If the operand of a postfix increment or decrement operation is a property or indexer access, the property or indexer must have both a `get` and a `set` accessor. If this is not the case, a binding-time error occurs.

Unary operator overload resolution (Unary operator overload resolution) is applied to select a specific operator implementation. Predefined `++` and `--` operators exist for the following types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, and any enum type. The predefined `++` operators return the value produced by adding 1 to the operand, and the predefined `--` operators return the value produced by subtracting 1 from the operand. In a `checked` context, if the result of this addition or subtraction is outside the range of the result type and the result type is an integral type or enum type, a `System.OverflowException` is thrown.

The run-time processing of a postfix increment or decrement operation of the form `x++` or `x--` consists of the following steps:

- If `x` is classified as a variable:
  - `x` is evaluated to produce the variable.
  - The value of `x` is saved.
  - The selected operator is invoked with the saved value of `x` as its argument.
  - The value returned by the operator is stored in the location given by the evaluation of `x`.
  - The saved value of `x` becomes the result of the operation.
- If `x` is classified as a property or indexer access:
  - The instance expression (if `x` is not `static`) and the argument list (if `x` is an indexer access) associated with `x` are evaluated, and the results are used in the subsequent `get` and `set` accessor invocations.
  - The `get` accessor of `x` is invoked and the returned value is saved.
  - The selected operator is invoked with the saved value of `x` as its argument.
  - The `set` accessor of `x` is invoked with the value returned by the operator as its `value` argument.
  - The saved value of `x` becomes the result of the operation.

The `++` and `--` operators also support prefix notation (Prefix increment and decrement operators). Typically, the result of `x++` or `x--` is the value of `x` before the operation, whereas the result of `++x` or `--x` is the value of `x` after the operation. In either case, `x` itself has the same value after the operation.

An `operator ++` or `operator --` implementation can be invoked using either postfix or prefix notation. It is not possible to have separate operator implementations for the two notations.

The new operator

The `new` operator is used to create new instances of types.

There are three forms of `new` expressions:

- Object creation expressions are used to create new instances of class types and value types.
- Array creation expressions are used to create new instances of array types.
- Delegate creation expressions are used to create new instances of delegate types.

The `new` operator implies creation of an instance of a type, but does not necessarily imply dynamic allocation of memory. In particular, instances of value types require no additional memory beyond the variables in which they reside, and no dynamic allocations occur when `new` is used to create instances of value types.

Object creation expressions

An *object_creation_expression* is used to create a new instance of a *class_type* or a *value_type*.

```
object_creation_expression
    : 'new' type '(' argument_list? ')' object_or_collection_initializer?
    | 'new' type object_or_collection_initializer
    ;

object_or_collection_initializer
    : object_initializer
    | collection_initializer
    ;
```

The *type* of an *object_creation_expression* must be a *class_type*, a *value_type* or a *type_parameter*. The *type* cannot be an `abstract` *class_type*.

The optional *argument_list* (Argument lists) is permitted only if the *type* is a *class_type* or a *struct_type*.

An object creation expression can omit the constructor argument list and enclosing parentheses provided it includes an object initializer or collection initializer. Omitting the constructor argument list and enclosing parentheses is equivalent to specifying an empty argument list.

Processing of an object creation expression that includes an object initializer or collection initializer consists of first processing the instance constructor and then processing the member or element initializations specified by the object initializer (Object initializers) or collection initializer (Collection initializers).

If any of the arguments in the optional *argument_list* has the compile-time type `dynamic` then the *object_creation_expression* is dynamically bound (Dynamic binding) and the following rules are applied at run-time using the run-time type of those arguments of the *argument_list* that have the compile time type `dynamic`. However, the object creation undergoes a limited compile time check as described in Compile-time checking of dynamic overload resolution.

The binding-time processing of an *object_creation_expression* of the form `new T(A)`, where `T` is a *class_type* or a *value_type* and `A` is an optional *argument_list*, consists of the following steps:

- If `T` is a *value_type* and `A` is not present:
  - The *object_creation_expression* is a default constructor invocation. The result of the *object_creation_expression* is a value of type `T`, namely the default value for `T` as defined in The System.ValueType type.
- Otherwise, if `T` is a *type_parameter* and `A` is not present:
  - If no value type constraint or constructor constraint (Type parameter constraints) has been specified for `T`, a binding-time error occurs.
  - The result of the *object_creation_expression* is a value of the run-time type that the type parameter has been bound to, namely the result of invoking the default constructor of that type. The run-time type may be a reference type or a value type.
- Otherwise, if `T` is a *class_type* or a *struct_type*:

- If `T` is an `abstract` *class_type*, a compile-time error occurs.
- The instance constructor to invoke is determined using the overload resolution rules of Overload resolution. The set of candidate instance constructors consists of all accessible instance constructors declared in `T` which are applicable with respect to `A` (Applicable function member). If the set of candidate instance constructors is empty, or if a single best instance constructor cannot be identified, a binding-time error occurs.
- The result of the *object_creation_expression* is a value of type `T`, namely the value produced by invoking the instance constructor determined in the step above.

- Otherwise, the *object_creation_expression* is invalid, and a binding-time error occurs.

Even if the *object_creation_expression* is dynamically bound, the compile-time type is still `T`.

The run-time processing of an *object_creation_expression* of the form `new T(A)`, where `T` is *class_type* or a *struct_type* and `A` is an optional *argument_list*, consists of the following steps:

- If `T` is a *class_type*:
  - A new instance of class `T` is allocated. If there is not enough memory available to allocate the new instance, a `System.OutOfMemoryException` is thrown and no further steps are executed.
  - All fields of the new instance are initialized to their default values (Default values).
  - The instance constructor is invoked according to the rules of function member invocation (Compile-time checking of dynamic overload resolution). A reference to the newly allocated instance is automatically passed to the instance constructor and the instance can be accessed from within that constructor as `this`.
- If `T` is a *struct_type*:
  - An instance of type `T` is created by allocating a temporary local variable. Since an instance constructor of a *struct_type* is required to definitely assign a value to each field of the instance being created, no initialization of the temporary variable is necessary.
  - The instance constructor is invoked according to the rules of function member invocation (Compile-time checking of dynamic overload resolution). A reference to the newly allocated instance is automatically passed to the instance constructor and the instance can be accessed from within that constructor as `this`.

Object initializers

An ***object initializer*** specifies values for zero or more fields, properties or indexed elements of an object.

```
object_initializer
    : '{' member_initializer_list? '}'
    | '{' member_initializer_list ',' '}'
    ;

member_initializer_list
    : member_initializer (',' member_initializer)*
    ;

member_initializer
    : initializer_target '=' initializer_value
    ;

initializer_target
    : identifier
    | '[' argument_list ']'
    ;

initializer_value
    : expression
    | object_or_collection_initializer
    ;
```

An object initializer consists of a sequence of member initializers, enclosed by `{` and `}` tokens and separated by commas. Each *member_initializer* designates a target for the initialization. An *identifier* must name an accessible field or property of the object being initialized, whereas an *argument_list* enclosed in square brackets must specify arguments for an accessible indexer on the object being initialized. It is an error for an object initializer to include more than one member initializer for the same field or property.

Each *initializer_target* is followed by an equals sign and either an expression, an object initializer or a collection initializer. It is not possible for expressions within the object initializer to refer to the newly created object it is initializing.

A member initializer that specifies an expression after the equals sign is processed in the same way as an assignment (Simple assignment) to the target.

A member initializer that specifies an object initializer after the equals sign is a ***nested object initializer***, i.e. an initialization of an embedded object. Instead of assigning a new value to the field or property, the assignments in the nested object initializer are treated as assignments to members of the field or property. Nested object initializers cannot be applied to properties with a value type, or to read-only fields with a value type.

A member initializer that specifies a collection initializer after the equals sign is an initialization of an embedded collection. Instead of assigning a new collection to the target field, property or indexer, the elements given in the initializer are added to the collection referenced by the target. The target must be of a collection type that satisfies the requirements specified in Collection initializers.

The arguments to an index initializer will always be evaluated exactly once. Thus, even if the arguments end up never getting used (e.g. because of an empty nested initializer), they will be evaluated for their side effects.

The following class represents a point with two coordinates:

```
public class Point
{
    int x, y;

    public int X { get { return x; } set { x = value; } }
    public int Y { get { return y; } set { y = value; } }
}
```

An instance of `Point` can be created and initialized as follows:

```
Point a = new Point { X = 0, Y = 1 };
```

which has the same effect as

```
Point __a = new Point();
__a.X = 0;
__a.Y = 1;
Point a = __a;
```

where `__a` is an otherwise invisible and inaccessible temporary variable. The following class represents a rectangle created from two points:

```
public class Rectangle
{
    Point p1, p2;

    public Point P1 { get { return p1; } set { p1 = value; } }
    public Point P2 { get { return p2; } set { p2 = value; } }
}
```

An instance of `Rectangle` can be created and initialized as follows:

```
Rectangle r = new Rectangle {
    P1 = new Point { X = 0, Y = 1 },
    P2 = new Point { X = 2, Y = 3 }
};
```

which has the same effect as

```
Rectangle __r = new Rectangle();
Point __p1 = new Point();
__p1.X = 0;
__p1.Y = 1;
__r.P1 = __p1;
Point __p2 = new Point();
__p2.X = 2;
__p2.Y = 3;
__r.P2 = __p2;
Rectangle r = __r;
```

where `__r`, `__p1` and `__p2` are temporary variables that are otherwise invisible and inaccessible.

If `Rectangle`'s constructor allocates the two embedded `Point` instances

```
public class Rectangle
{
    Point p1 = new Point();
    Point p2 = new Point();

    public Point P1 { get { return p1; } }
    public Point P2 { get { return p2; } }
}
```

the following construct can be used to initialize the embedded `Point` instances instead of assigning new instances:

```
Rectangle r = new Rectangle {
    P1 = { X = 0, Y = 1 },
    P2 = { X = 2, Y = 3 }
};
```

which has the same effect as

```
Rectangle __r = new Rectangle();
__r.P1.X = 0;
__r.P1.Y = 1;
__r.P2.X = 2;
__r.P2.Y = 3;
Rectangle r = __r;
```

Given an appropriate definition of C, the following example:

```
var c = new C {
    x = true,
    y = { a = "Hello" },
    z = { 1, 2, 3 },
    ["x"] = 5,
    [0,0] = { "a", "b" },
    [1,2] = {}
};
```

is equivalent to this series of assignments:

```
C __c = new C();
__c.x = true;
__c.y.a = "Hello";
__c.z.Add(1);
__c.z.Add(2);
__c.z.Add(3);
string __i1 = "x";
__c[__i1] = 5;
int __i2 = 0, __i3 = 0;
__c[__i2,__i3].Add("a");
__c[__i2,__i3].Add("b");
int __i4 = 1, __i5 = 2;
var c = __c;
```

where `__c`, etc., are generated variables that are invisible and inaccessible to the source code. Note that the arguments for `[0,0]` are evaluated only once, and the arguments for `[1,2]` are evaluated once even though they are never used.

Collection initializers

A collection initializer specifies the elements of a collection.

```
collection_initializer
    : '{' element_initializer_list '}'
    | '{' element_initializer_list ',' '}'
    ;

element_initializer_list
    : element_initializer (',' element_initializer)*
    ;

element_initializer
    : non_assignment_expression
    | '{' expression_list '}'
    ;

expression_list
    : expression (',' expression)*
    ;
```

A collection initializer consists of a sequence of element initializers, enclosed by `{` and `}` tokens and separated by commas. Each element initializer specifies an element to be added to the collection object being initialized, and consists of a list of expressions enclosed by `{` and `}` tokens and separated by commas. A single-expression element initializer can be written without braces, but cannot then be an assignment expression, to avoid ambiguity with member initializers. The *non_assignment_expression* production is defined in Expression.

The following is an example of an object creation expression that includes a collection initializer:

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

The collection object to which a collection initializer is applied must be of a type that implements `System.Collections.IEnumerable` or a compile-time error occurs. For each specified element in order, the collection initializer invokes an `Add` method on the target object with the expression list of the element initializer as argument list, applying normal member lookup and overload resolution for each invocation. Thus, the collection object must have an applicable instance or extension method with the name `Add` for each element initializer.

The following class represents a contact with a name and a list of phone numbers:

```
public class Contact
{
    string name;
    List<string> phoneNumbers = new List<string>();

    public string Name { get { return name; } set { name = value; } }

    public List<string> PhoneNumbers { get { return phoneNumbers; } }
}
```

A `List<Contact>` can be created and initialized as follows:

```
var contacts = new List<Contact> {
    new Contact {
        Name = "Chris Smith",
        PhoneNumbers = { "206-555-0101", "425-882-8080" }
    },
    new Contact {
        Name = "Bob Harris",
        PhoneNumbers = { "650-555-0199" }
    }
};
```

which has the same effect as

```
var __clist = new List<Contact>();
Contact __c1 = new Contact();
__c1.Name = "Chris Smith";
__c1.PhoneNumbers.Add("206-555-0101");
__c1.PhoneNumbers.Add("425-882-8080");
__clist.Add(__c1);
Contact __c2 = new Contact();
__c2.Name = "Bob Harris";
__c2.PhoneNumbers.Add("650-555-0199");
__clist.Add(__c2);
var contacts = __clist;
```

where `__clist`, `__c1` and `__c2` are temporary variables that are otherwise invisible and inaccessible.

Array creation expressions

An *array_creation_expression* is used to create a new instance of an *array_type*.

```
array_creation_expression
    : 'new' non_array_type '[' expression_list ']' rank_specifier* array_initializer?
    | 'new' array_type array_initializer
    | 'new' rank_specifier array_initializer
    ;
```

An array creation expression of the first form allocates an array instance of the type that results from deleting each of the individual expressions from the expression list. For example, the array creation expression `new int[10,20]` produces an array instance of type `int[,]`, and the array creation expression `new int[10][,]` produces an array of type `int[][,]`. Each expression in the expression list must be of type `int`, `uint`, `long`, or `ulong`, or implicitly convertible to one or more of these types. The value of each expression determines the length of the corresponding dimension in the newly allocated array instance. Since the length of an array dimension must be nonnegative, it is a compile-time error to have a *constant_expression* with a negative value in the expression list.

Except in an unsafe context (Unsafe contexts), the layout of arrays is unspecified.

If an array creation expression of the first form includes an array initializer, each expression in the expression list must be a constant and the rank and dimension lengths specified by the expression list must match those of the array initializer.

In an array creation expression of the second or third form, the rank of the specified array type or rank specifier must match that of the array initializer. The individual dimension lengths are inferred from the number of elements in each of the corresponding nesting levels of the array initializer. Thus, the expression

```
new int[,] {{0, 1}, {2, 3}, {4, 5}}
```

exactly corresponds to

```
new int[3, 2] {{0, 1}, {2, 3}, {4, 5}}
```

An array creation expression of the third form is referred to as an ***implicitly typed array creation expression***. It is similar to the second form, except that the element type of the array is not explicitly given, but determined as the best common type (Finding the best common type of a set of expressions) of the set of expressions in the array initializer. For a multidimensional array, i.e., one where the *rank_specifier* contains at least one comma, this set comprises all *expression*s found in nested *array_initializer*s.

Array initializers are described further in Array initializers.

The result of evaluating an array creation expression is classified as a value, namely a reference to the newly allocated array instance. The run-time processing of an array creation expression consists of the following steps:

- The dimension length expressions of the *expression_list* are evaluated in order, from left to right. Following evaluation of each expression, an implicit conversion (Implicit conversions) to one of the following types is performed: `int`, `uint`, `long`, `ulong`. The first type in this list for which an implicit conversion exists is chosen. If evaluation of an expression or the subsequent implicit conversion causes an exception, then no further expressions are evaluated and no further steps are executed.
- The computed values for the dimension lengths are validated as follows. If one or more of the values are less than zero, a `System.OverflowException` is thrown and no further steps are executed.
- An array instance with the given dimension lengths is allocated. If there is not enough memory available to allocate the new instance, a `System.OutOfMemoryException` is thrown and no further steps are executed.
- All elements of the new array instance are initialized to their default values (Default values).
- If the array creation expression contains an array initializer, then each expression in the array initializer is evaluated and assigned to its corresponding array element. The evaluations and assignments are performed in the order the expressions are written in the array initializer—in other words, elements are initialized in increasing index order, with the rightmost dimension increasing first. If evaluation of a given expression or the subsequent assignment to the corresponding array element causes an exception, then no further elements are initialized (and the remaining elements will thus have their default values).

An array creation expression permits instantiation of an array with elements of an array type, but the elements of such an array must be manually initialized. For example, the statement

```
int[][] a = new int[100][];
```

creates a single-dimensional array with 100 elements of type `int[]`. The initial value of each element is `null`. It is not possible for the same array creation expression to also instantiate the sub-arrays, and the statement

```
int[][] a = new int[100][5];        // Error
```

results in a compile-time error. Instantiation of the sub-arrays must instead be performed manually, as in

```
int[][] a = new int[100][];
for (int i = 0; i < 100; i++) a[i] = new int[5];
```

When an array of arrays has a "rectangular" shape, that is when the sub-arrays are all of the same length, it is more efficient to use a multi-dimensional array. In the example above, instantiation of the array of arrays creates 101 objects—one outer array and 100 sub-arrays. In contrast,

```
int[,] = new int[100, 5];
```

creates only a single object, a two-dimensional array, and accomplishes the allocation in a single statement.

The following are examples of implicitly typed array creation expressions:

```
var a = new[] { 1, 10, 100, 1000 };                     // int[]

var b = new[] { 1, 1.5, 2, 2.5 };                       // double[]

var c = new[,] { { "hello", null }, { "world", "!" } }; // string[,]

var d = new[] { 1, "one", 2, "two" };                   // Error
```

The last expression causes a compile-time error because neither `int` nor `string` is implicitly convertible to the other, and so there is no best common type. An explicitly typed array creation expression must be used in this case, for example specifying the type to be `object[]`. Alternatively, one of the elements can be cast to a common base type, which would then become the inferred element type.

Implicitly typed array creation expressions can be combined with anonymous object initializers (Anonymous object creation expressions) to create anonymously typed data structures. For example:

```
var contacts = new[] {
    new {
        Name = "Chris Smith",
        PhoneNumbers = new[] { "206-555-0101", "425-882-8080" }
    },
    new {
        Name = "Bob Harris",
        PhoneNumbers = new[] { "650-555-0199" }
    }
};
```

Delegate creation expressions

A *delegate_creation_expression* is used to create a new instance of a *delegate_type*.

```
delegate_creation_expression
    : 'new' delegate_type '(' expression ')'
    ;
```

The argument of a delegate creation expression must be a method group, an anonymous function or a value of either the compile time type `dynamic` or a *delegate_type*. If the argument is a method group, it identifies the method and, for an instance method, the object for which to create a delegate. If the argument is an anonymous function it directly defines the parameters and method body of the delegate target. If the argument is a value it identifies a delegate instance of which to create a copy.

If the *expression* has the compile-time type `dynamic`, the *delegate_creation_expression* is dynamically bound (Dynamic binding), and the rules below are applied at run-time using the run-time type of the *expression*. Otherwise the rules are applied at compile-time.

The binding-time processing of a *delegate_creation_expression* of the form `new D(E)`, where `D` is a *delegate_type* and `E` is an *expression*, consists of the following steps:

- If `E` is a method group, the delegate creation expression is processed in the same way as a method group conversion (Method group conversions) from `E` to `D`.
- If `E` is an anonymous function, the delegate creation expression is processed in the same way as an

anonymous function conversion (Anonymous function conversions) from `E` to `D`.

- If `E` is a value, `E` must be compatible (Delegate declarations) with `D`, and the result is a reference to a newly created delegate of type `D` that refers to the same invocation list as `E`. If `E` is not compatible with `D`, a compile-time error occurs.

The run-time processing of a *delegate_creation_expression* of the form `new D(E)`, where `D` is a *delegate_type* and `E` is an *expression*, consists of the following steps:

- If `E` is a method group, the delegate creation expression is evaluated as a method group conversion (Method group conversions) from `E` to `D`.
- If `E` is an anonymous function, the delegate creation is evaluated as an anonymous function conversion from `E` to `D` (Anonymous function conversions).
- If `E` is a value of a *delegate_type*:
  - `E` is evaluated. If this evaluation causes an exception, no further steps are executed.
  - If the value of `E` is `null`, a `System.NullReferenceException` is thrown and no further steps are executed.
  - A new instance of the delegate type `D` is allocated. If there is not enough memory available to allocate the new instance, a `System.OutOfMemoryException` is thrown and no further steps are executed.
  - The new delegate instance is initialized with the same invocation list as the delegate instance given by `E`.

The invocation list of a delegate is determined when the delegate is instantiated and then remains constant for the entire lifetime of the delegate. In other words, it is not possible to change the target callable entities of a delegate once it has been created. When two delegates are combined or one is removed from another (Delegate declarations), a new delegate results; no existing delegate has its contents changed.

It is not possible to create a delegate that refers to a property, indexer, user-defined operator, instance constructor, destructor, or static constructor.

As described above, when a delegate is created from a method group, the formal parameter list and return type of the delegate determine which of the overloaded methods to select. In the example

```
delegate double DoubleFunc(double x);

class A
{
    DoubleFunc f = new DoubleFunc(Square);

    static float Square(float x) {
        return x * x;
    }

    static double Square(double x) {
        return x * x;
    }
}
```

the `A.f` field is initialized with a delegate that refers to the second `Square` method because that method exactly matches the formal parameter list and return type of `DoubleFunc`. Had the second `Square` method not been present, a compile-time error would have occurred.

Anonymous object creation expressions

An *anonymous_object_creation_expression* is used to create an object of an anonymous type.

```
anonymous_object_creation_expression
    : 'new' anonymous_object_initializer
    ;

anonymous_object_initializer
    : '{' member_declarator_list? '}'
    | '{' member_declarator_list ',' '}'
    ;

member_declarator_list
    : member_declarator (',' member_declarator)*
    ;

member_declarator
    : simple_name
    | member_access
    | base_access
    | null_conditional_member_access
    | identifier '=' expression
    ;
```

An anonymous object initializer declares an anonymous type and returns an instance of that type. An anonymous type is a nameless class type that inherits directly from `object`. The members of an anonymous type are a sequence of read-only properties inferred from the anonymous object initializer used to create an instance of the type. Specifically, an anonymous object initializer of the form

```
new { p1 = e1, p2 = e2, ..., pn = en }
```

declares an anonymous type of the form

```
class __Anonymous1
{
    private readonly T1 f1;
    private readonly T2 f2;
    ...
    private readonly Tn fn;

    public __Anonymous1(T1 a1, T2 a2, ..., Tn an) {
        f1 = a1;
        f2 = a2;
        ...
        fn = an;
    }

    public T1 p1 { get { return f1; } }
    public T2 p2 { get { return f2; } }
    ...
    public Tn pn { get { return fn; } }

    public override bool Equals(object __o) { ... }
    public override int GetHashCode() { ... }
}
```

where each `Tx` is the type of the corresponding expression `ex`. The expression used in a *member_declarator* must have a type. Thus, it is a compile-time error for an expression in a *member_declarator* to be null or an anonymous function. It is also a compile-time error for the expression to have an unsafe type.

The names of an anonymous type and of the parameter to its `Equals` method are automatically generated by the compiler and cannot be referenced in program text.

Within the same program, two anonymous object initializers that specify a sequence of properties of the same

names and compile-time types in the same order will produce instances of the same anonymous type.

In the example

```
var p1 = new { Name = "Lawnmower", Price = 495.00 };
var p2 = new { Name = "Shovel", Price = 26.95 };
p1 = p2;
```

the assignment on the last line is permitted because `p1` and `p2` are of the same anonymous type.

The `Equals` and `GetHashcode` methods on anonymous types override the methods inherited from `object`, and are defined in terms of the `Equals` and `GetHashcode` of the properties, so that two instances of the same anonymous type are equal if and only if all their properties are equal.

A member declarator can be abbreviated to a simple name (Type inference), a member access (Compile-time checking of dynamic overload resolution), a base access (Base access) or a null-conditional member access (Null-conditional expressions as projection initializers). This is called a ***projection initializer*** and is shorthand for a declaration of and assignment to a property with the same name. Specifically, member declarators of the forms

```
identifier
expr.identifier
```

are precisely equivalent to the following, respectively:

```
identifier = identifier
identifier = expr.identifier
```

Thus, in a projection initializer the *identifier* selects both the value and the field or property to which the value is assigned. Intuitively, a projection initializer projects not just a value, but also the name of the value.

The typeof operator

The `typeof` operator is used to obtain the `System.Type` object for a type.

```
typeof_expression
    : 'typeof' '(' type ')'
    | 'typeof' '(' unbound_type_name ')'
    | 'typeof' '(' 'void' ')'
    ;

unbound_type_name
    : identifier generic_dimension_specifier?
    | identifier '::' identifier generic_dimension_specifier?
    | unbound_type_name '.' identifier generic_dimension_specifier?
    ;

generic_dimension_specifier
    : '<' comma* '>'
    ;

comma
    : ','
    ;
```

The first form of *typeof_expression* consists of a `typeof` keyword followed by a parenthesized *type*. The result of an expression of this form is the `System.Type` object for the indicated type. There is only one `System.Type` object for any given type. This means that for a type `T`, `typeof(T) == typeof(T)` is always true. The *type* cannot

be `dynamic`.

The second form of *typeof_expression* consists of a `typeof` keyword followed by a parenthesized *unbound_type_name*. An *unbound_type_name* is very similar to a *type_name* (Namespace and type names) except that an *unbound_type_name* contains *generic_dimension_specifier*s where a *type_name* contains *type_argument_list*s. When the operand of a *typeof_expression* is a sequence of tokens that satisfies the grammars of both *unbound_type_name* and *type_name*, namely when it contains neither a *generic_dimension_specifier* nor a *type_argument_list*, the sequence of tokens is considered to be a *type_name*. The meaning of an *unbound_type_name* is determined as follows:

- Convert the sequence of tokens to a *type_name* by replacing each *generic_dimension_specifier* with a *type_argument_list* having the same number of commas and the keyword `object` as each *type_argument*.
- Evaluate the resulting *type_name*, while ignoring all type parameter constraints.
- The *unbound_type_name* resolves to the unbound generic type associated with the resulting constructed type (Bound and unbound types).

The result of the *typeof_expression* is the `System.Type` object for the resulting unbound generic type.

The third form of *typeof_expression* consists of a `typeof` keyword followed by a parenthesized `void` keyword. The result of an expression of this form is the `System.Type` object that represents the absence of a type. The type object returned by `typeof(void)` is distinct from the type object returned for any type. This special type object is useful in class libraries that allow reflection onto methods in the language, where those methods wish to have a way to represent the return type of any method, including void methods, with an instance of `System.Type`.

The `typeof` operator can be used on a type parameter. The result is the `System.Type` object for the run-time type that was bound to the type parameter. The `typeof` operator can also be used on a constructed type or an unbound generic type (Bound and unbound types). The `System.Type` object for an unbound generic type is not the same as the `System.Type` object of the instance type. The instance type is always a closed constructed type at run-time so its `System.Type` object depends on the run-time type arguments in use, while the unbound generic type has no type arguments.

The example

```
using System;

class X<T>
{
    public static void PrintTypes() {
        Type[] t = {
            typeof(int),
            typeof(System.Int32),
            typeof(string),
            typeof(double[]),
            typeof(void),
            typeof(T),
            typeof(X<T>),
            typeof(X<X<T>>),
            typeof(X<>)
        };
        for (int i = 0; i < t.Length; i++) {
            Console.WriteLine(t[i]);
        }
    }
}

class Test
{
    static void Main() {
        X<int>.PrintTypes();
    }
}
```

produces the following output:

```
System.Int32
System.Int32
System.String
System.Double[]
System.Void
System.Int32
X`1[System.Int32]
X`1[X`1[System.Int32]]
X`1[T]
```

Note that `int` and `System.Int32` are the same type.

Also note that the result of `typeof(X<>)` does not depend on the type argument but the result of `typeof(X<T>)` does.

The checked and unchecked operators

The `checked` and `unchecked` operators are used to control the ***overflow checking context*** for integral-type arithmetic operations and conversions.

```
checked_expression
    : 'checked' '(' expression ')'
    ;

unchecked_expression
    : 'unchecked' '(' expression ')'
    ;
```

The `checked` operator evaluates the contained expression in a checked context, and the `unchecked` operator evaluates the contained expression in an unchecked context. A *checked_expression* or *unchecked_expression* corresponds exactly to a *parenthesized_expression* (Parenthesized expressions), except that the contained

expression is evaluated in the given overflow checking context.

The overflow checking context can also be controlled through the `checked` and `unchecked` statements (The checked and unchecked statements).

The following operations are affected by the overflow checking context established by the `checked` and `unchecked` operators and statements:

- The predefined `++` and `--` unary operators (Postfix increment and decrement operators and Prefix increment and decrement operators), when the operand is of an integral type.
- The predefined `-` unary operator (Unary minus operator), when the operand is of an integral type.
- The predefined `+`, `-`, `*`, and `/` binary operators (Arithmetic operators), when both operands are of integral types.
- Explicit numeric conversions (Explicit numeric conversions) from one integral type to another integral type, or from `float` or `double` to an integral type.

When one of the above operations produce a result that is too large to represent in the destination type, the context in which the operation is performed controls the resulting behavior:

- In a `checked` context, if the operation is a constant expression (Constant expressions), a compile-time error occurs. Otherwise, when the operation is performed at run-time, a `System.OverflowException` is thrown.
- In an `unchecked` context, the result is truncated by discarding any high-order bits that do not fit in the destination type.

For non-constant expressions (expressions that are evaluated at run-time) that are not enclosed by any `checked` or `unchecked` operators or statements, the default overflow checking context is `unchecked` unless external factors (such as compiler switches and execution environment configuration) call for `checked` evaluation.

For constant expressions (expressions that can be fully evaluated at compile-time), the default overflow checking context is always `checked`. Unless a constant expression is explicitly placed in an `unchecked` context, overflows that occur during the compile-time evaluation of the expression always cause compile-time errors.

The body of an anonymous function is not affected by `checked` or `unchecked` contexts in which the anonymous function occurs.

In the example

```
class Test
{
    static readonly int x = 1000000;
    static readonly int y = 1000000;

    static int F() {
        return checked(x * y);      // Throws OverflowException
    }

    static int G() {
        return unchecked(x * y);    // Returns -727379968
    }

    static int H() {
        return x * y;               // Depends on default
    }
}
```

no compile-time errors are reported since neither of the expressions can be evaluated at compile-time. At run-time, the `F` method throws a `System.OverflowException`, and the `G` method returns -727379968 (the lower 32 bits of the out-of-range result). The behavior of the `H` method depends on the default overflow checking

context for the compilation, but it is either the same as `F` or the same as `G`.

In the example

```
class Test
{
    const int x = 1000000;
    const int y = 1000000;

    static int F() {
        return checked(x * y);      // Compile error, overflow
    }

    static int G() {
        return unchecked(x * y);    // Returns -727379968
    }

    static int H() {
        return x * y;               // Compile error, overflow
    }
}
```

the overflows that occur when evaluating the constant expressions in `F` and `H` cause compile-time errors to be reported because the expressions are evaluated in a `checked` context. An overflow also occurs when evaluating the constant expression in `G`, but since the evaluation takes place in an `unchecked` context, the overflow is not reported.

The `checked` and `unchecked` operators only affect the overflow checking context for those operations that are textually contained within the "`(`" and "`)`" tokens. The operators have no effect on function members that are invoked as a result of evaluating the contained expression. In the example

```
class Test
{
    static int Multiply(int x, int y) {
        return x * y;
    }

    static int F() {
        return checked(Multiply(1000000, 1000000));
    }
}
```

the use of `checked` in `F` does not affect the evaluation of `x * y` in `Multiply`, so `x * y` is evaluated in the default overflow checking context.

The `unchecked` operator is convenient when writing constants of the signed integral types in hexadecimal notation. For example:

```
class Test
{
    public const int AllBits = unchecked((int)0xFFFFFFFF);

    public const int HighBit = unchecked((int)0x80000000);
}
```

Both of the hexadecimal constants above are of type `uint`. Because the constants are outside the `int` range, without the `unchecked` operator, the casts to `int` would produce compile-time errors.

The `checked` and `unchecked` operators and statements allow programmers to control certain aspects of some

numeric calculations. However, the behavior of some numeric operators depends on their operands' data types. For example, multiplying two decimals always results in an exception on overflow even within an explicitly `unchecked` construct. Similarly, multiplying two floats never results in an exception on overflow even within an explicitly `checked` construct. In addition, other operators are never affected by the mode of checking, whether default or explicit.

Default value expressions

A default value expression is used to obtain the default value (Default values) of a type. Typically a default value expression is used for type parameters, since it may not be known if the type parameter is a value type or a reference type. (No conversion exists from the `null` literal to a type parameter unless the type parameter is known to be a reference type.)

```
default_value_expression
    : 'default' '(' type ')'
    ;
```

If the *type* in a *default_value_expression* evaluates at run-time to a reference type, the result is `null` converted to that type. If the *type* in a *default_value_expression* evaluates at run-time to a value type, the result is the *value_type*'s default value (Default constructors).

A *default_value_expression* is a constant expression (Constant expressions) if the type is a reference type or a type parameter that is known to be a reference type (Type parameter constraints). In addition, a *default_value_expression* is a constant expression if the type is one of the following value types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, or any enumeration type.

Nameof expressions

A *nameof_expression* is used to obtain the name of a program entity as a constant string.

```
nameof_expression
    : 'nameof' '(' named_entity ')'
    ;

named_entity
    : simple_name
    | named_entity_target '.' identifier type_argument_list?
    ;

named_entity_target
    : 'this'
    | 'base'
    | named_entity
    | predefined_type
    | qualified_alias_member
    ;
```

Grammatically speaking, the *named_entity* operand is always an expression. Because `nameof` is not a reserved keyword, a nameof expression is always syntactically ambiguous with an invocation of the simple name `nameof`. For compatibility reasons, if a name lookup (Simple names) of the name `nameof` succeeds, the expression is treated as an *invocation_expression* -- regardless of whether the invocation is legal. Otherwise it is a *nameof_expression*.

The meaning of the *named_entity* of a *nameof_expression* is the meaning of it as an expression; that is, either as a *simple_name*, a *base_access* or a *member_access*. However, where the lookup described in Simple names and Member access results in an error because an instance member was found in a static context, a *nameof_expression* produces no such error.

It is a compile-time error for a *named_entity* designating a method group to have a *type_argument_list*. It is a

compile time error for a *named_entity_target* to have the type `dynamic`.

A *nameof_expression* is a constant expression of type `string`, and has no effect at runtime. Specifically, its *named_entity* is not evaluated, and is ignored for the purposes of definite assignment analysis (General rules for simple expressions). Its value is the last identifier of the *named_entity* before the optional final *type_argument_list*, transformed in the following way:

- The prefix "`@`", if used, is removed.
- Each *unicode_escape_sequence* is transformed into its corresponding Unicode character.
- Any *formatting_characters* are removed.

These are the same transformations applied in Identifiers when testing equality between identifiers.

TODO: examples

Anonymous method expressions

An *anonymous_method_expression* is one of two ways of defining an anonymous function. These are further described in Anonymous function expressions.

Unary operators

The `?`, `+`, `-`, `!`, `~`, `++`, `--`, cast, and `await` operators are called the unary operators.

```
unary_expression
    : primary_expression
    | null_conditional_expression
    | '+' unary_expression
    | '-' unary_expression
    | '!' unary_expression
    | '~' unary_expression
    | pre_increment_expression
    | pre_decrement_expression
    | cast_expression
    | await_expression
    | unary_expression_unsafe
    ;
```

If the operand of a *unary_expression* has the compile-time type `dynamic`, it is dynamically bound (Dynamic binding). In this case the compile-time type of the *unary_expression* is `dynamic`, and the resolution described below will take place at run-time using the run-time type of the operand.

Null-conditional operator

The null-conditional operator applies a list of operations to its operand only if that operand is non-null. Otherwise the result of applying the operator is `null`.

```
null_conditional_expression
    : primary_expression null_conditional_operations
    ;

null_conditional_operations
    : null_conditional_operations? '?' '.' identifier type_argument_list?
    | null_conditional_operations? '?' '[' argument_list ']'
    | null_conditional_operations '.' identifier type_argument_list?
    | null_conditional_operations '[' argument_list ']'
    | null_conditional_operations '(' argument_list? ')'
    ;
```

The list of operations can include member access and element access operations (which may themselves be

null-conditional), as well as invocation.

For example, the expression `a.b?[0]?.c()` is a *null_conditional_expression* with a *primary_expression* `a.b` and *null_conditional_operations* `?[0]` (null-conditional element access), `?.c` (null-conditional member access) and `()` (invocation).

For a *null_conditional_expression* `E` with a *primary_expression* `P`, let `E0` be the expression obtained by textually removing the leading `?` from each of the *null_conditional_operations* of `E` that have one. Conceptually, `E0` is the expression that will be evaluated if none of the null checks represented by the `?`s do find a `null`.

Also, let `E1` be the expression obtained by textually removing the leading `?` from just the first of the *null_conditional_operations* in `E`. This may lead to a *primary-expression* (if there was just one `?`) or to another *null_conditional_expression*.

For example, if `E` is the expression `a.b?[0]?.c()`, then `E0` is the expression `a.b[0].c()` and `E1` is the expression `a.b[0]?.c()`.

If `E0` is classified as nothing, then `E` is classified as nothing. Otherwise E is classified as a value.

`E0` and `E1` are used to determine the meaning of `E`:

- If `E` occurs as a *statement_expression* the meaning of `E` is the same as the statement

  ```
  if ((object)P != null) E1;
  ```

  except that P is evaluated only once.

- Otherwise, if `E0` is classified as nothing a compile-time error occurs.
- Otherwise, let `T0` be the type of `E0`.
  - If `T0` is a type parameter that is not known to be a reference type or a non-nullable value type, a compile-time error occurs.
  - If `T0` is a non-nullable value type, then the type of `E` is `T0?`, and the meaning of `E` is the same as

    ```
    ((object)P == null) ? (T0?)null : E1
    ```

    except that `P` is evaluated only once.
  - Otherwise the type of E is T0, and the meaning of E is the same as

    ```
    ((object)P == null) ? null : E1
    ```

    except that `P` is evaluated only once.

If `E1` is itself a *null_conditional_expression*, then these rules are applied again, nesting the tests for `null` until there are no further `?`'s, and the expression has been reduced all the way down to the primary-expression `E0`.

For example, if the expression `a.b?[0]?.c()` occurs as a statement-expression, as in the statement:

```
a.b?[0]?.c();
```

its meaning is equivalent to:

```
if (a.b != null) a.b[0]?.c();
```

which again is equivalent to:

```
if (a.b != null) if (a.b[0] != null) a.b[0].c();
```

Except that `a.b` and `a.b[0]` are evaluated only once.

If it occurs in a context where its value is used, as in:

```
var x = a.b?[0]?.c();
```

and assuming that the type of the final invocation is not a non-nullable value type, its meaning is equivalent to:

```
var x = (a.b == null) ? null : (a.b[0] == null) ? null : a.b[0].c();
```

except that `a.b` and `a.b[0]` are evaluated only once.

Null-conditional expressions as projection initializers

A null-conditional expression is only allowed as a *member_declarator* in an *anonymous_object_creation_expression* (Anonymous object creation expressions) if it ends with an (optionally null-conditional) member access. Grammatically, this requirement can be expressed as:

```
null_conditional_member_access
    : primary_expression null_conditional_operations? '?' '.' identifier type_argument_list?
    | primary_expression null_conditional_operations '.' identifier type_argument_list?
    ;
```

This is a special case of the grammar for *null_conditional_expression* above. The production for *member_declarator* in Anonymous object creation expressions then includes only *null_conditional_member_access*.

Null-conditional expressions as statement expressions

A null-conditional expression is only allowed as a *statement_expression* (Expression statements) if it ends with an invocation. Grammatically, this requirement can be expressed as:

```
null_conditional_invocation_expression
    : primary_expression null_conditional_operations '(' argument_list? ')'
    ;
```

This is a special case of the grammar for *null_conditional_expression* above. The production for *statement_expression* in Expression statements then includes only *null_conditional_invocation_expression*.

Unary plus operator

For an operation of the form `+x`, unary operator overload resolution (Unary operator overload resolution) is applied to select a specific operator implementation. The operand is converted to the parameter type of the selected operator, and the type of the result is the return type of the operator. The predefined unary plus operators are:

```
int operator +(int x);
uint operator +(uint x);
long operator +(long x);
ulong operator +(ulong x);
float operator +(float x);
double operator +(double x);
decimal operator +(decimal x);
```

For each of these operators, the result is simply the value of the operand.

Unary minus operator

For an operation of the form `-x`, unary operator overload resolution (Unary operator overload resolution) is applied to select a specific operator implementation. The operand is converted to the parameter type of the selected operator, and the type of the result is the return type of the operator. The predefined negation operators are:

- Integer negation:

  ```
  int operator -(int x);
  long operator -(long x);
  ```

  The result is computed by subtracting `x` from zero. If the value of `x` is the smallest representable value of the operand type (-2^31 for `int` or -2^63 for `long`), then the mathematical negation of `x` is not representable within the operand type. If this occurs within a `checked` context, a `System.OverflowException` is thrown; if it occurs within an `unchecked` context, the result is the value of the operand and the overflow is not reported.

  If the operand of the negation operator is of type `uint`, it is converted to type `long`, and the type of the result is `long`. An exception is the rule that permits the `int` value -2147483648 (-2^31) to be written as a decimal integer literal (Integer literals).

  If the operand of the negation operator is of type `ulong`, a compile-time error occurs. An exception is the rule that permits the `long` value -9223372036854775808 (-2^63) to be written as a decimal integer literal (Integer literals).

- Floating-point negation:

  ```
  float operator -(float x);
  double operator -(double x);
  ```

  The result is the value of `x` with its sign inverted. If `x` is NaN, the result is also NaN.

- Decimal negation:

  ```
  decimal operator -(decimal x);
  ```

  The result is computed by subtracting `x` from zero. Decimal negation is equivalent to using the unary minus operator of type `System.Decimal`.

Logical negation operator

For an operation of the form `!x`, unary operator overload resolution (Unary operator overload resolution) is applied to select a specific operator implementation. The operand is converted to the parameter type of the selected operator, and the type of the result is the return type of the operator. Only one predefined logical negation operator exists:

```
bool operator !(bool x);
```

This operator computes the logical negation of the operand: If the operand is `true`, the result is `false`. If the operand is `false`, the result is `true`.

Bitwise complement operator

For an operation of the form `~x`, unary operator overload resolution (Unary operator overload resolution) is applied to select a specific operator implementation. The operand is converted to the parameter type of the selected operator, and the type of the result is the return type of the operator. The predefined bitwise complement operators are:

```
int operator ~(int x);
uint operator ~(uint x);
long operator ~(long x);
ulong operator ~(ulong x);
```

For each of these operators, the result of the operation is the bitwise complement of `x`.

Every enumeration type `E` implicitly provides the following bitwise complement operator:

```
E operator ~(E x);
```

The result of evaluating `~x`, where `x` is an expression of an enumeration type `E` with an underlying type `U`, is exactly the same as evaluating `(E)(~(U)x)`, except that the conversion to `E` is always performed as if in an `unchecked` context (The checked and unchecked operators).

Prefix increment and decrement operators

```
pre_increment_expression
    : '++' unary_expression
    ;

pre_decrement_expression
    : '--' unary_expression
    ;
```

The operand of a prefix increment or decrement operation must be an expression classified as a variable, a property access, or an indexer access. The result of the operation is a value of the same type as the operand.

If the operand of a prefix increment or decrement operation is a property or indexer access, the property or indexer must have both a `get` and a `set` accessor. If this is not the case, a binding-time error occurs.

Unary operator overload resolution (Unary operator overload resolution) is applied to select a specific operator implementation. Predefined `++` and `--` operators exist for the following types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, and any enum type. The predefined `++` operators return the value produced by adding 1 to the operand, and the predefined `--` operators return the value produced by subtracting 1 from the operand. In a `checked` context, if the result of this addition or subtraction is outside the range of the result type and the result type is an integral type or enum type, a `System.OverflowException` is thrown.

The run-time processing of a prefix increment or decrement operation of the form `++x` or `--x` consists of the following steps:

- If `x` is classified as a variable:

- `x` is evaluated to produce the variable.
- The selected operator is invoked with the value of `x` as its argument.
- The value returned by the operator is stored in the location given by the evaluation of `x`.
- The value returned by the operator becomes the result of the operation.

- If `x` is classified as a property or indexer access:
  - The instance expression (if `x` is not `static`) and the argument list (if `x` is an indexer access) associated with `x` are evaluated, and the results are used in the subsequent `get` and `set` accessor invocations.
  - The `get` accessor of `x` is invoked.
  - The selected operator is invoked with the value returned by the `get` accessor as its argument.
  - The `set` accessor of `x` is invoked with the value returned by the operator as its `value` argument.
  - The value returned by the operator becomes the result of the operation.

The `++` and `--` operators also support postfix notation (Postfix increment and decrement operators). Typically, the result of `x++` or `x--` is the value of `x` before the operation, whereas the result of `++x` or `--x` is the value of `x` after the operation. In either case, `x` itself has the same value after the operation.

An `operator++` or `operator--` implementation can be invoked using either postfix or prefix notation. It is not possible to have separate operator implementations for the two notations.

Cast expressions

A *cast_expression* is used to explicitly convert an expression to a given type.

```
cast_expression
    : '(' type ')' unary_expression
    ;
```

A *cast_expression* of the form `(T)E`, where `T` is a *type* and `E` is a *unary_expression*, performs an explicit conversion (Explicit conversions) of the value of `E` to type `T`. If no explicit conversion exists from `E` to `T`, a binding-time error occurs. Otherwise, the result is the value produced by the explicit conversion. The result is always classified as a value, even if `E` denotes a variable.

The grammar for a *cast_expression* leads to certain syntactic ambiguities. For example, the expression `(x)-y` could either be interpreted as a *cast_expression* (a cast of `-y` to type `x`) or as an *additive_expression* combined with a *parenthesized_expression* (which computes the value `x - y)`.

To resolve *cast_expression* ambiguities, the following rule exists: A sequence of one or more *token*s (White space) enclosed in parentheses is considered the start of a *cast_expression* only if at least one of the following are true:

- The sequence of tokens is correct grammar for a *type*, but not for an *expression*.
- The sequence of tokens is correct grammar for a *type*, and the token immediately following the closing parentheses is the token "`~`", the token "`!`", the token "`(`", an *identifier* (Unicode character escape sequences), a *literal* (Literals), or any *keyword* (Keywords) except `as` and `is`.

The term "correct grammar" above means only that the sequence of tokens must conform to the particular grammatical production. It specifically does not consider the actual meaning of any constituent identifiers. For example, if `x` and `y` are identifiers, then `x.y` is correct grammar for a type, even if `x.y` doesn't actually denote a type.

From the disambiguation rule it follows that, if `x` and `y` are identifiers, `(x)y`, `(x)(y)`, and `(x)(-y)` are *cast_expression*s, but `(x)-y` is not, even if `x` identifies a type. However, if `x` is a keyword that identifies a predefined type (such as `int`), then all four forms are *cast_expression*s (because such a keyword could not

possibly be an expression by itself).

Await expressions

The await operator is used to suspend evaluation of the enclosing async function until the asynchronous operation represented by the operand has completed.

```
await_expression
    : 'await' unary_expression
    ;
```

An *await_expression* is only allowed in the body of an async function (Async functions). Within the nearest enclosing async function, an *await_expression* may not occur in these places:

- Inside a nested (non-async) anonymous function
- Inside the block of a *lock_statement*
- In an unsafe context

Note that an *await_expression* cannot occur in most places within a *query_expression*, because those are syntactically transformed to use non-async lambda expressions.

Inside of an async function, `await` cannot be used as an identifier. There is therefore no syntactic ambiguity between await-expressions and various expressions involving identifiers. Outside of async functions, `await` acts as a normal identifier.

The operand of an *await_expression* is called the ***task***. It represents an asynchronous operation that may or may not be complete at the time the *await_expression* is evaluated. The purpose of the await operator is to suspend execution of the enclosing async function until the awaited task is complete, and then obtain its outcome.

Awaitable expressions

The task of an await expression is required to be ***awaitable***. An expression `t` is awaitable if one of the following holds:

- `t` is of compile time type `dynamic`
- `t` has an accessible instance or extension method called `GetAwaiter` with no parameters and no type parameters, and a return type `A` for which all of the following hold:
  - `A` implements the interface `System.Runtime.CompilerServices.INotifyCompletion` (hereafter known as `INotifyCompletion` for brevity)
  - `A` has an accessible, readable instance property `IsCompleted` of type `bool`
  - `A` has an accessible instance method `GetResult` with no parameters and no type parameters

The purpose of the `GetAwaiter` method is to obtain an ***awaiter*** for the task. The type `A` is called the ***awaiter type*** for the await expression.

The purpose of the `IsCompleted` property is to determine if the task is already complete. If so, there is no need to suspend evaluation.

The purpose of the `INotifyCompletion.OnCompleted` method is to sign up a "continuation" to the task; i.e. a delegate (of type `System.Action`) that will be invoked once the task is complete.

The purpose of the `GetResult` method is to obtain the outcome of the task once it is complete. This outcome may be successful completion, possibly with a result value, or it may be an exception which is thrown by the `GetResult` method.

Classification of await expressions

The expression `await t` is classified the same way as the expression `(t).GetAwaiter().GetResult()`. Thus, if the return type of `GetResult` is `void`, the *await_expression* is classified as nothing. If it has a non-void return type

`T`, the *await_expression* is classified as a value of type `T`.

Runtime evaluation of await expressions

At runtime, the expression `await t` is evaluated as follows:

- An awaiter `a` is obtained by evaluating the expression `(t).GetAwaiter()`.
- A `bool` `b` is obtained by evaluating the expression `(a).IsCompleted`.
- If `b` is `false` then evaluation depends on whether `a` implements the interface `System.Runtime.CompilerServices.ICriticalNotifyCompletion` (hereafter known as `ICriticalNotifyCompletion` for brevity). This check is done at binding time; i.e. at runtime if `a` has the compile time type `dynamic`, and at compile time otherwise. Let `r` denote the resumption delegate (Async functions):
  - If `a` does not implement `ICriticalNotifyCompletion`, then the expression `(a as (INotifyCompletion)).OnCompleted(r)` is evaluated.
  - If `a` does implement `ICriticalNotifyCompletion`, then the expression `(a as (ICriticalNotifyCompletion)).UnsafeOnCompleted(r)` is evaluated.
  - Evaluation is then suspended, and control is returned to the current caller of the async function.
- Either immediately after (if `b` was `true`), or upon later invocation of the resumption delegate (if `b` was `false`), the expression `(a).GetResult()` is evaluated. If it returns a value, that value is the result of the *await_expression*. Otherwise the result is nothing.

An awaiter's implementation of the interface methods `INotifyCompletion.OnCompleted` and `ICriticalNotifyCompletion.UnsafeOnCompleted` should cause the delegate `r` to be invoked at most once. Otherwise, the behavior of the enclosing async function is undefined.

Arithmetic operators

The `*`, `/`, `%`, `+`, and `-` operators are called the arithmetic operators.

```
multiplicative_expression
    : unary_expression
    | multiplicative_expression '*' unary_expression
    | multiplicative_expression '/' unary_expression
    | multiplicative_expression '%' unary_expression
    ;

additive_expression
    : multiplicative_expression
    | additive_expression '+' multiplicative_expression
    | additive_expression '-' multiplicative_expression
    ;
```

If an operand of an arithmetic operator has the compile-time type `dynamic`, then the expression is dynamically bound (Dynamic binding). In this case the compile-time type of the expression is `dynamic`, and the resolution described below will take place at run-time using the run-time type of those operands that have the compile-time type `dynamic`.

Multiplication operator

For an operation of the form `x * y`, binary operator overload resolution (Binary operator overload resolution) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined multiplication operators are listed below. The operators all compute the product of `x` and `y`.

- Integer multiplication:

```
int operator *(int x, int y);
uint operator *(uint x, uint y);
long operator *(long x, long y);
ulong operator *(ulong x, ulong y);
```

In a `checked` context, if the product is outside the range of the result type, a `System.OverflowException` is thrown. In an `unchecked` context, overflows are not reported and any significant high-order bits outside the range of the result type are discarded.

- Floating-point multiplication:

```
float operator *(float x, float y);
double operator *(double x, double y);
```

The product is computed according to the rules of IEEE 754 arithmetic. The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, `x` and `y` are positive finite values. `z` is the result of `x * y`. If the result is too large for the destination type, `z` is infinity. If the result is too small for the destination type, `z` is zero.

	+Y	**-Y**	**+0**	**-0**	**+INF**	**-INF**	**NAN**
+x	+z	-z	+0	-0	+inf	-inf	NaN
-x	-z	+z	-0	+0	-inf	+inf	NaN
+0	+0	-0	+0	-0	NaN	NaN	NaN
-0	-0	+0	-0	+0	NaN	NaN	NaN
+inf	+inf	-inf	NaN	NaN	+inf	-inf	NaN
-inf	-inf	+inf	NaN	NaN	-inf	+inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Decimal multiplication:

```
decimal operator *(decimal x, decimal y);
```

If the resulting value is too large to represent in the `decimal` format, a `System.OverflowException` is thrown. If the result value is too small to represent in the `decimal` format, the result is zero. The scale of the result, before any rounding, is the sum of the scales of the two operands.

Decimal multiplication is equivalent to using the multiplication operator of type `System.Decimal`.

Division operator

For an operation of the form `x / y`, binary operator overload resolution (Binary operator overload resolution) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined division operators are listed below. The operators all compute the quotient of `x` and `y`.

- Integer division:

```
int operator /(int x, int y);
uint operator /(uint x, uint y);
long operator /(long x, long y);
ulong operator /(ulong x, ulong y);
```

If the value of the right operand is zero, a `System.DivideByZeroException` is thrown.

The division rounds the result towards zero. Thus the absolute value of the result is the largest possible integer that is less than or equal to the absolute value of the quotient of the two operands. The result is zero or positive when the two operands have the same sign and zero or negative when the two operands have opposite signs.

If the left operand is the smallest representable `int` or `long` value and the right operand is `-1`, an overflow occurs. In a `checked` context, this causes a `System.ArithmeticException` (or a subclass thereof) to be thrown. In an `unchecked` context, it is implementation-defined as to whether a `System.ArithmeticException` (or a subclass thereof) is thrown or the overflow goes unreported with the resulting value being that of the left operand.

- Floating-point division:

```
float operator /(float x, float y);
double operator /(double x, double y);
```

The quotient is computed according to the rules of IEEE 754 arithmetic. The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, `x` and `y` are positive finite values. `z` is the result of `x / y`. If the result is too large for the destination type, `z` is infinity. If the result is too small for the destination type, `z` is zero.

	+Y	-Y	+0	-0	+INF	-INF	NAN
+x	+z	-z	+inf	-inf	+0	-0	NaN
-x	-z	+z	-inf	+inf	-0	+0	NaN
+0	+0	-0	NaN	NaN	+0	-0	NaN
-0	-0	+0	NaN	NaN	-0	+0	NaN
+inf	+inf	-inf	+inf	-inf	NaN	NaN	NaN
-inf	-inf	+inf	-inf	+inf	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Decimal division:

```
decimal operator /(decimal x, decimal y);
```

If the value of the right operand is zero, a `System.DivideByZeroException` is thrown. If the resulting value is too large to represent in the `decimal` format, a `System.OverflowException` is thrown. If the result value is too small to represent in the `decimal` format, the result is zero. The scale of the result is the smallest scale that will preserve a result equal to the nearest representable decimal value to the true mathematical result.

Decimal division is equivalent to using the division operator of type `System.Decimal`.

Remainder operator

For an operation of the form `x % y`, binary operator overload resolution (Binary operator overload resolution) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined remainder operators are listed below. The operators all compute the remainder of the division between `x` and `y`.

- Integer remainder:

```
int operator %(int x, int y);
uint operator %(uint x, uint y);
long operator %(long x, long y);
ulong operator %(ulong x, ulong y);
```

The result of `x % y` is the value produced by `x - (x / y) * y`. If `y` is zero, a `System.DivideByZeroException` is thrown.

If the left operand is the smallest `int` or `long` value and the right operand is `-1`, a `System.OverflowException` is thrown. In no case does `x % y` throw an exception where `x / y` would not throw an exception.

- Floating-point remainder:

```
float operator %(float x, float y);
double operator %(double x, double y);
```

The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, `x` and `y` are positive finite values. `z` is the result of `x % y` and is computed as `x - n * y`, where `n` is the largest possible integer that is less than or equal to `x / y`. This method of computing the remainder is analogous to that used for integer operands, but differs from the IEEE 754 definition (in which `n` is the integer closest to `x / y`).

	+y	-y	+0	-0	+inf	-inf	NaN
+x	+z	+z	NaN	NaN	x	x	NaN
-x	-z	-z	NaN	NaN	-x	-x	NaN
+0	+0	+0	NaN	NaN	+0	+0	NaN
-0	-0	-0	NaN	NaN	-0	-0	NaN
+inf	NaN	NaN	NaN	NaN	NaN	NaN	NaN
-inf	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Decimal remainder:

```
decimal operator %(decimal x, decimal y);
```

If the value of the right operand is zero, a `System.DivideByZeroException` is thrown. The scale of the result, before any rounding, is the larger of the scales of the two operands, and the sign of the result, if non-zero, is the same as that of `x`.

Decimal remainder is equivalent to using the remainder operator of type `System.Decimal`.

Addition operator

For an operation of the form `x + y`, binary operator overload resolution (Binary operator overload resolution) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined addition operators are listed below. For numeric and enumeration types, the predefined addition operators compute the sum of the two operands. When one or both operands are of type string, the predefined addition operators concatenate the string representation of the operands.

- Integer addition:

```
int operator +(int x, int y);
uint operator +(uint x, uint y);
long operator +(long x, long y);
ulong operator +(ulong x, ulong y);
```

In a `checked` context, if the sum is outside the range of the result type, a `System.OverflowException` is thrown. In an `unchecked` context, overflows are not reported and any significant high-order bits outside the range of the result type are discarded.

- Floating-point addition:

```
float operator +(float x, float y);
double operator +(double x, double y);
```

The sum is computed according to the rules of IEEE 754 arithmetic. The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, `x` and `y` are nonzero finite values, and `z` is the result of `x + y`. If `x` and `y` have the same magnitude but opposite signs, `z` is positive zero. If `x + y` is too large to represent in the destination type, `z` is an infinity with the same sign as `x + y`.

	Y	**+0**	**-0**	**+INF**	**-INF**	**NAN**
x	z	x	x	+inf	-inf	NaN
+0	y	+0	+0	+inf	-inf	NaN
-0	y	+0	-0	+inf	-inf	NaN
+inf	+inf	+inf	+inf	+inf	NaN	NaN
-inf	-inf	-inf	-inf	NaN	-inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Decimal addition:

```
decimal operator +(decimal x, decimal y);
```

  If the resulting value is too large to represent in the `decimal` format, a `System.OverflowException` is thrown. The scale of the result, before any rounding, is the larger of the scales of the two operands.

  Decimal addition is equivalent to using the addition operator of type `System.Decimal`.

- Enumeration addition. Every enumeration type implicitly provides the following predefined operators, where `E` is the enum type, and `U` is the underlying type of `E`:

```
E operator +(E x, U y);
E operator +(U x, E y);
```

  At run-time these operators are evaluated exactly as `(E)((U)x + (U)y)`.

- String concatenation:

```
string operator +(string x, string y);
string operator +(string x, object y);
string operator +(object x, string y);
```

  These overloads of the binary `+` operator perform string concatenation. If an operand of string concatenation is `null`, an empty string is substituted. Otherwise, any non-string argument is converted to its string representation by invoking the virtual `ToString` method inherited from type `object`. If `ToString` returns `null`, an empty string is substituted.

```
using System;

class Test
{
    static void Main() {
        string s = null;
        Console.WriteLine("s = >" + s + "<");        // displays s = ><
        int i = 1;
        Console.WriteLine("i = " + i);               // displays i = 1
        float f = 1.2300E+15F;
        Console.WriteLine("f = " + f);               // displays f = 1.23E+15
        decimal d = 2.900m;
        Console.WriteLine("d = " + d);               // displays d = 2.900
    }
}
```

  The result of the string concatenation operator is a string that consists of the characters of the left operand followed by the characters of the right operand. The string concatenation operator never returns a `null` value. A `System.OutOfMemoryException` may be thrown if there is not enough memory available to allocate the resulting string.

- Delegate combination. Every delegate type implicitly provides the following predefined operator, where `D` is the delegate type:

```
D operator +(D x, D y);
```

  The binary `+` operator performs delegate combination when both operands are of some delegate type

`D`. (If the operands have different delegate types, a binding-time error occurs.) If the first operand is `null`, the result of the operation is the value of the second operand (even if that is also `null`). Otherwise, if the second operand is `null`, then the result of the operation is the value of the first operand. Otherwise, the result of the operation is a new delegate instance that, when invoked, invokes the first operand and then invokes the second operand. For examples of delegate combination, see Subtraction operator and Delegate invocation. Since `System.Delegate` is not a delegate type, `operator +` is not defined for it.

Subtraction operator

For an operation of the form `x - y`, binary operator overload resolution (Binary operator overload resolution) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined subtraction operators are listed below. The operators all subtract `y` from `x`.

- Integer subtraction:

```
int operator -(int x, int y);
uint operator -(uint x, uint y);
long operator -(long x, long y);
ulong operator -(ulong x, ulong y);
```

In a `checked` context, if the difference is outside the range of the result type, a `System.OverflowException` is thrown. In an `unchecked` context, overflows are not reported and any significant high-order bits outside the range of the result type are discarded.

- Floating-point subtraction:

```
float operator -(float x, float y);
double operator -(double x, double y);
```

The difference is computed according to the rules of IEEE 754 arithmetic. The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaNs. In the table, `x` and `y` are nonzero finite values, and `z` is the result of `x - y`. If `x` and `y` are equal, `z` is positive zero. If `x - y` is too large to represent in the destination type, `z` is an infinity with the same sign as `x - y`.

	Y	**+0**	**-0**	**+INF**	**-INF**	**NAN**
x	z	x	x	-inf	+inf	NaN
+0	-y	+0	+0	-inf	+inf	NaN
-0	-y	-0	+0	-inf	+inf	NaN
+inf	+inf	+inf	+inf	NaN	+inf	NaN
-inf	-inf	-inf	-inf	-inf	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Decimal subtraction:

```
decimal operator -(decimal x, decimal y);
```

If the resulting value is too large to represent in the `decimal` format, a `System.OverflowException` is thrown. The scale of the result, before any rounding, is the larger of the scales of the two operands.

Decimal subtraction is equivalent to using the subtraction operator of type `System.Decimal`.

- Enumeration subtraction. Every enumeration type implicitly provides the following predefined operator, where `E` is the enum type, and `U` is the underlying type of `E`:

  ```
  U operator -(E x, E y);
  ```

  This operator is evaluated exactly as `(U)((U)x - (U)y)`. In other words, the operator computes the difference between the ordinal values of `x` and `y`, and the type of the result is the underlying type of the enumeration.

  ```
  E operator -(E x, U y);
  ```

  This operator is evaluated exactly as `(E)((U)x - y)`. In other words, the operator subtracts a value from the underlying type of the enumeration, yielding a value of the enumeration.

- Delegate removal. Every delegate type implicitly provides the following predefined operator, where `D` is the delegate type:

  ```
  D operator -(D x, D y);
  ```

  The binary `-` operator performs delegate removal when both operands are of some delegate type `D`. If the operands have different delegate types, a binding-time error occurs. If the first operand is `null`, the result of the operation is `null`. Otherwise, if the second operand is `null`, then the result of the operation is the value of the first operand. Otherwise, both operands represent invocation lists (Delegate declarations) having one or more entries, and the result is a new invocation list consisting of the first operand's list with the second operand's entries removed from it, provided the second operand's list is a proper contiguous sublist of the first's. (To determine sublist equality, corresponding entries are compared as for the delegate equality operator (Delegate equality operators).) Otherwise, the result is the value of the left operand. Neither of the operands' lists is changed in the process. If the second operand's list matches multiple sublists of contiguous entries in the first operand's list, the right-most matching sublist of contiguous entries is removed. If removal results in an empty list, the result is `null`. For example:

```
delegate void D(int x);

class C
{
    public static void M1(int i) { /* ... */ }
    public static void M2(int i) { /* ... */ }
}

class Test
{
    static void Main() {
        D cd1 = new D(C.M1);
        D cd2 = new D(C.M2);
        D cd3 = cd1 + cd2 + cd2 + cd1;   // M1 + M2 + M2 + M1
        cd3 -= cd1;                      // => M1 + M2 + M2

        cd3 = cd1 + cd2 + cd2 + cd1;     // M1 + M2 + M2 + M1
        cd3 -= cd1 + cd2;                // => M2 + M1

        cd3 = cd1 + cd2 + cd2 + cd1;     // M1 + M2 + M2 + M1
        cd3 -= cd2 + cd2;                // => M1 + M1

        cd3 = cd1 + cd2 + cd2 + cd1;     // M1 + M2 + M2 + M1
        cd3 -= cd2 + cd1;                // => M1 + M2

        cd3 = cd1 + cd2 + cd2 + cd1;     // M1 + M2 + M2 + M1
        cd3 -= cd1 + cd1;                // => M1 + M2 + M2 + M1
    }
}
```

Shift operators

The `<<` and `>>` operators are used to perform bit shifting operations.

```
shift_expression
    : additive_expression
    | shift_expression '<<' additive_expression
    | shift_expression right_shift additive_expression
    ;
```

If an operand of a *shift_expression* has the compile-time type `dynamic`, then the expression is dynamically bound (Dynamic binding). In this case the compile-time type of the expression is `dynamic`, and the resolution described below will take place at run-time using the run-time type of those operands that have the compile-time type `dynamic`.

For an operation of the form `x << count` or `x >> count`, binary operator overload resolution (Binary operator overload resolution) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

When declaring an overloaded shift operator, the type of the first operand must always be the class or struct containing the operator declaration, and the type of the second operand must always be `int`.

The predefined shift operators are listed below.

- Shift left:

```
int operator <<(int x, int count);
uint operator <<(uint x, int count);
long operator <<(long x, int count);
ulong operator <<(ulong x, int count);
```

The `<<` operator shifts `x` left by a number of bits computed as described below.

The high-order bits outside the range of the result type of `x` are discarded, the remaining bits are shifted left, and the low-order empty bit positions are set to zero.

- Shift right:

```
int operator >>(int x, int count);
uint operator >>(uint x, int count);
long operator >>(long x, int count);
ulong operator >>(ulong x, int count);
```

The `>>` operator shifts `x` right by a number of bits computed as described below.

When `x` is of type `int` or `long`, the low-order bits of `x` are discarded, the remaining bits are shifted right, and the high-order empty bit positions are set to zero if `x` is non-negative and set to one if `x` is negative.

When `x` is of type `uint` or `ulong`, the low-order bits of `x` are discarded, the remaining bits are shifted right, and the high-order empty bit positions are set to zero.

For the predefined operators, the number of bits to shift is computed as follows:

- When the type of `x` is `int` or `uint`, the shift count is given by the low-order five bits of `count`. In other words, the shift count is computed from `count & 0x1F`.
- When the type of `x` is `long` or `ulong`, the shift count is given by the low-order six bits of `count`. In other words, the shift count is computed from `count & 0x3F`.

If the resulting shift count is zero, the shift operators simply return the value of `x`.

Shift operations never cause overflows and produce the same results in `checked` and `unchecked` contexts.

When the left operand of the `>>` operator is of a signed integral type, the operator performs an arithmetic shift right wherein the value of the most significant bit (the sign bit) of the operand is propagated to the high-order empty bit positions. When the left operand of the `>>` operator is of an unsigned integral type, the operator performs a logical shift right wherein high-order empty bit positions are always set to zero. To perform the opposite operation of that inferred from the operand type, explicit casts can be used. For example, if `x` is a variable of type `int`, the operation `unchecked((int)((uint)x >> y))` performs a logical shift right of `x`.

Relational and type-testing operators

The `==`, `!=`, `<`, `>`, `<=`, `>=`, `is` and `as` operators are called the relational and type-testing operators.

```
relational_expression
    : shift_expression
    | relational_expression '<' shift_expression
    | relational_expression '>' shift_expression
    | relational_expression '<=' shift_expression
    | relational_expression '>=' shift_expression
    | relational_expression 'is' type
    | relational_expression 'as' type
    ;

equality_expression
    : relational_expression
    | equality_expression '==' relational_expression
    | equality_expression '!=' relational_expression
    ;
```

The `is` operator is described in The is operator and the `as` operator is described in The as operator.

The `==`, `!=`, `<`, `>`, `<=` and `>=` operators are *comparison operators*.

If an operand of a comparison operator has the compile-time type `dynamic`, then the expression is dynamically bound (Dynamic binding). In this case the compile-time type of the expression is `dynamic`, and the resolution described below will take place at run-time using the run-time type of those operands that have the compile-time type `dynamic`.

For an operation of the form `x` *op* `y`, where *op* is a comparison operator, overload resolution (Binary operator overload resolution) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined comparison operators are described in the following sections. All predefined comparison operators return a result of type `bool`, as described in the following table.

OPERATION	RESULT
`x == y`	`true` if `x` is equal to `y`, `false` otherwise
`x != y`	`true` if `x` is not equal to `y`, `false` otherwise
`x < y`	`true` if `x` is less than `y`, `false` otherwise
`x > y`	`true` if `x` is greater than `y`, `false` otherwise
`x <= y`	`true` if `x` is less than or equal to `y`, `false` otherwise
`x >= y`	`true` if `x` is greater than or equal to `y`, `false` otherwise

Integer comparison operators

The predefined integer comparison operators are:

```
bool operator ==(int x, int y);
bool operator ==(uint x, uint y);
bool operator ==(long x, long y);
bool operator ==(ulong x, ulong y);

bool operator !=(int x, int y);
bool operator !=(uint x, uint y);
bool operator !=(long x, long y);
bool operator !=(ulong x, ulong y);

bool operator <(int x, int y);
bool operator <(uint x, uint y);
bool operator <(long x, long y);
bool operator <(ulong x, ulong y);

bool operator >(int x, int y);
bool operator >(uint x, uint y);
bool operator >(long x, long y);
bool operator >(ulong x, ulong y);

bool operator <=(int x, int y);
bool operator <=(uint x, uint y);
bool operator <=(long x, long y);
bool operator <=(ulong x, ulong y);

bool operator >=(int x, int y);
bool operator >=(uint x, uint y);
bool operator >=(long x, long y);
bool operator >=(ulong x, ulong y);
```

Each of these operators compares the numeric values of the two integer operands and returns a `bool` value that indicates whether the particular relation is `true` or `false`.

Floating-point comparison operators

The predefined floating-point comparison operators are:

```
bool operator ==(float x, float y);
bool operator ==(double x, double y);

bool operator !=(float x, float y);
bool operator !=(double x, double y);

bool operator <(float x, float y);
bool operator <(double x, double y);

bool operator >(float x, float y);
bool operator >(double x, double y);

bool operator <=(float x, float y);
bool operator <=(double x, double y);

bool operator >=(float x, float y);
bool operator >=(double x, double y);
```

The operators compare the operands according to the rules of the IEEE 754 standard:

- If either operand is NaN, the result is `false` for all operators except `!=`, for which the result is `true`. For any two operands, `x != y` always produces the same result as `!(x == y)`. However, when one or both operands are NaN, the `<`, `>`, `<=`, and `>=` operators do not produce the same results as the logical negation of the opposite operator. For example, if either of `x` and `y` is NaN, then `x < y` is `false`, but `!(x >= y)` is `true`.

- When neither operand is NaN, the operators compare the values of the two floating-point operands with respect to the ordering

  ```
  -inf < -max < ... < -min < -0.0 == +0.0 < +min < ... < +max < +inf
  ```

  where `min` and `max` are the smallest and largest positive finite values that can be represented in the given floating-point format. Notable effects of this ordering are:

  - Negative and positive zeros are considered equal.
  - A negative infinity is considered less than all other values, but equal to another negative infinity.
  - A positive infinity is considered greater than all other values, but equal to another positive infinity.

Decimal comparison operators

The predefined decimal comparison operators are:

```
bool operator ==(decimal x, decimal y);
bool operator !=(decimal x, decimal y);
bool operator <(decimal x, decimal y);
bool operator >(decimal x, decimal y);
bool operator <=(decimal x, decimal y);
bool operator >=(decimal x, decimal y);
```

Each of these operators compares the numeric values of the two decimal operands and returns a `bool` value that indicates whether the particular relation is `true` or `false`. Each decimal comparison is equivalent to using the corresponding relational or equality operator of type `System.Decimal`.

Boolean equality operators

The predefined boolean equality operators are:

```
bool operator ==(bool x, bool y);
bool operator !=(bool x, bool y);
```

The result of `==` is `true` if both `x` and `y` are `true` or if both `x` and `y` are `false`. Otherwise, the result is `false`.

The result of `!=` is `false` if both `x` and `y` are `true` or if both `x` and `y` are `false`. Otherwise, the result is `true`. When the operands are of type `bool`, the `!=` operator produces the same result as the `^` operator.

Enumeration comparison operators

Every enumeration type implicitly provides the following predefined comparison operators:

```
bool operator ==(E x, E y);
bool operator !=(E x, E y);
bool operator <(E x, E y);
bool operator >(E x, E y);
bool operator <=(E x, E y);
bool operator >=(E x, E y);
```

The result of evaluating `x op y`, where `x` and `y` are expressions of an enumeration type `E` with an underlying type `U`, and `op` is one of the comparison operators, is exactly the same as evaluating `((U)x) op ((U)y)`. In other words, the enumeration type comparison operators simply compare the underlying integral values of the two operands.

Reference type equality operators

The predefined reference type equality operators are:

```
bool operator ==(object x, object y);
bool operator !=(object x, object y);
```

The operators return the result of comparing the two references for equality or non-equality.

Since the predefined reference type equality operators accept operands of type `object`, they apply to all types that do not declare applicable `operator ==` and `operator !=` members. Conversely, any applicable user-defined equality operators effectively hide the predefined reference type equality operators.

The predefined reference type equality operators require one of the following:

- Both operands are a value of a type known to be a *reference_type* or the literal `null`. Furthermore, an explicit reference conversion (Explicit reference conversions) exists from the type of either operand to the type of the other operand.
- One operand is a value of type `T` where `T` is a *type_parameter* and the other operand is the literal `null`. Furthermore `T` does not have the value type constraint.

Unless one of these conditions are true, a binding-time error occurs. Notable implications of these rules are:

- It is a binding-time error to use the predefined reference type equality operators to compare two references that are known to be different at binding-time. For example, if the binding-time types of the operands are two class types `A` and `B`, and if neither `A` nor `B` derives from the other, then it would be impossible for the two operands to reference the same object. Thus, the operation is considered a binding-time error.
- The predefined reference type equality operators do not permit value type operands to be compared. Therefore, unless a struct type declares its own equality operators, it is not possible to compare values of that struct type.
- The predefined reference type equality operators never cause boxing operations to occur for their operands. It would be meaningless to perform such boxing operations, since references to the newly allocated boxed instances would necessarily differ from all other references.
- If an operand of a type parameter type `T` is compared to `null`, and the run-time type of `T` is a value type, the result of the comparison is `false`.

The following example checks whether an argument of an unconstrained type parameter type is `null`.

```
class C<T>
{
    void F(T x) {
        if (x == null) throw new ArgumentNullException();
        ...
    }
}
```

The `x == null` construct is permitted even though `T` could represent a value type, and the result is simply defined to be `false` when `T` is a value type.

For an operation of the form `x == y` or `x != y`, if any applicable `operator ==` or `operator !=` exists, the operator overload resolution (Binary operator overload resolution) rules will select that operator instead of the predefined reference type equality operator. However, it is always possible to select the predefined reference type equality operator by explicitly casting one or both of the operands to type `object`. The example

```
using System;

class Test
{
    static void Main() {
        string s = "Test";
        string t = string.Copy(s);
        Console.WriteLine(s == t);
        Console.WriteLine((object)s == t);
        Console.WriteLine(s == (object)t);
        Console.WriteLine((object)s == (object)t);
    }
}
```

produces the output

```
True
False
False
False
```

The `s` and `t` variables refer to two distinct `string` instances containing the same characters. The first comparison outputs `True` because the predefined string equality operator (String equality operators) is selected when both operands are of type `string`. The remaining comparisons all output `False` because the predefined reference type equality operator is selected when one or both of the operands are of type `object`.

Note that the above technique is not meaningful for value types. The example

```
class Test
{
    static void Main() {
        int i = 123;
        int j = 123;
        System.Console.WriteLine((object)i == (object)j);
    }
}
```

outputs `False` because the casts create references to two separate instances of boxed `int` values.

String equality operators

The predefined string equality operators are:

```
bool operator ==(string x, string y);
bool operator !=(string x, string y);
```

Two `string` values are considered equal when one of the following is true:

- Both values are `null`.
- Both values are non-null references to string instances that have identical lengths and identical characters in each character position.

The string equality operators compare string values rather than string references. When two separate string instances contain the exact same sequence of characters, the values of the strings are equal, but the references are different. As described in Reference type equality operators, the reference type equality operators can be used to compare string references instead of string values.

Delegate equality operators

Every delegate type implicitly provides the following predefined comparison operators:

```
bool operator ==(System.Delegate x, System.Delegate y);
bool operator !=(System.Delegate x, System.Delegate y);
```

Two delegate instances are considered equal as follows:

- If either of the delegate instances is `null`, they are equal if and only if both are `null`.
- If the delegates have different run-time type they are never equal.
- If both of the delegate instances have an invocation list (Delegate declarations), those instances are equal if and only if their invocation lists are the same length, and each entry in one's invocation list is equal (as defined below) to the corresponding entry, in order, in the other's invocation list.

The following rules govern the equality of invocation list entries:

- If two invocation list entries both refer to the same static method then the entries are equal.
- If two invocation list entries both refer to the same non-static method on the same target object (as defined by the reference equality operators) then the entries are equal.
- Invocation list entries produced from evaluation of semantically identical *anonymous_method_expression*s or *lambda_expression*s with the same (possibly empty) set of captured outer variable instances are permitted (but not required) to be equal.

Equality operators and null

The `==` and `!=` operators permit one operand to be a value of a nullable type and the other to be the `null` literal, even if no predefined or user-defined operator (in unlifted or lifted form) exists for the operation.

For an operation of one of the forms

```
x == null
null == x
x != null
null != x
```

where `x` is an expression of a nullable type, if operator overload resolution (Binary operator overload resolution) fails to find an applicable operator, the result is instead computed from the `HasValue` property of `x`. Specifically, the first two forms are translated into `!x.HasValue`, and last two forms are translated into `x.HasValue`.

The is operator

The `is` operator is used to dynamically check if the run-time type of an object is compatible with a given type. The result of the operation `E is T`, where `E` is an expression and `T` is a type, is a boolean value indicating whether `E` can successfully be converted to type `T` by a reference conversion, a boxing conversion, or an unboxing conversion. The operation is evaluated as follows, after type arguments have been substituted for all type parameters:

- If `E` is an anonymous function, a compile-time error occurs
- If `E` is a method group or the `null` literal, or if the type of `E` is a reference type or a nullable type and the value of `E` is null, the result is false.
- Otherwise, let `D` represent the dynamic type of `E` as follows:
  - If the type of `E` is a reference type, `D` is the run-time type of the instance reference by `E`.
  - If the type of `E` is a nullable type, `D` is the underlying type of that nullable type.
  - If the type of `E` is a non-nullable value type, `D` is the type of `E`.
- The result of the operation depends on `D` and `T` as follows:

- If `T` is a reference type, the result is true if `D` and `T` are the same type, if `D` is a reference type and an implicit reference conversion from `D` to `T` exists, or if `D` is a value type and a boxing conversion from `D` to `T` exists.
- If `T` is a nullable type, the result is true if `D` is the underlying type of `T`.
- If `T` is a non-nullable value type, the result is true if `D` and `T` are the same type.
- Otherwise, the result is false.

Note that user defined conversions, are not considered by the `is` operator.

The as operator

The `as` operator is used to explicitly convert a value to a given reference type or nullable type. Unlike a cast expression (Cast expressions), the `as` operator never throws an exception. Instead, if the indicated conversion is not possible, the resulting value is `null`.

In an operation of the form `E as T`, `E` must be an expression and `T` must be a reference type, a type parameter known to be a reference type, or a nullable type. Furthermore, at least one of the following must be true, or otherwise a compile-time error occurs:

- An identity (Identity conversion), implicit nullable (Implicit nullable conversions), implicit reference (Implicit reference conversions), boxing (Boxing conversions), explicit nullable (Explicit nullable conversions), explicit reference (Explicit reference conversions), or unboxing (Unboxing conversions) conversion exists from `E` to `T`.
- The type of `E` or `T` is an open type.
- `E` is the `null` literal.

If the compile-time type of `E` is not `dynamic`, the operation `E as T` produces the same result as

```
E is T ? (T)(E) : (T)null
```

except that `E` is only evaluated once. The compiler can be expected to optimize `E as T` to perform at most one dynamic type check as opposed to the two dynamic type checks implied by the expansion above.

If the compile-time type of `E` is `dynamic`, unlike the cast operator the `as` operator is not dynamically bound (Dynamic binding). Therefore the expansion in this case is:

```
E is T ? (T)(object)(E) : (T)null
```

Note that some conversions, such as user defined conversions, are not possible with the `as` operator and should instead be performed using cast expressions.

In the example

```
class X
{

    public string F(object o) {
        return o as string;        // OK, string is a reference type
    }

    public T G<T>(object o) where T: Attribute {
        return o as T;             // Ok, T has a class constraint
    }

    public U H<U>(object o) {
        return o as U;             // Error, U is unconstrained
    }
}
```

the type parameter `T` of `G` is known to be a reference type, because it has the class constraint. The type parameter `U` of `H` is not however; hence the use of the `as` operator in `H` is disallowed.

Logical operators

The `&`, `^`, and `|` operators are called the logical operators.

```
and_expression
    : equality_expression
    | and_expression '&' equality_expression
    ;

exclusive_or_expression
    : and_expression
    | exclusive_or_expression '^' and_expression
    ;

inclusive_or_expression
    : exclusive_or_expression
    | inclusive_or_expression '|' exclusive_or_expression
    ;
```

If an operand of a logical operator has the compile-time type `dynamic`, then the expression is dynamically bound (Dynamic binding). In this case the compile-time type of the expression is `dynamic`, and the resolution described below will take place at run-time using the run-time type of those operands that have the compile-time type `dynamic`.

For an operation of the form `x op y`, where `op` is one of the logical operators, overload resolution (Binary operator overload resolution) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined logical operators are described in the following sections.

Integer logical operators

The predefined integer logical operators are:

```
int operator &(int x, int y);
uint operator &(uint x, uint y);
long operator &(long x, long y);
ulong operator &(ulong x, ulong y);

int operator |(int x, int y);
uint operator |(uint x, uint y);
long operator |(long x, long y);
ulong operator |(ulong x, ulong y);

int operator ^(int x, int y);
uint operator ^(uint x, uint y);
long operator ^(long x, long y);
ulong operator ^(ulong x, ulong y);
```

The `&` operator computes the bitwise logical `AND` of the two operands, the `|` operator computes the bitwise logical `OR` of the two operands, and the `^` operator computes the bitwise logical exclusive `OR` of the two operands. No overflows are possible from these operations.

Enumeration logical operators

Every enumeration type `E` implicitly provides the following predefined logical operators:

```
E operator &(E x, E y);
E operator |(E x, E y);
E operator ^(E x, E y);
```

The result of evaluating `x op y`, where `x` and `y` are expressions of an enumeration type `E` with an underlying type `U`, and `op` is one of the logical operators, is exactly the same as evaluating `(E)((U)x op (U)y)`. In other words, the enumeration type logical operators simply perform the logical operation on the underlying type of the two operands.

Boolean logical operators

The predefined boolean logical operators are:

```
bool operator &(bool x, bool y);
bool operator |(bool x, bool y);
bool operator ^(bool x, bool y);
```

The result of `x & y` is `true` if both `x` and `y` are `true`. Otherwise, the result is `false`.

The result of `x | y` is `true` if either `x` or `y` is `true`. Otherwise, the result is `false`.

The result of `x ^ y` is `true` if `x` is `true` and `y` is `false`, or `x` is `false` and `y` is `true`. Otherwise, the result is `false`. When the operands are of type `bool`, the `^` operator computes the same result as the `!=` operator.

Nullable boolean logical operators

The nullable boolean type `bool?` can represent three values, `true`, `false`, and `null`, and is conceptually similar to the three-valued type used for boolean expressions in SQL. To ensure that the results produced by the `&` and `|` operators for `bool?` operands are consistent with SQL's three-valued logic, the following predefined operators are provided:

```
bool? operator &(bool? x, bool? y);
bool? operator |(bool? x, bool? y);
```

The following table lists the results produced by these operators for all combinations of the values `true`, `false`, and `null`.

`x`	`y`	`x & y`	`x | y`
`true`	`true`	`true`	`true`
`true`	`false`	`false`	`true`
`true`	`null`	`null`	`true`
`false`	`true`	`false`	`true`
`false`	`false`	`false`	`false`
`false`	`null`	`false`	`null`
`null`	`true`	`null`	`true`
`null`	`false`	`false`	`null`
`null`	`null`	`null`	`null`

Conditional logical operators

The `&&` and `||` operators are called the conditional logical operators. They are also called the "short-circuiting" logical operators.

```
conditional_and_expression
    : inclusive_or_expression
    | conditional_and_expression '&&' inclusive_or_expression
    ;

conditional_or_expression
    : conditional_and_expression
    | conditional_or_expression '||' conditional_and_expression
    ;
```

The `&&` and `||` operators are conditional versions of the `&` and `|` operators:

- The operation `x && y` corresponds to the operation `x & y`, except that `y` is evaluated only if `x` is not `false`.
- The operation `x || y` corresponds to the operation `x | y`, except that `y` is evaluated only if `x` is not `true`.

If an operand of a conditional logical operator has the compile-time type `dynamic`, then the expression is dynamically bound (Dynamic binding). In this case the compile-time type of the expression is `dynamic`, and the resolution described below will take place at run-time using the run-time type of those operands that have the compile-time type `dynamic`.

An operation of the form `x && y` or `x || y` is processed by applying overload resolution (Binary operator overload resolution) as if the operation was written `x & y` or `x | y`. Then,

- If overload resolution fails to find a single best operator, or if overload resolution selects one of the

predefined integer logical operators, a binding-time error occurs.

- Otherwise, if the selected operator is one of the predefined boolean logical operators (Boolean logical operators) or nullable boolean logical operators (Nullable boolean logical operators), the operation is processed as described in Boolean conditional logical operators.
- Otherwise, the selected operator is a user-defined operator, and the operation is processed as described in User-defined conditional logical operators.

It is not possible to directly overload the conditional logical operators. However, because the conditional logical operators are evaluated in terms of the regular logical operators, overloads of the regular logical operators are, with certain restrictions, also considered overloads of the conditional logical operators. This is described further in User-defined conditional logical operators.

Boolean conditional logical operators

When the operands of `&&` or `||` are of type `bool`, or when the operands are of types that do not define an applicable `operator &` or `operator |`, but do define implicit conversions to `bool`, the operation is processed as follows:

- The operation `x && y` is evaluated as `x ? y : false`. In other words, `x` is first evaluated and converted to type `bool`. Then, if `x` is `true`, `y` is evaluated and converted to type `bool`, and this becomes the result of the operation. Otherwise, the result of the operation is `false`.
- The operation `x || y` is evaluated as `x ? true : y`. In other words, `x` is first evaluated and converted to type `bool`. Then, if `x` is `true`, the result of the operation is `true`. Otherwise, `y` is evaluated and converted to type `bool`, and this becomes the result of the operation.

User-defined conditional logical operators

When the operands of `&&` or `||` are of types that declare an applicable user-defined `operator &` or `operator |`, both of the following must be true, where `T` is the type in which the selected operator is declared:

- The return type and the type of each parameter of the selected operator must be `T`. In other words, the operator must compute the logical `AND` or the logical `OR` of two operands of type `T`, and must return a result of type `T`.
- `T` must contain declarations of `operator true` and `operator false`.

A binding-time error occurs if either of these requirements is not satisfied. Otherwise, the `&&` or `||` operation is evaluated by combining the user-defined `operator true` or `operator false` with the selected user-defined operator:

- The operation `x && y` is evaluated as `T.false(x) ? x : T.&(x, y)`, where `T.false(x)` is an invocation of the `operator false` declared in `T`, and `T.&(x, y)` is an invocation of the selected `operator &`. In other words, `x` is first evaluated and `operator false` is invoked on the result to determine if `x` is definitely false. Then, if `x` is definitely false, the result of the operation is the value previously computed for `x`. Otherwise, `y` is evaluated, and the selected `operator &` is invoked on the value previously computed for `x` and the value computed for `y` to produce the result of the operation.
- The operation `x || y` is evaluated as `T.true(x) ? x : T.|(x, y)`, where `T.true(x)` is an invocation of the `operator true` declared in `T`, and `T.|(x,y)` is an invocation of the selected `operator|`. In other words, `x` is first evaluated and `operator true` is invoked on the result to determine if `x` is definitely true. Then, if `x` is definitely true, the result of the operation is the value previously computed for `x`. Otherwise, `y` is evaluated, and the selected `operator |` is invoked on the value previously computed for `x` and the value computed for `y` to produce the result of the operation.

In either of these operations, the expression given by `x` is only evaluated once, and the expression given by `y` is either not evaluated or evaluated exactly once.

For an example of a type that implements `operator true` and `operator false`, see Database boolean type.

The null coalescing operator

The `??` operator is called the null coalescing operator.

```
null_coalescing_expression
    : conditional_or_expression
    | conditional_or_expression '??' null_coalescing_expression
    ;
```

A null coalescing expression of the form `a ?? b` requires `a` to be of a nullable type or reference type. If `a` is non-null, the result of `a ?? b` is `a`; otherwise, the result is `b`. The operation evaluates `b` only if `a` is null.

The null coalescing operator is right-associative, meaning that operations are grouped from right to left. For example, an expression of the form `a ?? b ?? c` is evaluated as `a ?? (b ?? c)`. In general terms, an expression of the form `E1 ?? E2 ?? ... ?? En` returns the first of the operands that is non-null, or null if all operands are null.

The type of the expression `a ?? b` depends on which implicit conversions are available on the operands. In order of preference, the type of `a ?? b` is `A0`, `A`, or `B`, where `A` is the type of `a` (provided that `a` has a type), `B` is the type of `b` (provided that `b` has a type), and `A0` is the underlying type of `A` if `A` is a nullable type, or `A` otherwise. Specifically, `a ?? b` is processed as follows:

- If `A` exists and is not a nullable type or a reference type, a compile-time error occurs.
- If `b` is a dynamic expression, the result type is `dynamic`. At run-time, `a` is first evaluated. If `a` is not null, `a` is converted to dynamic, and this becomes the result. Otherwise, `b` is evaluated, and this becomes the result.
- Otherwise, if `A` exists and is a nullable type and an implicit conversion exists from `b` to `A0`, the result type is `A0`. At run-time, `a` is first evaluated. If `a` is not null, `a` is unwrapped to type `A0`, and this becomes the result. Otherwise, `b` is evaluated and converted to type `A0`, and this becomes the result.
- Otherwise, if `A` exists and an implicit conversion exists from `b` to `A`, the result type is `A`. At run-time, `a` is first evaluated. If `a` is not null, `a` becomes the result. Otherwise, `b` is evaluated and converted to type `A`, and this becomes the result.
- Otherwise, if `b` has a type `B` and an implicit conversion exists from `a` to `B`, the result type is `B`. At run-time, `a` is first evaluated. If `a` is not null, `a` is unwrapped to type `A0` (if `A` exists and is nullable) and converted to type `B`, and this becomes the result. Otherwise, `b` is evaluated and becomes the result.
- Otherwise, `a` and `b` are incompatible, and a compile-time error occurs.

Conditional operator

The `?:` operator is called the conditional operator. It is at times also called the ternary operator.

```
conditional_expression
    : null_coalescing_expression
    | null_coalescing_expression '?' expression ':' expression
    ;
```

A conditional expression of the form `b ? x : y` first evaluates the condition `b`. Then, if `b` is `true`, `x` is evaluated and becomes the result of the operation. Otherwise, `y` is evaluated and becomes the result of the operation. A conditional expression never evaluates both `x` and `y`.

The conditional operator is right-associative, meaning that operations are grouped from right to left. For

example, an expression of the form `a ? b : c ? d : e` is evaluated as `a ? b : (c ? d : e)`.

The first operand of the `?:` operator must be an expression that can be implicitly converted to `bool`, or an expression of a type that implements `operator true`. If neither of these requirements is satisfied, a compile-time error occurs.

The second and third operands, `x` and `y`, of the `?:` operator control the type of the conditional expression.

- If `x` has type `X` and `y` has type `Y` then
  - If an implicit conversion (Implicit conversions) exists from `X` to `Y`, but not from `Y` to `X`, then `Y` is the type of the conditional expression.
  - If an implicit conversion (Implicit conversions) exists from `Y` to `X`, but not from `X` to `Y`, then `X` is the type of the conditional expression.
  - Otherwise, no expression type can be determined, and a compile-time error occurs.
- If only one of `x` and `y` has a type, and both `x` and `y`, of are implicitly convertible to that type, then that is the type of the conditional expression.
- Otherwise, no expression type can be determined, and a compile-time error occurs.

The run-time processing of a conditional expression of the form `b ? x : y` consists of the following steps:

- First, `b` is evaluated, and the `bool` value of `b` is determined:
  - If an implicit conversion from the type of `b` to `bool` exists, then this implicit conversion is performed to produce a `bool` value.
  - Otherwise, the `operator true` defined by the type of `b` is invoked to produce a `bool` value.
- If the `bool` value produced by the step above is `true`, then `x` is evaluated and converted to the type of the conditional expression, and this becomes the result of the conditional expression.
- Otherwise, `y` is evaluated and converted to the type of the conditional expression, and this becomes the result of the conditional expression.

Anonymous function expressions

An ***anonymous function*** is an expression that represents an "in-line" method definition. An anonymous function does not have a value or type in and of itself, but is convertible to a compatible delegate or expression tree type. The evaluation of an anonymous function conversion depends on the target type of the conversion: If it is a delegate type, the conversion evaluates to a delegate value referencing the method which the anonymous function defines. If it is an expression tree type, the conversion evaluates to an expression tree which represents the structure of the method as an object structure.

For historical reasons there are two syntactic flavors of anonymous functions, namely *lambda_expression*s and *anonymous_method_expression*s. For almost all purposes, *lambda_expression*s are more concise and expressive than *anonymous_method_expression*s, which remain in the language for backwards compatibility.

```
lambda_expression
    : anonymous_function_signature '=>' anonymous_function_body
    ;

anonymous_method_expression
    : 'delegate' explicit_anonymous_function_signature? block
    ;

anonymous_function_signature
    : explicit_anonymous_function_signature
    | implicit_anonymous_function_signature
    ;

explicit_anonymous_function_signature
    : '(' explicit_anonymous_function_parameter_list? ')'
    ;

explicit_anonymous_function_parameter_list
    : explicit_anonymous_function_parameter (',' explicit_anonymous_function_parameter)*
    ;

explicit_anonymous_function_parameter
    : anonymous_function_parameter_modifier? type identifier
    ;

anonymous_function_parameter_modifier
    : 'ref'
    | 'out'
    ;

implicit_anonymous_function_signature
    : '(' implicit_anonymous_function_parameter_list? ')'
    | implicit_anonymous_function_parameter
    ;

implicit_anonymous_function_parameter_list
    : implicit_anonymous_function_parameter (',' implicit_anonymous_function_parameter)*
    ;

implicit_anonymous_function_parameter
    : identifier
    ;

anonymous_function_body
    : expression
    | block
    ;
```

The `=>` operator has the same precedence as assignment (`=`) and is right-associative.

An anonymous function with the `async` modifier is an async function and follows the rules described in Async functions.

The parameters of an anonymous function in the form of a *lambda_expression* can be explicitly or implicitly typed. In an explicitly typed parameter list, the type of each parameter is explicitly stated. In an implicitly typed parameter list, the types of the parameters are inferred from the context in which the anonymous function occurs—specifically, when the anonymous function is converted to a compatible delegate type or expression tree type, that type provides the parameter types (Anonymous function conversions).

In an anonymous function with a single, implicitly typed parameter, the parentheses may be omitted from the parameter list. In other words, an anonymous function of the form

```
( param ) => expr
```

can be abbreviated to

```
param => expr
```

The parameter list of an anonymous function in the form of an *anonymous_method_expression* is optional. If given, the parameters must be explicitly typed. If not, the anonymous function is convertible to a delegate with any parameter list not containing `out` parameters.

A *block* body of an anonymous function is reachable (End points and reachability) unless the anonymous function occurs inside an unreachable statement.

Some examples of anonymous functions follow below:

```
x => x + 1                           // Implicitly typed, expression body
x => { return x + 1; }               // Implicitly typed, statement body
(int x) => x + 1                     // Explicitly typed, expression body
(int x) => { return x + 1; }         // Explicitly typed, statement body
(x, y) => x * y                      // Multiple parameters
() => Console.WriteLine()            // No parameters
async (t1,t2) => await t1 + await t2 // Async
delegate (int x) { return x + 1; }   // Anonymous method expression
delegate { return 1 + 1; }           // Parameter list omitted
```

The behavior of *lambda_expression*s and *anonymous_method_expression*s is the same except for the following points:

- *anonymous_method_expression*s permit the parameter list to be omitted entirely, yielding convertibility to delegate types of any list of value parameters.
- *lambda_expression*s permit parameter types to be omitted and inferred whereas *anonymous_method_expression*s require parameter types to be explicitly stated.
- The body of a *lambda_expression* can be an expression or a statement block whereas the body of an *anonymous_method_expression* must be a statement block.
- Only *lambda_expression*s have conversions to compatible expression tree types (Expression tree types).

Anonymous function signatures

The optional *anonymous_function_signature* of an anonymous function defines the names and optionally the types of the formal parameters for the anonymous function. The scope of the parameters of the anonymous function is the *anonymous_function_body*. (Scopes) Together with the parameter list (if given) the anonymous-method-body constitutes a declaration space (Declarations). It is thus a compile-time error for the name of a parameter of the anonymous function to match the name of a local variable, local constant or parameter whose scope includes the *anonymous_method_expression* or *lambda_expression*.

If an anonymous function has an *explicit_anonymous_function_signature*, then the set of compatible delegate types and expression tree types is restricted to those that have the same parameter types and modifiers in the same order. In contrast to method group conversions (Method group conversions), contra-variance of anonymous function parameter types is not supported. If an anonymous function does not have an *anonymous_function_signature*, then the set of compatible delegate types and expression tree types is restricted to those that have no `out` parameters.

Note that an *anonymous_function_signature* cannot include attributes or a parameter array. Nevertheless, an *anonymous_function_signature* may be compatible with a delegate type whose parameter list contains a parameter array.

Note also that conversion to an expression tree type, even if compatible, may still fail at compile-time (Expression tree types).

Anonymous function bodies

The body (*expression* or *block*) of an anonymous function is subject to the following rules:

- If the anonymous function includes a signature, the parameters specified in the signature are available in the body. If the anonymous function has no signature it can be converted to a delegate type or expression type having parameters (Anonymous function conversions), but the parameters cannot be accessed in the body.
- Except for `ref` or `out` parameters specified in the signature (if any) of the nearest enclosing anonymous function, it is a compile-time error for the body to access a `ref` or `out` parameter.
- When the type of `this` is a struct type, it is a compile-time error for the body to access `this`. This is true whether the access is explicit (as in `this.x`) or implicit (as in `x` where `x` is an instance member of the struct). This rule simply prohibits such access and does not affect whether member lookup results in a member of the struct.
- The body has access to the outer variables (Outer variables) of the anonymous function. Access of an outer variable will reference the instance of the variable that is active at the time the *lambda_expression* or *anonymous_method_expression* is evaluated (Evaluation of anonymous function expressions).
- It is a compile-time error for the body to contain a `goto` statement, `break` statement, or `continue` statement whose target is outside the body or within the body of a contained anonymous function.
- A `return` statement in the body returns control from an invocation of the nearest enclosing anonymous function, not from the enclosing function member. An expression specified in a `return` statement must be implicitly convertible to the return type of the delegate type or expression tree type to which the nearest enclosing *lambda_expression* or *anonymous_method_expression* is converted (Anonymous function conversions).

It is explicitly unspecified whether there is any way to execute the block of an anonymous function other than through evaluation and invocation of the *lambda_expression* or *anonymous_method_expression*. In particular, the compiler may choose to implement an anonymous function by synthesizing one or more named methods or types. The names of any such synthesized elements must be of a form reserved for compiler use.

Overload resolution and anonymous functions

Anonymous functions in an argument list participate in type inference and overload resolution. Please refer to Type inference and Overload resolution for the exact rules.

The following example illustrates the effect of anonymous functions on overload resolution.

```
class ItemList<T>: List<T>
{
    public int Sum(Func<T,int> selector) {
        int sum = 0;
        foreach (T item in this) sum += selector(item);
        return sum;
    }

    public double Sum(Func<T,double> selector) {
        double sum = 0;
        foreach (T item in this) sum += selector(item);
        return sum;
    }
}
```

The `ItemList<T>` class has two `Sum` methods. Each takes a `selector` argument, which extracts the value to sum over from a list item. The extracted value can be either an `int` or a `double` and the resulting sum is likewise either an `int` or a `double`.

The `Sum` methods could for example be used to compute sums from a list of detail lines in an order.

```
class Detail
{
    public int UnitCount;
    public double UnitPrice;
    ...
}

void ComputeSums() {
    ItemList<Detail> orderDetails = GetOrderDetails(...);
    int totalUnits = orderDetails.Sum(d => d.UnitCount);
    double orderTotal = orderDetails.Sum(d => d.UnitPrice * d.UnitCount);
    ...
}
```

In the first invocation of `orderDetails.Sum`, both `Sum` methods are applicable because the anonymous function `d => d. UnitCount` is compatible with both `Func<Detail,int>` and `Func<Detail,double>`. However, overload resolution picks the first `Sum` method because the conversion to `Func<Detail,int>` is better than the conversion to `Func<Detail,double>`.

In the second invocation of `orderDetails.Sum`, only the second `Sum` method is applicable because the anonymous function `d => d.UnitPrice * d.UnitCount` produces a value of type `double`. Thus, overload resolution picks the second `Sum` method for that invocation.

Anonymous functions and dynamic binding

An anonymous function cannot be a receiver, argument or operand of a dynamically bound operation.

Outer variables

Any local variable, value parameter, or parameter array whose scope includes the *lambda_expression* or *anonymous_method_expression* is called an ***outer variable*** of the anonymous function. In an instance function member of a class, the `this` value is considered a value parameter and is an outer variable of any anonymous function contained within the function member.

Captured outer variables

When an outer variable is referenced by an anonymous function, the outer variable is said to have been ***captured*** by the anonymous function. Ordinarily, the lifetime of a local variable is limited to execution of the block or statement with which it is associated (Local variables). However, the lifetime of a captured outer variable is extended at least until the delegate or expression tree created from the anonymous function becomes eligible for garbage collection.

In the example

```
using System;

delegate int D();

class Test
{
    static D F() {
        int x = 0;
        D result = () => ++x;
        return result;
    }

    static void Main() {
        D d = F();
        Console.WriteLine(d());
        Console.WriteLine(d());
        Console.WriteLine(d());
    }
}
```

the local variable `x` is captured by the anonymous function, and the lifetime of `x` is extended at least until the delegate returned from `F` becomes eligible for garbage collection (which doesn't happen until the very end of the program). Since each invocation of the anonymous function operates on the same instance of `x`, the output of the example is:

```
1
2
3
```

When a local variable or a value parameter is captured by an anonymous function, the local variable or parameter is no longer considered to be a fixed variable (Fixed and moveable variables), but is instead considered to be a moveable variable. Thus any `unsafe` code that takes the address of a captured outer variable must first use the `fixed` statement to fix the variable.

Note that unlike an uncaptured variable, a captured local variable can be simultaneously exposed to multiple threads of execution.

Instantiation of local variables

A local variable is considered to be *instantiated* when execution enters the scope of the variable. For example, when the following method is invoked, the local variable `x` is instantiated and initialized three times—once for each iteration of the loop.

```
static void F() {
    for (int i = 0; i < 3; i++) {
        int x = i * 2 + 1;
        ...
    }
}
```

However, moving the declaration of `x` outside the loop results in a single instantiation of `x`:

```
static void F() {
    int x;
    for (int i = 0; i < 3; i++) {
        x = i * 2 + 1;
        ...
    }
}
```

When not captured, there is no way to observe exactly how often a local variable is instantiated—because the lifetimes of the instantiations are disjoint, it is possible for each instantiation to simply use the same storage location. However, when an anonymous function captures a local variable, the effects of instantiation become apparent.

The example

```
using System;

delegate void D();

class Test
{
    static D[] F() {
        D[] result = new D[3];
        for (int i = 0; i < 3; i++) {
            int x = i * 2 + 1;
            result[i] = () => { Console.WriteLine(x); };
        }
        return result;
    }

    static void Main() {
        foreach (D d in F()) d();
    }
}
```

produces the output:

```
1
3
5
```

However, when the declaration of `x` is moved outside the loop:

```
static D[] F() {
    D[] result = new D[3];
    int x;
    for (int i = 0; i < 3; i++) {
        x = i * 2 + 1;
        result[i] = () => { Console.WriteLine(x); };
    }
    return result;
}
```

the output is:

```
5
5
5
```

If a for-loop declares an iteration variable, that variable itself is considered to be declared outside of the loop. Thus, if the example is changed to capture the iteration variable itself:

```
static D[] F() {
    D[] result = new D[3];
    for (int i = 0; i < 3; i++) {
        result[i] = () => { Console.WriteLine(i); };
    }
    return result;
}
```

only one instance of the iteration variable is captured, which produces the output:

```
3
3
3
```

It is possible for anonymous function delegates to share some captured variables yet have separate instances of others. For example, if `F` is changed to

```
static D[] F() {
    D[] result = new D[3];
    int x = 0;
    for (int i = 0; i < 3; i++) {
        int y = 0;
        result[i] = () => { Console.WriteLine("{0} {1}", ++x, ++y); };
    }
    return result;
}
```

the three delegates capture the same instance of `x` but separate instances of `y`, and the output is:

```
1 1
2 1
3 1
```

Separate anonymous functions can capture the same instance of an outer variable. In the example:

```
using System;

delegate void Setter(int value);

delegate int Getter();

class Test
{
    static void Main() {
        int x = 0;
        Setter s = (int value) => { x = value; };
        Getter g = () => { return x; };
        s(5);
        Console.WriteLine(g());
        s(10);
        Console.WriteLine(g());
    }
}
```

the two anonymous functions capture the same instance of the local variable `x`, and they can thus

"communicate" through that variable. The output of the example is:

```
5
10
```

Evaluation of anonymous function expressions

An anonymous function `F` must always be converted to a delegate type `D` or an expression tree type `E`, either directly or through the execution of a delegate creation expression `new D(F)`. This conversion determines the result of the anonymous function, as described in Anonymous function conversions.

Query expressions

Query expressions provide a language integrated syntax for queries that is similar to relational and hierarchical query languages such as SQL and XQuery.

```
query_expression
    : from_clause query_body
    ;

from_clause
    : 'from' type? identifier 'in' expression
    ;

query_body
    : query_body_clauses? select_or_group_clause query_continuation?
    ;

query_body_clauses
    : query_body_clause
    | query_body_clauses query_body_clause
    ;

query_body_clause
    : from_clause
    | let_clause
    | where_clause
    | join_clause
    | join_into_clause
    | orderby_clause
    ;

let_clause
    : 'let' identifier '=' expression
    ;

where_clause
    : 'where' boolean_expression
    ;

join_clause
    : 'join' type? identifier 'in' expression 'on' expression 'equals' expression
    ;

join_into_clause
    : 'join' type? identifier 'in' expression 'on' expression 'equals' expression 'into' identifier
    ;

orderby_clause
    : 'orderby' orderings
    ;

orderings
    : ordering (',' ordering)*
```

```
    ;

ordering
    : expression ordering_direction?
    ;

ordering_direction
    : 'ascending'
    | 'descending'
    ;

select_or_group_clause
    : select_clause
    | group_clause
    ;

select_clause
    : 'select' expression
    ;

group_clause
    : 'group' expression 'by' expression
    ;

query_continuation
    : 'into' identifier query_body
    ;
```

A query expression begins with a `from` clause and ends with either a `select` or `group` clause. The initial `from` clause can be followed by zero or more `from`, `let`, `where`, `join` or `orderby` clauses. Each `from` clause is a generator introducing a ***range variable*** which ranges over the elements of a ***sequence***. Each `let` clause introduces a range variable representing a value computed by means of previous range variables. Each `where` clause is a filter that excludes items from the result. Each `join` clause compares specified keys of the source sequence with keys of another sequence, yielding matching pairs. Each `orderby` clause reorders items according to specified criteria.The final `select` or `group` clause specifies the shape of the result in terms of the range variables. Finally, an `into` clause can be used to "splice" queries by treating the results of one query as a generator in a subsequent query.

Ambiguities in query expressions

Query expressions contain a number of "contextual keywords", i.e., identifiers that have special meaning in a given context. Specifically these are `from`, `where`, `join`, `on`, `equals`, `into`, `let`, `orderby`, `ascending`, `descending`, `select`, `group` and `by`. In order to avoid ambiguities in query expressions caused by mixed use of these identifiers as keywords or simple names, these identifiers are considered keywords when occurring anywhere within a query expression.

For this purpose, a query expression is any expression that starts with "`from identifier`" followed by any token except "`;`", "`=`" or "`,`".

In order to use these words as identifiers within a query expression, they can be prefixed with "`@`" (Identifiers).

Query expression translation

The C# language does not specify the execution semantics of query expressions. Rather, query expressions are translated into invocations of methods that adhere to the *query expression pattern* (The query expression pattern). Specifically, query expressions are translated into invocations of methods named `Where`, `Select`, `SelectMany`, `Join`, `GroupJoin`, `OrderBy`, `OrderByDescending`, `ThenBy`, `ThenByDescending`, `GroupBy`, and `Cast` .These methods are expected to have particular signatures and result types, as described in The query expression pattern. These methods can be instance methods of the object being queried or extension methods that are external to the object, and they implement the actual execution of the query.

The translation from query expressions to method invocations is a syntactic mapping that occurs before any

type binding or overload resolution has been performed. The translation is guaranteed to be syntactically correct, but it is not guaranteed to produce semantically correct C# code. Following translation of query expressions, the resulting method invocations are processed as regular method invocations, and this may in turn uncover errors, for example if the methods do not exist, if arguments have wrong types, or if the methods are generic and type inference fails.

A query expression is processed by repeatedly applying the following translations until no further reductions are possible. The translations are listed in order of application: each section assumes that the translations in the preceding sections have been performed exhaustively, and once exhausted, a section will not later be revisited in the processing of the same query expression.

Assignment to range variables is not allowed in query expressions. However a C# implementation is permitted to not always enforce this restriction, since this may sometimes not be possible with the syntactic translation scheme presented here.

Certain translations inject range variables with transparent identifiers denoted by `*`. The special properties of transparent identifiers are discussed further in Transparent identifiers.

Select and groupby clauses with continuations

A query expression with a continuation

```
from ... into x ...
```

is translated into

```
from x in ( from ... ) ...
```

The translations in the following sections assume that queries have no `into` continuations.

The example

```
from c in customers
group c by c.Country into g
select new { Country = g.Key, CustCount = g.Count() }
```

is translated into

```
from g in
    from c in customers
    group c by c.Country
select new { Country = g.Key, CustCount = g.Count() }
```

the final translation of which is

```
customers.
GroupBy(c => c.Country).
Select(g => new { Country = g.Key, CustCount = g.Count() })
```

Explicit range variable types

A `from` clause that explicitly specifies a range variable type

```
from T x in e
```

is translated into

```
from x in ( e ) . Cast < T > ( )
```

A `join` clause that explicitly specifies a range variable type

```
join T x in e on k1 equals k2
```

is translated into

```
join x in ( e ) . Cast < T > ( ) on k1 equals k2
```

The translations in the following sections assume that queries have no explicit range variable types.

The example

```
from Customer c in customers
where c.City == "London"
select c
```

is translated into

```
from c in customers.Cast<Customer>()
where c.City == "London"
select c
```

the final translation of which is

```
customers.
Cast<Customer>().
Where(c => c.City == "London")
```

Explicit range variable types are useful for querying collections that implement the non-generic `IEnumerable` interface, but not the generic `IEnumerable<T>` interface. In the example above, this would be the case if `customers` were of type `ArrayList`.

Degenerate query expressions

A query expression of the form

```
from x in e select x
```

is translated into

```
( e ) . Select ( x => x )
```

The example

```
from c in customers
select c
```

is translated into

```
customers.Select(c => c)
```

A degenerate query expression is one that trivially selects the elements of the source. A later phase of the translation removes degenerate queries introduced by other translation steps by replacing them with their source. It is important however to ensure that the result of a query expression is never the source object itself, as that would reveal the type and identity of the source to the client of the query. Therefore this step protects degenerate queries written directly in source code by explicitly calling `Select` on the source. It is then up to the implementers of `Select` and other query operators to ensure that these methods never return the source object itself.

From, let, where, join and orderby clauses

A query expression with a second `from` clause followed by a `select` clause

```
from x1 in e1
from x2 in e2
select v
```

is translated into

```
( e1 ) . SelectMany( x1 => e2 , ( x1 , x2 ) => v )
```

A query expression with a second `from` clause followed by something other than a `select` clause:

```
from x1 in e1
from x2 in e2
...
```

is translated into

```
from * in ( e1 ) . SelectMany( x1 => e2 , ( x1 , x2 ) => new { x1 , x2 } )
...
```

A query expression with a `let` clause

```
from x in e
let y = f
...
```

is translated into

```
from * in ( e ) . Select ( x => new { x , y = f } )
...
```

A query expression with a `where` clause

```
from x in e
where f
...
```

is translated into

```
from x in ( e ) . Where ( x => f )
...
```

A query expression with a `join` clause without an `into` followed by a `select` clause

```
from x1 in e1
join x2 in e2 on k1 equals k2
select v
```

is translated into

```
( e1 ) . Join( e2 , x1 => k1 , x2 => k2 , ( x1 , x2 ) => v )
```

A query expression with a `join` clause without an `into` followed by something other than a `select` clause

```
from x1 in e1
join x2 in e2 on k1 equals k2
...
```

is translated into

```
from * in ( e1 ) . Join( e2 , x1 => k1 , x2 => k2 , ( x1 , x2 ) => new { x1 , x2 })
...
```

A query expression with a `join` clause with an `into` followed by a `select` clause

```
from x1 in e1
join x2 in e2 on k1 equals k2 into g
select v
```

is translated into

```
( e1 ) . GroupJoin( e2 , x1 => k1 , x2 => k2 , ( x1 , g ) => v )
```

A query expression with a `join` clause with an `into` followed by something other than a `select` clause

```
from x1 in e1
join x2 in e2 on k1 equals k2 into g
...
```

is translated into

```
from * in ( e1 ) . GroupJoin( e2 , x1 => k1 , x2 => k2 , ( x1 , g ) => new { x1 , g })
...
```

A query expression with an `orderby` clause

```
from x in e
orderby k1 , k2 , ..., kn
...
```

is translated into

```
from x in ( e ) .
OrderBy ( x => k1 ) .
ThenBy ( x => k2 ) .
... .
ThenBy ( x => kn )
...
```

If an ordering clause specifies a `descending` direction indicator, an invocation of `OrderByDescending` or `ThenByDescending` is produced instead.

The following translations assume that there are no `let`, `where`, `join` or `orderby` clauses, and no more than the one initial `from` clause in each query expression.

The example

```
from c in customers
from o in c.Orders
select new { c.Name, o.OrderID, o.Total }
```

is translated into

```
customers.
SelectMany(c => c.Orders,
     (c,o) => new { c.Name, o.OrderID, o.Total }
)
```

The example

```
from c in customers
from o in c.Orders
orderby o.Total descending
select new { c.Name, o.OrderID, o.Total }
```

is translated into

```
from * in customers.
    SelectMany(c => c.Orders, (c,o) => new { c, o })
orderby o.Total descending
select new { c.Name, o.OrderID, o.Total }
```

the final translation of which is

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(x => x.o.Total).
Select(x => new { x.c.Name, x.o.OrderID, x.o.Total })
```

where `x` is a compiler generated identifier that is otherwise invisible and inaccessible.

The example

```
from o in orders
let t = o.Details.Sum(d => d.UnitPrice * d.Quantity)
where t >= 1000
select new { o.OrderID, Total = t }
```

is translated into

```
from * in orders.
    Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice * d.Quantity) })
where t >= 1000
select new { o.OrderID, Total = t }
```

the final translation of which is

```
orders.
Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice * d.Quantity) }).
Where(x => x.t >= 1000).
Select(x => new { x.o.OrderID, Total = x.t })
```

where `x` is a compiler generated identifier that is otherwise invisible and inaccessible.

The example

```
from c in customers
join o in orders on c.CustomerID equals o.CustomerID
select new { c.Name, o.OrderDate, o.Total }
```

is translated into

```
customers.Join(orders, c => c.CustomerID, o => o.CustomerID,
    (c, o) => new { c.Name, o.OrderDate, o.Total })
```

The example

```
from c in customers
join o in orders on c.CustomerID equals o.CustomerID into co
let n = co.Count()
where n >= 10
select new { c.Name, OrderCount = n }
```

is translated into

```
from * in customers.
    GroupJoin(orders, c => c.CustomerID, o => o.CustomerID,
        (c, co) => new { c, co })
let n = co.Count()
where n >= 10
select new { c.Name, OrderCount = n }
```

the final translation of which is

```
customers.
GroupJoin(orders, c => c.CustomerID, o => o.CustomerID,
    (c, co) => new { c, co }).
Select(x => new { x, n = x.co.Count() }).
Where(y => y.n >= 10).
Select(y => new { y.x.c.Name, OrderCount = y.n })
```

where `x` and `y` are compiler generated identifiers that are otherwise invisible and inaccessible.

The example

```
from o in orders
orderby o.Customer.Name, o.Total descending
select o
```

has the final translation

```
orders.
OrderBy(o => o.Customer.Name).
ThenByDescending(o => o.Total)
```

Select clauses

A query expression of the form

```
from x in e select v
```

is translated into

```
( e ) . Select ( x => v )
```

except when v is the identifier *x*, the translation is simply

```
( e )
```

For example

```
from c in customers.Where(c => c.City == "London")
select c
```

is simply translated into

```
customers.Where(c => c.City == "London")
```

Groupby clauses

A query expression of the form

```
from x in e group v by k
```

is translated into

```
( e ) . GroupBy ( x => k , x => v )
```

except when v is the identifier x, the translation is

```
( e ) . GroupBy ( x => k )
```

The example

```
from c in customers
group c.Name by c.Country
```

is translated into

```
customers.
GroupBy(c => c.Country, c => c.Name)
```

Transparent identifiers

Certain translations inject range variables with ***transparent identifiers*** denoted by `*`. Transparent identifiers are not a proper language feature; they exist only as an intermediate step in the query expression translation process.

When a query translation injects a transparent identifier, further translation steps propagate the transparent identifier into anonymous functions and anonymous object initializers. In those contexts, transparent identifiers have the following behavior:

- When a transparent identifier occurs as a parameter in an anonymous function, the members of the associated anonymous type are automatically in scope in the body of the anonymous function.
- When a member with a transparent identifier is in scope, the members of that member are in scope as well.
- When a transparent identifier occurs as a member declarator in an anonymous object initializer, it introduces a member with a transparent identifier.
- In the translation steps described above, transparent identifiers are always introduced together with anonymous types, with the intent of capturing multiple range variables as members of a single object. An implementation of C# is permitted to use a different mechanism than anonymous types to group together multiple range variables. The following translation examples assume that anonymous types are used, and show how transparent identifiers can be translated away.

The example

```
from c in customers
from o in c.Orders
orderby o.Total descending
select new { c.Name, o.Total }
```

is translated into

```
from * in customers.
    SelectMany(c => c.Orders, (c,o) => new { c, o })
orderby o.Total descending
select new { c.Name, o.Total }
```

which is further translated into

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(* => o.Total).
Select(* => new { c.Name, o.Total })
```

which, when transparent identifiers are erased, is equivalent to

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(x => x.o.Total).
Select(x => new { x.c.Name, x.o.Total })
```

where `x` is a compiler generated identifier that is otherwise invisible and inaccessible.

The example

```
from c in customers
join o in orders on c.CustomerID equals o.CustomerID
join d in details on o.OrderID equals d.OrderID
join p in products on d.ProductID equals p.ProductID
select new { c.Name, o.OrderDate, p.ProductName }
```

is translated into

```
from * in customers.
    Join(orders, c => c.CustomerID, o => o.CustomerID,
        (c, o) => new { c, o })
join d in details on o.OrderID equals d.OrderID
join p in products on d.ProductID equals p.ProductID
select new { c.Name, o.OrderDate, p.ProductName }
```

which is further reduced to

```
customers.
Join(orders, c => c.CustomerID, o => o.CustomerID, (c, o) => new { c, o }).
Join(details, * => o.OrderID, d => d.OrderID, (*, d) => new { *, d }).
Join(products, * => d.ProductID, p => p.ProductID, (*, p) => new { *, p }).
Select(* => new { c.Name, o.OrderDate, p.ProductName })
```

the final translation of which is

```
customers.
Join(orders, c => c.CustomerID, o => o.CustomerID,
    (c, o) => new { c, o }).
Join(details, x => x.o.OrderID, d => d.OrderID,
    (x, d) => new { x, d }).
Join(products, y => y.d.ProductID, p => p.ProductID,
    (y, p) => new { y, p }).
Select(z => new { z.y.x.c.Name, z.y.x.o.OrderDate, z.p.ProductName })
```

where `x`, `y`, and `z` are compiler generated identifiers that are otherwise invisible and inaccessible.

The query expression pattern

The *Query expression pattern* establishes a pattern of methods that types can implement to support query expressions. Because query expressions are translated to method invocations by means of a syntactic mapping, types have considerable flexibility in how they implement the query expression pattern. For example, the

methods of the pattern can be implemented as instance methods or as extension methods because the two have the same invocation syntax, and the methods can request delegates or expression trees because anonymous functions are convertible to both.

The recommended shape of a generic type `C<T>` that supports the query expression pattern is shown below. A generic type is used in order to illustrate the proper relationships between parameter and result types, but it is possible to implement the pattern for non-generic types as well.

```
delegate R Func<T1,R>(T1 arg1);

delegate R Func<T1,T2,R>(T1 arg1, T2 arg2);

class C
{
    public C<T> Cast<T>();
}

class C<T> : C
{
    public C<T> Where(Func<T,bool> predicate);

    public C<U> Select<U>(Func<T,U> selector);

    public C<V> SelectMany<U,V>(Func<T,C<U>> selector,
        Func<T,U,V> resultSelector);

    public C<V> Join<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,
        Func<U,K> innerKeySelector, Func<T,U,V> resultSelector);

    public C<V> GroupJoin<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,
        Func<U,K> innerKeySelector, Func<T,C<U>,V> resultSelector);

    public O<T> OrderBy<K>(Func<T,K> keySelector);

    public O<T> OrderByDescending<K>(Func<T,K> keySelector);

    public C<G<K,T>> GroupBy<K>(Func<T,K> keySelector);

    public C<G<K,E>> GroupBy<K,E>(Func<T,K> keySelector,
        Func<T,E> elementSelector);
}

class O<T> : C<T>
{
    public O<T> ThenBy<K>(Func<T,K> keySelector);

    public O<T> ThenByDescending<K>(Func<T,K> keySelector);
}

class G<K,T> : C<T>
{
    public K Key { get; }
}
```

The methods above use the generic delegate types `Func<T1,R>` and `Func<T1,T2,R>`, but they could equally well have used other delegate or expression tree types with the same relationships in parameter and result types.

Notice the recommended relationship between `C<T>` and `O<T>` which ensures that the `ThenBy` and `ThenByDescending` methods are available only on the result of an `OrderBy` or `OrderByDescending`. Also notice the recommended shape of the result of `GroupBy` -- a sequence of sequences, where each inner sequence has an additional `Key` property.

The `System.Linq` namespace provides an implementation of the query operator pattern for any type that

implements the `System.Collections.Generic.IEnumerable<T>` interface.

Assignment operators

The assignment operators assign a new value to a variable, a property, an event, or an indexer element.

```
assignment
    : unary_expression assignment_operator expression
    ;

assignment_operator
    : '='
    | '+='
    | '-='
    | '*='
    | '/='
    | '%='
    | '&='
    | '|='
    | '^='
    | '<<='
    | right_shift_assignment
    ;
```

The left operand of an assignment must be an expression classified as a variable, a property access, an indexer access, or an event access.

The `=` operator is called the ***simple assignment operator***. It assigns the value of the right operand to the variable, property, or indexer element given by the left operand. The left operand of the simple assignment operator may not be an event access (except as described in Field-like events). The simple assignment operator is described in Simple assignment.

The assignment operators other than the `=` operator are called the ***compound assignment operators***. These operators perform the indicated operation on the two operands, and then assign the resulting value to the variable, property, or indexer element given by the left operand. The compound assignment operators are described in Compound assignment.

The `+=` and `-=` operators with an event access expression as the left operand are called the *event assignment operators*. No other assignment operator is valid with an event access as the left operand. The event assignment operators are described in Event assignment.

The assignment operators are right-associative, meaning that operations are grouped from right to left. For example, an expression of the form `a = b = c` is evaluated as `a = (b = c)`.

Simple assignment

The `=` operator is called the simple assignment operator.

If the left operand of a simple assignment is of the form `E.P` or `E[Ei]` where `E` has the compile-time type `dynamic`, then the assignment is dynamically bound (Dynamic binding). In this case the compile-time type of the assignment expression is `dynamic`, and the resolution described below will take place at run-time based on the run-time type of `E`.

In a simple assignment, the right operand must be an expression that is implicitly convertible to the type of the left operand. The operation assigns the value of the right operand to the variable, property, or indexer element given by the left operand.

The result of a simple assignment expression is the value assigned to the left operand. The result has the same type as the left operand and is always classified as a value.

If the left operand is a property or indexer access, the property or indexer must have a `set` accessor. If this is not the case, a binding-time error occurs.

The run-time processing of a simple assignment of the form `x = y` consists of the following steps:

- If `x` is classified as a variable:
  - `x` is evaluated to produce the variable.
  - `y` is evaluated and, if required, converted to the type of `x` through an implicit conversion (Implicit conversions).
  - If the variable given by `x` is an array element of a *reference_type*, a run-time check is performed to ensure that the value computed for `y` is compatible with the array instance of which `x` is an element. The check succeeds if `y` is `null`, or if an implicit reference conversion (Implicit reference conversions) exists from the actual type of the instance referenced by `y` to the actual element type of the array instance containing `x`. Otherwise, a `System.ArrayTypeMismatchException` is thrown.
  - The value resulting from the evaluation and conversion of `y` is stored into the location given by the evaluation of `x`.
- If `x` is classified as a property or indexer access:
  - The instance expression (if `x` is not `static`) and the argument list (if `x` is an indexer access) associated with `x` are evaluated, and the results are used in the subsequent `set` accessor invocation.
  - `y` is evaluated and, if required, converted to the type of `x` through an implicit conversion (Implicit conversions).
  - The `set` accessor of `x` is invoked with the value computed for `y` as its `value` argument.

The array co-variance rules (Array covariance) permit a value of an array type `A[]` to be a reference to an instance of an array type `B[]`, provided an implicit reference conversion exists from `B` to `A`. Because of these rules, assignment to an array element of a *reference_type* requires a run-time check to ensure that the value being assigned is compatible with the array instance. In the example

```
string[] sa = new string[10];
object[] oa = sa;

oa[0] = null;               // Ok
oa[1] = "Hello";            // Ok
oa[2] = new ArrayList();    // ArrayTypeMismatchException
```

the last assignment causes a `System.ArrayTypeMismatchException` to be thrown because an instance of `ArrayList` cannot be stored in an element of a `string[]`.

When a property or indexer declared in a *struct_type* is the target of an assignment, the instance expression associated with the property or indexer access must be classified as a variable. If the instance expression is classified as a value, a binding-time error occurs. Because of Member access, the same rule also applies to fields.

Given the declarations:

```
struct Point
{
    int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int X {
        get { return x; }
        set { x = value; }
    }

    public int Y {
        get { return y; }
        set { y = value; }
    }
}

struct Rectangle
{
    Point a, b;

    public Rectangle(Point a, Point b) {
        this.a = a;
        this.b = b;
    }

    public Point A {
        get { return a; }
        set { a = value; }
    }

    public Point B {
        get { return b; }
        set { b = value; }
    }
}
```

in the example

```
Point p = new Point();
p.X = 100;
p.Y = 100;
Rectangle r = new Rectangle();
r.A = new Point(10, 10);
r.B = p;
```

the assignments to `p.X`, `p.Y`, `r.A`, and `r.B` are permitted because `p` and `r` are variables. However, in the example

```
Rectangle r = new Rectangle();
r.A.X = 10;
r.A.Y = 10;
r.B.X = 100;
r.B.Y = 100;
```

the assignments are all invalid, since `r.A` and `r.B` are not variables.

Compound assignment

If the left operand of a compound assignment is of the form `E.P` or `E[Ei]` where `E` has the compile-time type

`dynamic`, then the assignment is dynamically bound (Dynamic binding). In this case the compile-time type of the assignment expression is `dynamic`, and the resolution described below will take place at run-time based on the run-time type of `E`.

An operation of the form `x op= y` is processed by applying binary operator overload resolution (Binary operator overload resolution) as if the operation was written `x op y`. Then,

- If the return type of the selected operator is implicitly convertible to the type of `x`, the operation is evaluated as `x = x op y`, except that `x` is evaluated only once.
- Otherwise, if the selected operator is a predefined operator, if the return type of the selected operator is explicitly convertible to the type of `x`, and if `y` is implicitly convertible to the type of `x` or the operator is a shift operator, then the operation is evaluated as `x = (T)(x op y)`, where `T` is the type of `x`, except that `x` is evaluated only once.
- Otherwise, the compound assignment is invalid, and a binding-time error occurs.

The term "evaluated only once" means that in the evaluation of `x op y`, the results of any constituent expressions of `x` are temporarily saved and then reused when performing the assignment to `x`. For example, in the assignment `A()[B()] += C()`, where `A` is a method returning `int[]`, and `B` and `C` are methods returning `int`, the methods are invoked only once, in the order `A`, `B`, `C`.

When the left operand of a compound assignment is a property access or indexer access, the property or indexer must have both a `get` accessor and a `set` accessor. If this is not the case, a binding-time error occurs.

The second rule above permits `x op= y` to be evaluated as `x = (T)(x op y)` in certain contexts. The rule exists such that the predefined operators can be used as compound operators when the left operand is of type `sbyte`, `byte`, `short`, `ushort`, or `char`. Even when both arguments are of one of those types, the predefined operators produce a result of type `int`, as described in Binary numeric promotions. Thus, without a cast it would not be possible to assign the result to the left operand.

The intuitive effect of the rule for predefined operators is simply that `x op= y` is permitted if both of `x op y` and `x = y` are permitted. In the example

```
byte b = 0;
char ch = '\0';
int i = 0;

b += 1;             // Ok
b += 1000;          // Error, b = 1000 not permitted
b += i;             // Error, b = i not permitted
b += (byte)i;       // Ok

ch += 1;            // Error, ch = 1 not permitted
ch += (char)1;      // Ok
```

the intuitive reason for each error is that a corresponding simple assignment would also have been an error.

This also means that compound assignment operations support lifted operations. In the example

```
int? i = 0;
i += 1;             // Ok
```

the lifted operator `+(int?,int?)` is used.

Event assignment

If the left operand of a `+=` or `-=` operator is classified as an event access, then the expression is evaluated as follows:

- The instance expression, if any, of the event access is evaluated.
- The right operand of the `+=` or `-=` operator is evaluated, and, if required, converted to the type of the left operand through an implicit conversion (Implicit conversions).
- An event accessor of the event is invoked, with argument list consisting of the right operand, after evaluation and, if necessary, conversion. If the operator was `+=`, the `add` accessor is invoked; if the operator was `-=`, the `remove` accessor is invoked.

An event assignment expression does not yield a value. Thus, an event assignment expression is valid only in the context of a *statement_expression* (Expression statements).

Expression

An *expression* is either a *non_assignment_expression* or an *assignment*.

```
expression
    : non_assignment_expression
    | assignment
    ;

non_assignment_expression
    : conditional_expression
    | lambda_expression
    | query_expression
    ;
```

Constant expressions

A *constant_expression* is an expression that can be fully evaluated at compile-time.

```
constant_expression
    : expression
    ;
```

A constant expression must be the `null` literal or a value with one of the following types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, `object`, `string`, or any enumeration type. Only the following constructs are permitted in constant expressions:

- Literals (including the `null` literal).
- References to `const` members of class and struct types.
- References to members of enumeration types.
- References to `const` parameters or local variables
- Parenthesized sub-expressions, which are themselves constant expressions.
- Cast expressions, provided the target type is one of the types listed above.
- `checked` and `unchecked` expressions
- Default value expressions
- Nameof expressions
- The predefined `+`, `-`, `!`, and `~` unary operators.
- The predefined `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `|`, `^`, `&&`, `||`, `==`, `!=`, `<`, `>`, `<=`, and `>=` binary operators, provided each operand is of a type listed above.
- The `?:` conditional operator.

The following conversions are permitted in constant expressions:

- Identity conversions
- Numeric conversions
- Enumeration conversions
- Constant expression conversions
- Implicit and explicit reference conversions, provided that the source of the conversions is a constant expression that evaluates to the null value.

Other conversions including boxing, unboxing and implicit reference conversions of non-null values are not permitted in constant expressions. For example:

```
class C {
    const object i = 5;         // error: boxing conversion not permitted
    const object str = "hello"; // error: implicit reference conversion
}
```

the initialization of i is an error because a boxing conversion is required. The initialization of str is an error because an implicit reference conversion from a non-null value is required.

Whenever an expression fulfills the requirements listed above, the expression is evaluated at compile-time. This is true even if the expression is a sub-expression of a larger expression that contains non-constant constructs.

The compile-time evaluation of constant expressions uses the same rules as run-time evaluation of non-constant expressions, except that where run-time evaluation would have thrown an exception, compile-time evaluation causes a compile-time error to occur.

Unless a constant expression is explicitly placed in an `unchecked` context, overflows that occur in integral-type arithmetic operations and conversions during the compile-time evaluation of the expression always cause compile-time errors (Constant expressions).

Constant expressions occur in the contexts listed below. In these contexts, a compile-time error occurs if an expression cannot be fully evaluated at compile-time.

- Constant declarations (Constants).
- Enumeration member declarations (Enum members).
- Default arguments of formal parameter lists (Method parameters)
- `case` labels of a `switch` statement (The switch statement).
- `goto case` statements (The goto statement).
- Dimension lengths in an array creation expression (Array creation expressions) that includes an initializer.
- Attributes (Attributes).

An implicit constant expression conversion (Implicit constant expression conversions) permits a constant expression of type `int` to be converted to `sbyte`, `byte`, `short`, `ushort`, `uint`, or `ulong`, provided the value of the constant expression is within the range of the destination type.

Boolean expressions

A *boolean_expression* is an expression that yields a result of type `bool`; either directly or through application of `operator true` in certain contexts as specified in the following.

```
boolean_expression
    : expression
    ;
```

The controlling conditional expression of an *if_statement* (The if statement), *while_statement* (The while

statement), *do_statement* (The do statement), or *for_statement* (The for statement) is a *boolean_expression*. The controlling conditional expression of the `?:` operator (Conditional operator) follows the same rules as a *boolean_expression*, but for reasons of operator precedence is classified as a *conditional_or_expression*.

A *boolean_expression* `E` is required to be able to produce a value of type `bool`, as follows:

- If `E` is implicitly convertible to `bool` then at runtime that implicit conversion is applied.
- Otherwise, unary operator overload resolution (Unary operator overload resolution) is used to find a unique best implementation of operator `true` on `E`, and that implementation is applied at runtime.
- If no such operator is found, a binding-time error occurs.

The `DBBool` struct type in Database boolean type provides an example of a type that implements `operator true` and `operator false`.

Statements

18.12.2021 • 54 minutes to read

C# provides a variety of statements. Most of these statements will be familiar to developers who have programmed in C and C++.

```
statement
    : labeled_statement
    | declaration_statement
    | embedded_statement
    ;

embedded_statement
    : block
    | empty_statement
    | expression_statement
    | selection_statement
    | iteration_statement
    | jump_statement
    | try_statement
    | checked_statement
    | unchecked_statement
    | lock_statement
    | using_statement
    | yield_statement
    | embedded_statement_unsafe
    ;
```

The *embedded_statement* nonterminal is used for statements that appear within other statements. The use of *embedded_statement* rather than *statement* excludes the use of declaration statements and labeled statements in these contexts. The example

```
void F(bool b) {
    if (b)
        int i = 44;
}
```

results in a compile-time error because an `if` statement requires an *embedded_statement* rather than a *statement* for its if branch. If this code were permitted, then the variable `i` would be declared, but it could never be used. Note, however, that by placing `i`'s declaration in a block, the example is valid.

End points and reachability

Every statement has an ***end point***. In intuitive terms, the end point of a statement is the location that immediately follows the statement. The execution rules for composite statements (statements that contain embedded statements) specify the action that is taken when control reaches the end point of an embedded statement. For example, when control reaches the end point of a statement in a block, control is transferred to the next statement in the block.

If a statement can possibly be reached by execution, the statement is said to be ***reachable***. Conversely, if there is no possibility that a statement will be executed, the statement is said to be ***unreachable***.

In the example

```
void F() {
    Console.WriteLine("reachable");
    goto Label;
    Console.WriteLine("unreachable");
    Label:
    Console.WriteLine("reachable");
}
```

the second invocation of `Console.WriteLine` is unreachable because there is no possibility that the statement will be executed.

A warning is reported if the compiler determines that a statement is unreachable. It is specifically not an error for a statement to be unreachable.

To determine whether a particular statement or end point is reachable, the compiler performs flow analysis according to the reachability rules defined for each statement. The flow analysis takes into account the values of constant expressions (Constant expressions) that control the behavior of statements, but the possible values of non-constant expressions are not considered. In other words, for purposes of control flow analysis, a non-constant expression of a given type is considered to have any possible value of that type.

In the example

```
void F() {
    const int i = 1;
    if (i == 2) Console.WriteLine("unreachable");
}
```

the boolean expression of the `if` statement is a constant expression because both operands of the `==` operator are constants. As the constant expression is evaluated at compile-time, producing the value `false`, the `Console.WriteLine` invocation is considered unreachable. However, if `i` is changed to be a local variable

```
void F() {
    int i = 1;
    if (i == 2) Console.WriteLine("reachable");
}
```

the `Console.WriteLine` invocation is considered reachable, even though, in reality, it will never be executed.

The *block* of a function member is always considered reachable. By successively evaluating the reachability rules of each statement in a block, the reachability of any given statement can be determined.

In the example

```
void F(int x) {
    Console.WriteLine("start");
    if (x < 0) Console.WriteLine("negative");
}
```

the reachability of the second `Console.WriteLine` is determined as follows:

- The first `Console.WriteLine` expression statement is reachable because the block of the `F` method is reachable.
- The end point of the first `Console.WriteLine` expression statement is reachable because that statement is reachable.
- The `if` statement is reachable because the end point of the first `Console.WriteLine` expression statement is reachable.

- The second `Console.WriteLine` expression statement is reachable because the boolean expression of the `if` statement does not have the constant value `false`.

There are two situations in which it is a compile-time error for the end point of a statement to be reachable:

- Because the `switch` statement does not permit a switch section to "fall through" to the next switch section, it is a compile-time error for the end point of the statement list of a switch section to be reachable. If this error occurs, it is typically an indication that a `break` statement is missing.
- It is a compile-time error for the end point of the block of a function member that computes a value to be reachable. If this error occurs, it typically is an indication that a `return` statement is missing.

Blocks

A *block* permits multiple statements to be written in contexts where a single statement is allowed.

```
block
    : '{' statement_list? '}'
    ;
```

A *block* consists of an optional *statement_list* (Statement lists), enclosed in braces. If the statement list is omitted, the block is said to be empty.

A block may contain declaration statements (Declaration statements). The scope of a local variable or constant declared in a block is the block.

A block is executed as follows:

- If the block is empty, control is transferred to the end point of the block.
- If the block is not empty, control is transferred to the statement list. When and if control reaches the end point of the statement list, control is transferred to the end point of the block.

The statement list of a block is reachable if the block itself is reachable.

The end point of a block is reachable if the block is empty or if the end point of the statement list is reachable.

A *block* that contains one or more `yield` statements (The yield statement) is called an iterator block. Iterator blocks are used to implement function members as iterators (Iterators). Some additional restrictions apply to iterator blocks:

- It is a compile-time error for a `return` statement to appear in an iterator block (but `yield return` statements are permitted).
- It is a compile-time error for an iterator block to contain an unsafe context (Unsafe contexts). An iterator block always defines a safe context, even when its declaration is nested in an unsafe context.

Statement lists

A ***statement list*** consists of one or more statements written in sequence. Statement lists occur in *block*s (Blocks) and in *switch_block*s (The switch statement).

```
statement_list
    : statement+
    ;
```

A statement list is executed by transferring control to the first statement. When and if control reaches the end point of a statement, control is transferred to the next statement. When and if control reaches the end point of the last statement, control is transferred to the end point of the statement list.

A statement in a statement list is reachable if at least one of the following is true:

- The statement is the first statement and the statement list itself is reachable.
- The end point of the preceding statement is reachable.
- The statement is a labeled statement and the label is referenced by a reachable `goto` statement.

The end point of a statement list is reachable if the end point of the last statement in the list is reachable.

The empty statement

An *empty_statement* does nothing.

```
empty_statement
    : ';'
    ;
```

An empty statement is used when there are no operations to perform in a context where a statement is required.

Execution of an empty statement simply transfers control to the end point of the statement. Thus, the end point of an empty statement is reachable if the empty statement is reachable.

An empty statement can be used when writing a `while` statement with a null body:

```
bool ProcessMessage() {...}

void ProcessMessages() {
    while (ProcessMessage())
        ;
}
```

Also, an empty statement can be used to declare a label just before the closing "`}`" of a block:

```
void F() {
    ...
    if (done) goto exit;
    ...
    exit: ;
}
```

Labeled statements

A *labeled_statement* permits a statement to be prefixed by a label. Labeled statements are permitted in blocks, but are not permitted as embedded statements.

```
labeled_statement
    : identifier ':' statement
    ;
```

A labeled statement declares a label with the name given by the *identifier*. The scope of a label is the whole block in which the label is declared, including any nested blocks. It is a compile-time error for two labels with the same name to have overlapping scopes.

A label can be referenced from `goto` statements (The goto statement) within the scope of the label. This means that `goto` statements can transfer control within blocks and out of blocks, but never into blocks.

Labels have their own declaration space and do not interfere with other identifiers. The example

```
int F(int x) {
    if (x >= 0) goto x;
    x = -x;
    x: return x;
}
```

is valid and uses the name `x` as both a parameter and a label.

Execution of a labeled statement corresponds exactly to execution of the statement following the label.

In addition to the reachability provided by normal flow of control, a labeled statement is reachable if the label is referenced by a reachable `goto` statement. (Exception: If a `goto` statement is inside a `try` that includes a `finally` block, and the labeled statement is outside the `try`, and the end point of the `finally` block is unreachable, then the labeled statement is not reachable from that `goto` statement.)

Declaration statements

A *declaration_statement* declares a local variable or constant. Declaration statements are permitted in blocks, but are not permitted as embedded statements.

```
declaration_statement
    : local_variable_declaration ';'
    | local_constant_declaration ';'
    ;
```

Local variable declarations

A *local_variable_declaration* declares one or more local variables.

```
local_variable_declaration
    : local_variable_type local_variable_declarators
    ;

local_variable_type
    : type
    | 'var'
    ;

local_variable_declarators
    : local_variable_declarator
    | local_variable_declarators ',' local_variable_declarator
    ;

local_variable_declarator
    : identifier
    | identifier '=' local_variable_initializer
    ;

local_variable_initializer
    : expression
    | array_initializer
    | local_variable_initializer_unsafe
    ;
```

The *local_variable_type* of a *local_variable_declaration* either directly specifies the type of the variables introduced by the declaration, or indicates with the identifier `var` that the type should be inferred based on an initializer. The type is followed by a list of *local_variable_declarator*s, each of which introduces a new variable. A *local_variable_declarator* consists of an *identifier* that names the variable, optionally followed by an "`=`" token and a *local_variable_initializer* that gives the initial value of the variable.

In the context of a local variable declaration, the identifier var acts as a contextual keyword (Keywords).When the *local_variable_type* is specified as `var` and no type named `var` is in scope, the declaration is an ***implicitly typed local variable declaration***, whose type is inferred from the type of the associated initializer expression. Implicitly typed local variable declarations are subject to the following restrictions:

- The *local_variable_declaration* cannot include multiple *local_variable_declarator*s.
- The *local_variable_declarator* must include a *local_variable_initializer*.
- The *local_variable_initializer* must be an *expression*.
- The initializer *expression* must have a compile-time type.
- The initializer *expression* cannot refer to the declared variable itself

The following are examples of incorrect implicitly typed local variable declarations:

```
var x;               // Error, no initializer to infer type from
var y = {1, 2, 3};   // Error, array initializer not permitted
var z = null;        // Error, null does not have a type
var u = x => x + 1;  // Error, anonymous functions do not have a type
var v = v++;         // Error, initializer cannot refer to variable itself
```

The value of a local variable is obtained in an expression using a *simple_name* (Simple names), and the value of a local variable is modified using an *assignment* (Assignment operators). A local variable must be definitely assigned (Definite assignment) at each location where its value is obtained.

The scope of a local variable declared in a *local_variable_declaration* is the block in which the declaration occurs. It is an error to refer to a local variable in a textual position that precedes the *local_variable_declarator* of the local variable. Within the scope of a local variable, it is a compile-time error to declare another local variable or constant with the same name.

A local variable declaration that declares multiple variables is equivalent to multiple declarations of single variables with the same type. Furthermore, a variable initializer in a local variable declaration corresponds exactly to an assignment statement that is inserted immediately after the declaration.

The example

```
void F() {
    int x = 1, y, z = x * 2;
}
```

corresponds exactly to

```
void F() {
    int x; x = 1;
    int y;
    int z; z = x * 2;
}
```

In an implicitly typed local variable declaration, the type of the local variable being declared is taken to be the same as the type of the expression used to initialize the variable. For example:

```
var i = 5;
var s = "Hello";
var d = 1.0;
var numbers = new int[] {1, 2, 3};
var orders = new Dictionary<int,Order>();
```

The implicitly typed local variable declarations above are precisely equivalent to the following explicitly typed declarations:

```
int i = 5;
string s = "Hello";
double d = 1.0;
int[] numbers = new int[] {1, 2, 3};
Dictionary<int,Order> orders = new Dictionary<int,Order>();
```

Local constant declarations

A *local_constant_declaration* declares one or more local constants.

```
local_constant_declaration
    : 'const' type constant_declarators
    ;

constant_declarators
    : constant_declarator (',' constant_declarator)*
    ;

constant_declarator
    : identifier '=' constant_expression
    ;
```

The *type* of a *local_constant_declaration* specifies the type of the constants introduced by the declaration. The type is followed by a list of *constant_declarator*s, each of which introduces a new constant. A *constant_declarator* consists of an *identifier* that names the constant, followed by an "`=`" token, followed by a *constant_expression* (Constant expressions) that gives the value of the constant.

The *type* and *constant_expression* of a local constant declaration must follow the same rules as those of a constant member declaration (Constants).

The value of a local constant is obtained in an expression using a *simple_name* (Simple names).

The scope of a local constant is the block in which the declaration occurs. It is an error to refer to a local constant in a textual position that precedes its *constant_declarator*. Within the scope of a local constant, it is a compile-time error to declare another local variable or constant with the same name.

A local constant declaration that declares multiple constants is equivalent to multiple declarations of single constants with the same type.

Expression statements

An *expression_statement* evaluates a given expression. The value computed by the expression, if any, is discarded.

```
expression_statement
    : statement_expression ';'
    ;

statement_expression
    : invocation_expression
    | null_conditional_invocation_expression
    | object_creation_expression
    | assignment
    | post_increment_expression
    | post_decrement_expression
    | pre_increment_expression
    | pre_decrement_expression
    | await_expression
    ;
```

Not all expressions are permitted as statements. In particular, expressions such as `x + y` and `x == 1` that merely compute a value (which will be discarded), are not permitted as statements.

Execution of an *expression_statement* evaluates the contained expression and then transfers control to the end point of the *expression_statement*. The end point of an *expression_statement* is reachable if that *expression_statement* is reachable.

Selection statements

Selection statements select one of a number of possible statements for execution based on the value of some expression.

```
selection_statement
    : if_statement
    | switch_statement
    ;
```

The if statement

The `if` statement selects a statement for execution based on the value of a boolean expression.

```
if_statement
    : 'if' '(' boolean_expression ')' embedded_statement
    | 'if' '(' boolean_expression ')' embedded_statement 'else' embedded_statement
    ;
```

An `else` part is associated with the lexically nearest preceding `if` that is allowed by the syntax. Thus, an `if` statement of the form

```
if (x) if (y) F(); else G();
```

is equivalent to

```
if (x) {
    if (y) {
        F();
    }
    else {
        G();
    }
}
```

An `if` statement is executed as follows:

- The *boolean_expression* (Boolean expressions) is evaluated.
- If the boolean expression yields `true`, control is transferred to the first embedded statement. When and if control reaches the end point of that statement, control is transferred to the end point of the `if` statement.
- If the boolean expression yields `false` and if an `else` part is present, control is transferred to the second embedded statement. When and if control reaches the end point of that statement, control is transferred to the end point of the `if` statement.
- If the boolean expression yields `false` and if an `else` part is not present, control is transferred to the end point of the `if` statement.

The first embedded statement of an `if` statement is reachable if the `if` statement is reachable and the boolean expression does not have the constant value `false`.

The second embedded statement of an `if` statement, if present, is reachable if the `if` statement is reachable and the boolean expression does not have the constant value `true`.

The end point of an `if` statement is reachable if the end point of at least one of its embedded statements is reachable. In addition, the end point of an `if` statement with no `else` part is reachable if the `if` statement is reachable and the boolean expression does not have the constant value `true`.

The switch statement

The switch statement selects for execution a statement list having an associated switch label that corresponds to the value of the switch expression.

```
switch_statement
    : 'switch' '(' expression ')' switch_block
    ;

switch_block
    : '{' switch_section* '}'
    ;

switch_section
    : switch_label+ statement_list
    ;

switch_label
    : 'case' constant_expression ':'
    | 'default' ':'
    ;
```

A *switch_statement* consists of the keyword `switch`, followed by a parenthesized expression (called the switch expression), followed by a *switch_block*. The *switch_block* consists of zero or more *switch_section*s, enclosed in braces. Each *switch_section* consists of one or more *switch_label*s followed by a *statement_list* (Statement lists).

The ***governing type*** of a `switch` statement is established by the switch expression.

- If the type of the switch expression is `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `bool`, `char`, `string`, or an *enum_type*, or if it is the nullable type corresponding to one of these types, then that is the governing type of the `switch` statement.
- Otherwise, exactly one user-defined implicit conversion (User-defined conversions) must exist from the type of the switch expression to one of the following possible governing types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `string`, or, a nullable type corresponding to one of those types.
- Otherwise, if no such implicit conversion exists, or if more than one such implicit conversion exists, a compile-time error occurs.

The constant expression of each `case` label must denote a value that is implicitly convertible (Implicit conversions) to the governing type of the `switch` statement. A compile-time error occurs if two or more `case` labels in the same `switch` statement specify the same constant value.

There can be at most one `default` label in a switch statement.

A `switch` statement is executed as follows:

- The switch expression is evaluated and converted to the governing type.
- If one of the constants specified in a `case` label in the same `switch` statement is equal to the value of the switch expression, control is transferred to the statement list following the matched `case` label.
- If none of the constants specified in `case` labels in the same `switch` statement is equal to the value of the switch expression, and if a `default` label is present, control is transferred to the statement list following the `default` label.
- If none of the constants specified in `case` labels in the same `switch` statement is equal to the value of the switch expression, and if no `default` label is present, control is transferred to the end point of the `switch` statement.

If the end point of the statement list of a switch section is reachable, a compile-time error occurs. This is known as the "no fall through" rule. The example

```
switch (i) {
case 0:
    CaseZero();
    break;
case 1:
    CaseOne();
    break;
default:
    CaseOthers();
    break;
}
```

is valid because no switch section has a reachable end point. Unlike C and C++, execution of a switch section is not permitted to "fall through" to the next switch section, and the example

```
switch (i) {
case 0:
    CaseZero();
case 1:
    CaseZeroOrOne();
default:
    CaseAny();
}
```

results in a compile-time error. When execution of a switch section is to be followed by execution of another switch section, an explicit `goto case` or `goto default` statement must be used:

```
switch (i) {
case 0:
    CaseZero();
    goto case 1;
case 1:
    CaseZeroOrOne();
    goto default;
default:
    CaseAny();
    break;
}
```

Multiple labels are permitted in a *switch_section*. The example

```
switch (i) {
case 0:
    CaseZero();
    break;
case 1:
    CaseOne();
    break;
case 2:
default:
    CaseTwo();
    break;
}
```

is valid. The example does not violate the "no fall through" rule because the labels `case 2:` and `default:` are part of the same *switch_section*.

The "no fall through" rule prevents a common class of bugs that occur in C and C++ when `break` statements are accidentally omitted. In addition, because of this rule, the switch sections of a `switch` statement can be arbitrarily rearranged without affecting the behavior of the statement. For example, the sections of the `switch` statement above can be reversed without affecting the behavior of the statement:

```
switch (i) {
default:
    CaseAny();
    break;
case 1:
    CaseZeroOrOne();
    goto default;
case 0:
    CaseZero();
    goto case 1;
}
```

The statement list of a switch section typically ends in a `break`, `goto case`, or `goto default` statement, but any construct that renders the end point of the statement list unreachable is permitted. For example, a `while` statement controlled by the boolean expression `true` is known to never reach its end point. Likewise, a `throw` or `return` statement always transfers control elsewhere and never reaches its end point. Thus, the following example is valid:

```
switch (i) {
case 0:
    while (true) F();
case 1:
    throw new ArgumentException();
case 2:
    return;
}
```

The governing type of a `switch` statement may be the type `string`. For example:

```
void DoCommand(string command) {
    switch (command.ToLower()) {
    case "run":
        DoRun();
        break;
    case "save":
        DoSave();
        break;
    case "quit":
        DoQuit();
        break;
    default:
        InvalidCommand(command);
        break;
    }
}
```

Like the string equality operators (String equality operators), the `switch` statement is case sensitive and will execute a given switch section only if the switch expression string exactly matches a `case` label constant.

When the governing type of a `switch` statement is `string`, the value `null` is permitted as a case label constant.

The *statement_list*s of a *switch_block* may contain declaration statements (Declaration statements). The scope of a local variable or constant declared in a switch block is the switch block.

The statement list of a given switch section is reachable if the `switch` statement is reachable and at least one of the following is true:

- The switch expression is a non-constant value.
- The switch expression is a constant value that matches a `case` label in the switch section.
- The switch expression is a constant value that doesn't match any `case` label, and the switch section contains the `default` label.
- A switch label of the switch section is referenced by a reachable `goto case` or `goto default` statement.

The end point of a `switch` statement is reachable if at least one of the following is true:

- The `switch` statement contains a reachable `break` statement that exits the `switch` statement.
- The `switch` statement is reachable, the switch expression is a non-constant value, and no `default` label is present.
- The `switch` statement is reachable, the switch expression is a constant value that doesn't match any `case` label, and no `default` label is present.

Iteration statements

Iteration statements repeatedly execute an embedded statement.

```
iteration_statement
    : while_statement
    | do_statement
    | for_statement
    | foreach_statement
    ;
```

The while statement

The `while` statement conditionally executes an embedded statement zero or more times.

```
while_statement
    : 'while' '(' boolean_expression ')' embedded_statement
    ;
```

A `while` statement is executed as follows:

- The *boolean_expression* (Boolean expressions) is evaluated.
- If the boolean expression yields `true`, control is transferred to the embedded statement. When and if control reaches the end point of the embedded statement (possibly from execution of a `continue` statement), control is transferred to the beginning of the `while` statement.
- If the boolean expression yields `false`, control is transferred to the end point of the `while` statement.

Within the embedded statement of a `while` statement, a `break` statement (The break statement) may be used to transfer control to the end point of the `while` statement (thus ending iteration of the embedded statement), and a `continue` statement (The continue statement) may be used to transfer control to the end point of the embedded statement (thus performing another iteration of the `while` statement).

The embedded statement of a `while` statement is reachable if the `while` statement is reachable and the boolean expression does not have the constant value `false`.

The end point of a `while` statement is reachable if at least one of the following is true:

- The `while` statement contains a reachable `break` statement that exits the `while` statement.
- The `while` statement is reachable and the boolean expression does not have the constant value `true`.

The do statement

The `do` statement conditionally executes an embedded statement one or more times.

```
do_statement
    : 'do' embedded_statement 'while' '(' boolean_expression ')' ';'
    ;
```

A `do` statement is executed as follows:

- Control is transferred to the embedded statement.
- When and if control reaches the end point of the embedded statement (possibly from execution of a `continue` statement), the *boolean_expression* (Boolean expressions) is evaluated. If the boolean expression yields `true`, control is transferred to the beginning of the `do` statement. Otherwise, control is transferred to the end point of the `do` statement.

Within the embedded statement of a `do` statement, a `break` statement (The break statement) may be used to transfer control to the end point of the `do` statement (thus ending iteration of the embedded statement), and a `continue` statement (The continue statement) may be used to transfer control to the end point of the embedded statement.

The embedded statement of a `do` statement is reachable if the `do` statement is reachable.

The end point of a `do` statement is reachable if at least one of the following is true:

- The `do` statement contains a reachable `break` statement that exits the `do` statement.
- The end point of the embedded statement is reachable and the boolean expression does not have the constant value `true`.

The for statement

The `for` statement evaluates a sequence of initialization expressions and then, while a condition is true, repeatedly executes an embedded statement and evaluates a sequence of iteration expressions.

```
for_statement
    : 'for' '(' for_initializer? ';' for_condition? ';' for_iterator? ')' embedded_statement
    ;

for_initializer
    : local_variable_declaration
    | statement_expression_list
    ;

for_condition
    : boolean_expression
    ;

for_iterator
    : statement_expression_list
    ;

statement_expression_list
    : statement_expression (',' statement_expression)*
    ;
```

The *for_initializer*, if present, consists of either a *local_variable_declaration* (Local variable declarations) or a list of *statement_expression*s (Expression statements) separated by commas. The scope of a local variable declared by a *for_initializer* starts at the *local_variable_declarator* for the variable and extends to the end of the embedded statement. The scope includes the *for_condition* and the *for_iterator*.

The *for_condition*, if present, must be a *boolean_expression* (Boolean expressions).

The *for_iterator*, if present, consists of a list of *statement_expression*s (Expression statements) separated by commas.

A for statement is executed as follows:

- If a *for_initializer* is present, the variable initializers or statement expressions are executed in the order they are written. This step is only performed once.
- If a *for_condition* is present, it is evaluated.
- If the *for_condition* is not present or if the evaluation yields `true`, control is transferred to the embedded statement. When and if control reaches the end point of the embedded statement (possibly from execution of a `continue` statement), the expressions of the *for_iterator*, if any, are evaluated in sequence, and then another iteration is performed, starting with evaluation of the *for_condition* in the step above.
- If the *for_condition* is present and the evaluation yields `false`, control is transferred to the end point of the `for` statement.

Within the embedded statement of a `for` statement, a `break` statement (The break statement) may be used to transfer control to the end point of the `for` statement (thus ending iteration of the embedded statement), and a `continue` statement (The continue statement) may be used to transfer control to the end point of the embedded

statement (thus executing the *for_iterator* and performing another iteration of the `for` statement, starting with the *for_condition*).

The embedded statement of a `for` statement is reachable if one of the following is true:

- The `for` statement is reachable and no *for_condition* is present.
- The `for` statement is reachable and a *for_condition* is present and does not have the constant value `false`.

The end point of a `for` statement is reachable if at least one of the following is true:

- The `for` statement contains a reachable `break` statement that exits the `for` statement.
- The `for` statement is reachable and a *for_condition* is present and does not have the constant value `true`.

The foreach statement

The `foreach` statement enumerates the elements of a collection, executing an embedded statement for each element of the collection.

```
foreach_statement
    : 'foreach' '(' local_variable_type identifier 'in' expression ')' embedded_statement
    ;
```

The *type* and *identifier* of a `foreach` statement declare the ***iteration variable*** of the statement. If the `var` identifier is given as the *local_variable_type*, and no type named `var` is in scope, the iteration variable is said to be an ***implicitly typed iteration variable***, and its type is taken to be the element type of the `foreach` statement, as specified below. The iteration variable corresponds to a read-only local variable with a scope that extends over the embedded statement. During execution of a `foreach` statement, the iteration variable represents the collection element for which an iteration is currently being performed. A compile-time error occurs if the embedded statement attempts to modify the iteration variable (via assignment or the `++` and `--` operators) or pass the iteration variable as a `ref` or `out` parameter.

In the following, for brevity, `IEnumerable`, `IEnumerator`, `IEnumerable<T>` and `IEnumerator<T>` refer to the corresponding types in the namespaces `System.Collections` and `System.Collections.Generic`.

The compile-time processing of a foreach statement first determines the ***collection type***, ***enumerator type*** and ***element type*** of the expression. This determination proceeds as follows:

- If the type `X` of *expression* is an array type then there is an implicit reference conversion from `X` to the `IEnumerable` interface (since `System.Array` implements this interface). The ***collection type*** is the `IEnumerable` interface, the ***enumerator type*** is the `IEnumerator` interface and the ***element type*** is the element type of the array type `X`.
- If the type `X` of *expression* is `dynamic` then there is an implicit conversion from *expression* to the `IEnumerable` interface (Implicit dynamic conversions). The ***collection type*** is the `IEnumerable` interface and the ***enumerator type*** is the `IEnumerator` interface. If the `var` identifier is given as the *local_variable_type* then the ***element type*** is `dynamic`, otherwise it is `object`.
- Otherwise, determine whether the type `X` has an appropriate `GetEnumerator` method:
  - Perform member lookup on the type `X` with identifier `GetEnumerator` and no type arguments. If the member lookup does not produce a match, or it produces an ambiguity, or produces a match that is not a method group, check for an enumerable interface as described below. It is recommended that a warning be issued if member lookup produces anything except a method group or no match.
  - Perform overload resolution using the resulting method group and an empty argument list. If overload resolution results in no applicable methods, results in an ambiguity, or results in a single best method but that method is either static or not public, check for an enumerable interface as described

below. It is recommended that a warning be issued if overload resolution produces anything except an unambiguous public instance method or no applicable methods.
    - If the return type `E` of the `GetEnumerator` method is not a class, struct or interface type, an error is produced and no further steps are taken.
    - Member lookup is performed on `E` with the identifier `Current` and no type arguments. If the member lookup produces no match, the result is an error, or the result is anything except a public instance property that permits reading, an error is produced and no further steps are taken.
    - Member lookup is performed on `E` with the identifier `MoveNext` and no type arguments. If the member lookup produces no match, the result is an error, or the result is anything except a method group, an error is produced and no further steps are taken.
    - Overload resolution is performed on the method group with an empty argument list. If overload resolution results in no applicable methods, results in an ambiguity, or results in a single best method but that method is either static or not public, or its return type is not `bool`, an error is produced and no further steps are taken.
    - The ***collection type*** is `X`, the ***enumerator type*** is `E`, and the ***element type*** is the type of the `Current` property.
- Otherwise, check for an enumerable interface:
    - If among all the types `Ti` for which there is an implicit conversion from `X` to `IEnumerable<Ti>`, there is a unique type `T` such that `T` is not `dynamic` and for all the other `Ti` there is an implicit conversion from `IEnumerable<T>` to `IEnumerable<Ti>`, then the ***collection type*** is the interface `IEnumerable<T>`, the ***enumerator type*** is the interface `IEnumerator<T>`, and the ***element type*** is `T`.
    - Otherwise, if there is more than one such type `T`, then an error is produced and no further steps are taken.
    - Otherwise, if there is an implicit conversion from `X` to the `System.Collections.IEnumerable` interface, then the ***collection type*** is this interface, the ***enumerator type*** is the interface `System.Collections.IEnumerator`, and the ***element type*** is `object`.
    - Otherwise, an error is produced and no further steps are taken.

The above steps, if successful, unambiguously produce a collection type `C`, enumerator type `E` and element type `T`. A foreach statement of the form

```
foreach (V v in x) embedded_statement
```

is then expanded to:

```
{
    E e = ((C)(x)).GetEnumerator();
    try {
        while (e.MoveNext()) {
            V v = (V)(T)e.Current;
            embedded_statement
        }
    }
    finally {
        ... // Dispose e
    }
}
```

The variable `e` is not visible to or accessible to the expression `x` or the embedded statement or any other source code of the program. The variable `v` is read-only in the embedded statement. If there is not an explicit conversion (Explicit conversions) from `T` (the element type) to `V` (the *local_variable_type* in the foreach statement), an error is produced and no further steps are taken. If `x` has the value `null`, a

`System.NullReferenceException` is thrown at run-time.

An implementation is permitted to implement a given foreach-statement differently, e.g. for performance reasons, as long as the behavior is consistent with the above expansion.

The placement of `v` inside the while loop is important for how it is captured by any anonymous function occurring in the *embedded_statement*.

For example:

```
int[] values = { 7, 9, 13 };
Action f = null;

foreach (var value in values)
{
    if (f == null) f = () => Console.WriteLine("First value: " + value);
}

f();
```

If `v` was declared outside of the while loop, it would be shared among all iterations, and its value after the for loop would be the final value, `13`, which is what the invocation of `f` would print. Instead, because each iteration has its own variable `v`, the one captured by `f` in the first iteration will continue to hold the value `7`, which is what will be printed. (Note: earlier versions of C# declared `v` outside of the while loop.)

The body of the finally block is constructed according to the following steps:

- If there is an implicit conversion from `E` to the `System.IDisposable` interface, then
  - If `E` is a non-nullable value type then the finally clause is expanded to the semantic equivalent of:

    ```
    finally {
        ((System.IDisposable)e).Dispose();
    }
    ```

  - Otherwise the finally clause is expanded to the semantic equivalent of:

    ```
    finally {
        if (e != null) ((System.IDisposable)e).Dispose();
    }
    ```

    except that if `E` is a value type, or a type parameter instantiated to a value type, then the cast of `e` to `System.IDisposable` will not cause boxing to occur.
- Otherwise, if `E` is a sealed type, the finally clause is expanded to an empty block:

  ```
  finally {
  }
  ```

- Otherwise, the finally clause is expanded to:

  ```
  finally {
      System.IDisposable d = e as System.IDisposable;
      if (d != null) d.Dispose();
  }
  ```

The local variable `d` is not visible to or accessible to any user code. In particular, it does not conflict with any other variable whose scope includes the finally block.

The order in which `foreach` traverses the elements of an array, is as follows: For single-dimensional arrays elements are traversed in increasing index order, starting with index `0` and ending with index `Length - 1`. For multi-dimensional arrays, elements are traversed such that the indices of the rightmost dimension are increased first, then the next left dimension, and so on to the left.

The following example prints out each value in a two-dimensional array, in element order:

```
using System;

class Test
{
    static void Main() {
        double[,] values = {
            {1.2, 2.3, 3.4, 4.5},
            {5.6, 6.7, 7.8, 8.9}
        };

        foreach (double elementValue in values)
            Console.Write("{0} ", elementValue);

        Console.WriteLine();
    }
}
```

The output produced is as follows:

```
1.2 2.3 3.4 4.5 5.6 6.7 7.8 8.9
```

In the example

```
int[] numbers = { 1, 3, 5, 7, 9 };
foreach (var n in numbers) Console.WriteLine(n);
```

the type of `n` is inferred to be `int`, the element type of `numbers`.

Jump statements

Jump statements unconditionally transfer control.

```
jump_statement
    : break_statement
    | continue_statement
    | goto_statement
    | return_statement
    | throw_statement
    ;
```

The location to which a jump statement transfers control is called the ***target*** of the jump statement.

When a jump statement occurs within a block, and the target of that jump statement is outside that block, the jump statement is said to ***exit*** the block. While a jump statement may transfer control out of a block, it can never transfer control into a block.

Execution of jump statements is complicated by the presence of intervening `try` statements. In the absence of

such `try` statements, a jump statement unconditionally transfers control from the jump statement to its target. In the presence of such intervening `try` statements, execution is more complex. If the jump statement exits one or more `try` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all intervening `try` statements have been executed.

In the example

```
using System;

class Test
{
    static void Main() {
        while (true) {
            try {
                try {
                    Console.WriteLine("Before break");
                    break;
                }
                finally {
                    Console.WriteLine("Innermost finally block");
                }
            }
            finally {
                Console.WriteLine("Outermost finally block");
            }
        }
        Console.WriteLine("After break");
    }
}
```

the `finally` blocks associated with two `try` statements are executed before control is transferred to the target of the jump statement.

The output produced is as follows:

```
Before break
Innermost finally block
Outermost finally block
After break
```

The break statement

The `break` statement exits the nearest enclosing `switch`, `while`, `do`, `for`, or `foreach` statement.

```
break_statement
    : 'break' ';'
    ;
```

The target of a `break` statement is the end point of the nearest enclosing `switch`, `while`, `do`, `for`, or `foreach` statement. If a `break` statement is not enclosed by a `switch`, `while`, `do`, `for`, or `foreach` statement, a compile-time error occurs.

When multiple `switch`, `while`, `do`, `for`, or `foreach` statements are nested within each other, a `break` statement applies only to the innermost statement. To transfer control across multiple nesting levels, a `goto` statement (The goto statement) must be used.

A `break` statement cannot exit a `finally` block (The try statement). When a `break` statement occurs within a

`finally` block, the target of the `break` statement must be within the same `finally` block; otherwise, a compile-time error occurs.

A `break` statement is executed as follows:

- If the `break` statement exits one or more `try` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all intervening `try` statements have been executed.
- Control is transferred to the target of the `break` statement.

Because a `break` statement unconditionally transfers control elsewhere, the end point of a `break` statement is never reachable.

The continue statement

The `continue` statement starts a new iteration of the nearest enclosing `while`, `do`, `for`, or `foreach` statement.

```
continue_statement
    : 'continue' ';'
    ;
```

The target of a `continue` statement is the end point of the embedded statement of the nearest enclosing `while`, `do`, `for`, or `foreach` statement. If a `continue` statement is not enclosed by a `while`, `do`, `for`, or `foreach` statement, a compile-time error occurs.

When multiple `while`, `do`, `for`, or `foreach` statements are nested within each other, a `continue` statement applies only to the innermost statement. To transfer control across multiple nesting levels, a `goto` statement (The goto statement) must be used.

A `continue` statement cannot exit a `finally` block (The try statement). When a `continue` statement occurs within a `finally` block, the target of the `continue` statement must be within the same `finally` block; otherwise a compile-time error occurs.

A `continue` statement is executed as follows:

- If the `continue` statement exits one or more `try` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all intervening `try` statements have been executed.
- Control is transferred to the target of the `continue` statement.

Because a `continue` statement unconditionally transfers control elsewhere, the end point of a `continue` statement is never reachable.

The goto statement

The `goto` statement transfers control to a statement that is marked by a label.

```
goto_statement
    : 'goto' identifier ';'
    | 'goto' 'case' constant_expression ';'
    | 'goto' 'default' ';'
    ;
```

The target of a `goto` *identifier* statement is the labeled statement with the given label. If a label with the given

name does not exist in the current function member, or if the `goto` statement is not within the scope of the label, a compile-time error occurs. This rule permits the use of a `goto` statement to transfer control out of a nested scope, but not into a nested scope. In the example

```
using System;

class Test
{
    static void Main(string[] args) {
        string[,] table = {
            {"Red", "Blue", "Green"},
            {"Monday", "Wednesday", "Friday"}
        };

        foreach (string str in args) {
            int row, colm;
            for (row = 0; row <= 1; ++row)
                for (colm = 0; colm <= 2; ++colm)
                    if (str == table[row,colm])
                        goto done;

            Console.WriteLine("{0} not found", str);
            continue;
    done:
            Console.WriteLine("Found {0} at [{1}][{2}]", str, row, colm);
        }
    }
}
```

a `goto` statement is used to transfer control out of a nested scope.

The target of a `goto case` statement is the statement list in the immediately enclosing `switch` statement (The switch statement), which contains a `case` label with the given constant value. If the `goto case` statement is not enclosed by a `switch` statement, if the *constant_expression* is not implicitly convertible (Implicit conversions) to the governing type of the nearest enclosing `switch` statement, or if the nearest enclosing `switch` statement does not contain a `case` label with the given constant value, a compile-time error occurs.

The target of a `goto default` statement is the statement list in the immediately enclosing `switch` statement (The switch statement), which contains a `default` label. If the `goto default` statement is not enclosed by a `switch` statement, or if the nearest enclosing `switch` statement does not contain a `default` label, a compile-time error occurs.

A `goto` statement cannot exit a `finally` block (The try statement). When a `goto` statement occurs within a `finally` block, the target of the `goto` statement must be within the same `finally` block, or otherwise a compile-time error occurs.

A `goto` statement is executed as follows:

- If the `goto` statement exits one or more `try` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all intervening `try` statements have been executed.
- Control is transferred to the target of the `goto` statement.

Because a `goto` statement unconditionally transfers control elsewhere, the end point of a `goto` statement is never reachable.

The return statement

The `return` statement returns control to the current caller of the function in which the `return` statement

appears.

```
return_statement
    : 'return' expression? ';'
    ;
```

A `return` statement with no expression can be used only in a function member that does not compute a value, that is, a method with the result type (Method body) `void`, the `set` accessor of a property or indexer, the `add` and `remove` accessors of an event, an instance constructor, a static constructor, or a destructor.

A `return` statement with an expression can only be used in a function member that computes a value, that is, a method with a non-void result type, the `get` accessor of a property or indexer, or a user-defined operator. An implicit conversion (Implicit conversions) must exist from the type of the expression to the return type of the containing function member.

Return statements can also be used in the body of anonymous function expressions (Anonymous function expressions), and participate in determining which conversions exist for those functions.

It is a compile-time error for a `return` statement to appear in a `finally` block (The try statement).

A `return` statement is executed as follows:

- If the `return` statement specifies an expression, the expression is evaluated and the resulting value is converted to the return type of the containing function by an implicit conversion. The result of the conversion becomes the result value produced by the function.
- If the `return` statement is enclosed by one or more `try` or `catch` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all enclosing `try` statements have been executed.
- If the containing function is not an async function, control is returned to the caller of the containing function along with the result value, if any.
- If the containing function is an async function, control is returned to the current caller, and the result value, if any, is recorded in the return task as described in (Enumerator interfaces).

Because a `return` statement unconditionally transfers control elsewhere, the end point of a `return` statement is never reachable.

The throw statement

The `throw` statement throws an exception.

```
throw_statement
    : 'throw' expression? ';'
    ;
```

A `throw` statement with an expression throws the value produced by evaluating the expression. The expression must denote a value of the class type `System.Exception`, of a class type that derives from `System.Exception` or of a type parameter type that has `System.Exception` (or a subclass thereof) as its effective base class. If evaluation of the expression produces `null`, a `System.NullReferenceException` is thrown instead.

A `throw` statement with no expression can be used only in a `catch` block, in which case that statement re-throws the exception that is currently being handled by that `catch` block.

Because a `throw` statement unconditionally transfers control elsewhere, the end point of a `throw` statement is never reachable.

When an exception is thrown, control is transferred to the first `catch` clause in an enclosing `try` statement that can handle the exception. The process that takes place from the point of the exception being thrown to the point of transferring control to a suitable exception handler is known as ***exception propagation***. Propagation of an exception consists of repeatedly evaluating the following steps until a `catch` clause that matches the exception is found. In this description, the ***throw point*** is initially the location at which the exception is thrown.

- In the current function member, each `try` statement that encloses the throw point is examined. For each statement `S`, starting with the innermost `try` statement and ending with the outermost `try` statement, the following steps are evaluated:
  - If the `try` block of `S` encloses the throw point and if S has one or more `catch` clauses, the `catch` clauses are examined in order of appearance to locate a suitable handler for the exception, according to the rules specified in Section The try statement. If a matching `catch` clause is located, the exception propagation is completed by transferring control to the block of that `catch` clause.
  - Otherwise, if the `try` block or a `catch` block of `S` encloses the throw point and if `S` has a `finally` block, control is transferred to the `finally` block. If the `finally` block throws another exception, processing of the current exception is terminated. Otherwise, when control reaches the end point of the `finally` block, processing of the current exception is continued.
- If an exception handler was not located in the current function invocation, the function invocation is terminated, and one of the following occurs:
  - If the current function is non-async, the steps above are repeated for the caller of the function with a throw point corresponding to the statement from which the function member was invoked.
  - If the current function is async and task-returning, the exception is recorded in the return task, which is put into a faulted or cancelled state as described in Enumerator interfaces.
  - If the current function is async and void-returning, the synchronization context of the current thread is notified as described in Enumerable interfaces.
- If the exception processing terminates all function member invocations in the current thread, indicating that the thread has no handler for the exception, then the thread is itself terminated. The impact of such termination is implementation-defined.

The try statement

The `try` statement provides a mechanism for catching exceptions that occur during execution of a block. Furthermore, the `try` statement provides the ability to specify a block of code that is always executed when control leaves the `try` statement.

```
try_statement
    : 'try' block catch_clause+
    | 'try' block finally_clause
    | 'try' block catch_clause+ finally_clause
    ;

catch_clause
    : 'catch' exception_specifier? exception_filter?  block
    ;

exception_specifier
    : '(' type identifier? ')'
    ;

exception_filter
    : 'when' '(' expression ')'
    ;

finally_clause
    : 'finally' block
    ;
```

There are three possible forms of `try` statements:

- A `try` block followed by one or more `catch` blocks.
- A `try` block followed by a `finally` block.
- A `try` block followed by one or more `catch` blocks followed by a `finally` block.

When a `catch` clause specifies an *exception_specifier*, the type must be `System.Exception`, a type that derives from `System.Exception` or a type parameter type that has `System.Exception` (or a subclass thereof) as its effective base class.

When a `catch` clause specifies both an *exception_specifier* with an *identifier*, an ***exception variable*** of the given name and type is declared. The exception variable corresponds to a local variable with a scope that extends over the `catch` clause. During execution of the *exception_filter* and *block*, the exception variable represents the exception currently being handled. For purposes of definite assignment checking, the exception variable is considered definitely assigned in its entire scope.

Unless a `catch` clause includes an exception variable name, it is impossible to access the exception object in the filter and `catch` block.

A `catch` clause that does not specify an *exception_specifier* is called a general `catch` clause.

Some programming languages may support exceptions that are not representable as an object derived from `System.Exception`, although such exceptions could never be generated by C# code. A general `catch` clause may be used to catch such exceptions. Thus, a general `catch` clause is semantically different from one that specifies the type `System.Exception`, in that the former may also catch exceptions from other languages.

In order to locate a handler for an exception, `catch` clauses are examined in lexical order. If a `catch` clause specifies a type but no exception filter, it is a compile-time error for a later `catch` clause in the same `try` statement to specify a type that is the same as, or is derived from, that type. If a `catch` clause specifies no type and no filter, it must be the last `catch` clause for that `try` statement.

Within a `catch` block, a `throw` statement (The throw statement) with no expression can be used to re-throw the exception that was caught by the `catch` block. Assignments to an exception variable do not alter the exception that is re-thrown.

In the example

```
using System;

class Test
{
    static void F() {
        try {
            G();
        }
        catch (Exception e) {
            Console.WriteLine("Exception in F: " + e.Message);
            e = new Exception("F");
            throw;                // re-throw
        }
    }

    static void G() {
        throw new Exception("G");
    }

    static void Main() {
        try {
            F();
        }
        catch (Exception e) {
            Console.WriteLine("Exception in Main: " + e.Message);
        }
    }
}
```

the method `F` catches an exception, writes some diagnostic information to the console, alters the exception variable, and re-throws the exception. The exception that is re-thrown is the original exception, so the output produced is:

```
Exception in F: G
Exception in Main: G
```

If the first catch block had thrown `e` instead of rethrowing the current exception, the output produced would be as follows:

```
Exception in F: G
Exception in Main: F
```

It is a compile-time error for a `break`, `continue`, or `goto` statement to transfer control out of a `finally` block. When a `break`, `continue`, or `goto` statement occurs in a `finally` block, the target of the statement must be within the same `finally` block, or otherwise a compile-time error occurs.

It is a compile-time error for a `return` statement to occur in a `finally` block.

A `try` statement is executed as follows:

- Control is transferred to the `try` block.
- When and if control reaches the end point of the `try` block:
  - If the `try` statement has a `finally` block, the `finally` block is executed.
  - Control is transferred to the end point of the `try` statement.
- If an exception is propagated to the `try` statement during execution of the `try` block:
  - The `catch` clauses, if any, are examined in order of appearance to locate a suitable handler for the

exception. If a `catch` clause does not specify a type, or specifies the exception type or a base type of the exception type:

- If the `catch` clause declares an exception variable, the exception object is assigned to the exception variable.
- If the `catch` clause declares an exception filter, the filter is evaluated. If it evaluates to `false`, the catch clause is not a match, and the search continues through any subsequent `catch` clauses for a suitable handler.
- Otherwise, the `catch` clause is considered a match, and control is transferred to the matching `catch` block.
- When and if control reaches the end point of the `catch` block:
  - If the `try` statement has a `finally` block, the `finally` block is executed.
  - Control is transferred to the end point of the `try` statement.
- If an exception is propagated to the `try` statement during execution of the `catch` block:
  - If the `try` statement has a `finally` block, the `finally` block is executed.
  - The exception is propagated to the next enclosing `try` statement.

- If the `try` statement has no `catch` clauses or if no `catch` clause matches the exception:
  - If the `try` statement has a `finally` block, the `finally` block is executed.
  - The exception is propagated to the next enclosing `try` statement.

The statements of a `finally` block are always executed when control leaves a `try` statement. This is true whether the control transfer occurs as a result of normal execution, as a result of executing a `break`, `continue`, `goto`, or `return` statement, or as a result of propagating an exception out of the `try` statement.

If an exception is thrown during execution of a `finally` block, and is not caught within the same finally block, the exception is propagated to the next enclosing `try` statement. If another exception was in the process of being propagated, that exception is lost. The process of propagating an exception is discussed further in the description of the `throw` statement (The throw statement).

The `try` block of a `try` statement is reachable if the `try` statement is reachable.

A `catch` block of a `try` statement is reachable if the `try` statement is reachable.

The `finally` block of a `try` statement is reachable if the `try` statement is reachable.

The end point of a `try` statement is reachable if both of the following are true:

- The end point of the `try` block is reachable or the end point of at least one `catch` block is reachable.
- If a `finally` block is present, the end point of the `finally` block is reachable.

The checked and unchecked statements

The `checked` and `unchecked` statements are used to control the ***overflow checking context*** for integral-type arithmetic operations and conversions.

```
checked_statement
    : 'checked' block
    ;

unchecked_statement
    : 'unchecked' block
    ;
```

The `checked` statement causes all expressions in the *block* to be evaluated in a checked context, and the

`unchecked` statement causes all expressions in the *block* to be evaluated in an unchecked context.

The `checked` and `unchecked` statements are precisely equivalent to the `checked` and `unchecked` operators (The checked and unchecked operators), except that they operate on blocks instead of expressions.

The lock statement

The `lock` statement obtains the mutual-exclusion lock for a given object, executes a statement, and then releases the lock.

```
lock_statement
    : 'lock' '(' expression ')' embedded_statement
    ;
```

The expression of a `lock` statement must denote a value of a type known to be a *reference_type*. No implicit boxing conversion (Boxing conversions) is ever performed for the expression of a `lock` statement, and thus it is a compile-time error for the expression to denote a value of a *value_type*.

A `lock` statement of the form

```
lock (x) ...
```

where `x` is an expression of a *reference_type*, is precisely equivalent to

```
bool __lockWasTaken = false;
try {
    System.Threading.Monitor.Enter(x, ref __lockWasTaken);
    ...
}
finally {
    if (__lockWasTaken) System.Threading.Monitor.Exit(x);
}
```

except that `x` is only evaluated once.

While a mutual-exclusion lock is held, code executing in the same execution thread can also obtain and release the lock. However, code executing in other threads is blocked from obtaining the lock until the lock is released.

Locking `System.Type` objects in order to synchronize access to static data is not recommended. Other code might lock on the same type, which can result in deadlock. A better approach is to synchronize access to static data by locking a private static object. For example:

```
class Cache
{
    private static readonly object synchronizationObject = new object();

    public static void Add(object x) {
        lock (Cache.synchronizationObject) {
            ...
        }
    }

    public static void Remove(object x) {
        lock (Cache.synchronizationObject) {
            ...
        }
    }
}
```

The using statement

The `using` statement obtains one or more resources, executes a statement, and then disposes of the resource.

```
using_statement
    : 'using' '(' resource_acquisition ')' embedded_statement
    ;

resource_acquisition
    : local_variable_declaration
    | expression
    ;
```

A ***resource*** is a class or struct that implements `System.IDisposable`, which includes a single parameterless method named `Dispose`. Code that is using a resource can call `Dispose` to indicate that the resource is no longer needed. If `Dispose` is not called, then automatic disposal eventually occurs as a consequence of garbage collection.

If the form of *resource_acquisition* is *local_variable_declaration* then the type of the *local_variable_declaration* must be either `dynamic` or a type that can be implicitly converted to `System.IDisposable`. If the form of *resource_acquisition* is *expression* then this expression must be implicitly convertible to `System.IDisposable`.

Local variables declared in a *resource_acquisition* are read-only, and must include an initializer. A compile-time error occurs if the embedded statement attempts to modify these local variables (via assignment or the `++` and `--` operators) , take the address of them, or pass them as `ref` or `out` parameters.

A `using` statement is translated into three parts: acquisition, usage, and disposal. Usage of the resource is implicitly enclosed in a `try` statement that includes a `finally` clause. This `finally` clause disposes of the resource. If a `null` resource is acquired, then no call to `Dispose` is made, and no exception is thrown. If the resource is of type `dynamic` it is dynamically converted through an implicit dynamic conversion (Implicit dynamic conversions) to `IDisposable` during acquisition in order to ensure that the conversion is successful before the usage and disposal.

A `using` statement of the form

```
using (ResourceType resource = expression) statement
```

corresponds to one of three possible expansions. When `ResourceType` is a non-nullable value type, the expansion is

```
{
    ResourceType resource = expression;
    try {
        statement;
    }
    finally {
        ((IDisposable)resource).Dispose();
    }
}
```

Otherwise, when `ResourceType` is a nullable value type or a reference type other than `dynamic`, the expansion is

```
{
    ResourceType resource = expression;
    try {
        statement;
    }
    finally {
        if (resource != null) ((IDisposable)resource).Dispose();
    }
}
```

Otherwise, when `ResourceType` is `dynamic`, the expansion is

```
{
    ResourceType resource = expression;
    IDisposable d = (IDisposable)resource;
    try {
        statement;
    }
    finally {
        if (d != null) d.Dispose();
    }
}
```

In either expansion, the `resource` variable is read-only in the embedded statement, and the `d` variable is inaccessible in, and invisible to, the embedded statement.

An implementation is permitted to implement a given using-statement differently, e.g. for performance reasons, as long as the behavior is consistent with the above expansion.

A `using` statement of the form

```
using (expression) statement
```

has the same three possible expansions. In this case `ResourceType` is implicitly the compile-time type of the `expression`, if it has one. Otherwise the interface `IDisposable` itself is used as the `ResourceType`. The `resource` variable is inaccessible in, and invisible to, the embedded statement.

When a *resource_acquisition* takes the form of a *local_variable_declaration*, it is possible to acquire multiple resources of a given type. A `using` statement of the form

```
using (ResourceType r1 = e1, r2 = e2, ..., rN = eN) statement
```

is precisely equivalent to a sequence of nested `using` statements:

```
using (ResourceType r1 = e1)
    using (ResourceType r2 = e2)
        ...
            using (ResourceType rN = eN)
                statement
```

The example below creates a file named `log.txt` and writes two lines of text to the file. The example then opens that same file for reading and copies the contained lines of text to the console.

```
using System;
using System.IO;

class Test
{
    static void Main() {
        using (TextWriter w = File.CreateText("log.txt")) {
            w.WriteLine("This is line one");
            w.WriteLine("This is line two");
        }

        using (TextReader r = File.OpenText("log.txt")) {
            string s;
            while ((s = r.ReadLine()) != null) {
                Console.WriteLine(s);
            }

        }
    }
}
```

Since the `TextWriter` and `TextReader` classes implement the `IDisposable` interface, the example can use `using` statements to ensure that the underlying file is properly closed following the write or read operations.

The yield statement

The `yield` statement is used in an iterator block (Blocks) to yield a value to the enumerator object (Enumerator objects) or enumerable object (Enumerable objects) of an iterator or to signal the end of the iteration.

```
yield_statement
    : 'yield' 'return' expression ';'
    | 'yield' 'break' ';'
    ;
```

`yield` is not a reserved word; it has special meaning only when used immediately before a `return` or `break` keyword. In other contexts, `yield` can be used as an identifier.

There are several restrictions on where a `yield` statement can appear, as described in the following.

- It is a compile-time error for a `yield` statement (of either form) to appear outside a *method_body*, *operator_body* or *accessor_body*
- It is a compile-time error for a `yield` statement (of either form) to appear inside an anonymous function.
- It is a compile-time error for a `yield` statement (of either form) to appear in the `finally` clause of a `try` statement.
- It is a compile-time error for a `yield return` statement to appear anywhere in a `try` statement that contains any `catch` clauses.

The following example shows some valid and invalid uses of `yield` statements.

```
delegate IEnumerable<int> D();

IEnumerator<int> GetEnumerator() {
    try {
        yield return 1;        // Ok
        yield break;           // Ok
    }
    finally {
        yield return 2;        // Error, yield in finally
        yield break;           // Error, yield in finally
    }

    try {
        yield return 3;        // Error, yield return in try...catch
        yield break;           // Ok
    }
    catch {
        yield return 4;        // Error, yield return in try...catch
        yield break;           // Ok
    }

    D d = delegate {
        yield return 5;        // Error, yield in an anonymous function
    };
}

int MyMethod() {
    yield return 1;            // Error, wrong return type for an iterator block
}
```

An implicit conversion (Implicit conversions) must exist from the type of the expression in the `yield return` statement to the yield type (Yield type) of the iterator.

A `yield return` statement is executed as follows:

- The expression given in the statement is evaluated, implicitly converted to the yield type, and assigned to the `Current` property of the enumerator object.
- Execution of the iterator block is suspended. If the `yield return` statement is within one or more `try` blocks, the associated `finally` blocks are not executed at this time.
- The `MoveNext` method of the enumerator object returns `true` to its caller, indicating that the enumerator object successfully advanced to the next item.

The next call to the enumerator object's `MoveNext` method resumes execution of the iterator block from where it was last suspended.

A `yield break` statement is executed as follows:

- If the `yield break` statement is enclosed by one or more `try` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all enclosing `try` statements have been executed.
- Control is returned to the caller of the iterator block. This is either the `MoveNext` method or `Dispose` method of the enumerator object.

Because a `yield break` statement unconditionally transfers control elsewhere, the end point of a `yield break` statement is never reachable.

Namespaces

18.12.2021 • 18 minutes to read

C# programs are organized using namespaces. Namespaces are used both as an "internal" organization system for a program, and as an "external" organization system—a way of presenting program elements that are exposed to other programs.

Using directives (Using directives) are provided to facilitate the use of namespaces.

Compilation units

A *compilation_unit* defines the overall structure of a source file. A compilation unit consists of zero or more *using_directive*s followed by zero or more *global_attributes* followed by zero or more *namespace_member_declaration*s.

```
compilation_unit
    : extern_alias_directive* using_directive* global_attributes? namespace_member_declaration*
    ;
```

A C# program consists of one or more compilation units, each contained in a separate source file. When a C# program is compiled, all of the compilation units are processed together. Thus, compilation units can depend on each other, possibly in a circular fashion.

The *using_directive*s of a compilation unit affect the *global_attributes* and *namespace_member_declaration*s of that compilation unit, but have no effect on other compilation units.

The *global_attributes* (Attributes) of a compilation unit permit the specification of attributes for the target assembly and module. Assemblies and modules act as physical containers for types. An assembly may consist of several physically separate modules.

The *namespace_member_declaration*s of each compilation unit of a program contribute members to a single declaration space called the global namespace. For example:

File `A.cs`:

```
class A {}
```

File `B.cs`:

```
class B {}
```

The two compilation units contribute to the single global namespace, in this case declaring two classes with the fully qualified names `A` and `B`. Because the two compilation units contribute to the same declaration space, it would have been an error if each contained a declaration of a member with the same name.

Namespace declarations

A *namespace_declaration* consists of the keyword `namespace`, followed by a namespace name and body, optionally followed by a semicolon.

```
namespace_declaration
    : 'namespace' qualified_identifier namespace_body ';'?
    ;

qualified_identifier
    : identifier ('.' identifier)*
    ;

namespace_body
    : '{' extern_alias_directive* using_directive* namespace_member_declaration* '}'
    ;
```

A *namespace_declaration* may occur as a top-level declaration in a *compilation_unit* or as a member declaration within another *namespace_declaration*. When a *namespace_declaration* occurs as a top-level declaration in a *compilation_unit*, the namespace becomes a member of the global namespace. When a *namespace_declaration* occurs within another *namespace_declaration*, the inner namespace becomes a member of the outer namespace. In either case, the name of a namespace must be unique within the containing namespace.

Namespaces are implicitly `public` and the declaration of a namespace cannot include any access modifiers.

Within a *namespace_body*, the optional *using_directive*s import the names of other namespaces, types and members, allowing them to be referenced directly instead of through qualified names. The optional *namespace_member_declaration*s contribute members to the declaration space of the namespace. Note that all *using_directive*s must appear before any member declarations.

The *qualified_identifier* of a *namespace_declaration* may be a single identifier or a sequence of identifiers separated by "`.`" tokens. The latter form permits a program to define a nested namespace without lexically nesting several namespace declarations. For example,

```
namespace N1.N2
{
    class A {}

    class B {}
}
```

is semantically equivalent to

```
namespace N1
{
    namespace N2
    {
        class A {}

        class B {}
    }
}
```

Namespaces are open-ended, and two namespace declarations with the same fully qualified name contribute to the same declaration space (Declarations). In the example

```
namespace N1.N2
{
    class A {}
}

namespace N1.N2
{
    class B {}
}
```

the two namespace declarations above contribute to the same declaration space, in this case declaring two classes with the fully qualified names `N1.N2.A` and `N1.N2.B`. Because the two declarations contribute to the same declaration space, it would have been an error if each contained a declaration of a member with the same name.

Extern aliases

An *extern_alias_directive* introduces an identifier that serves as an alias for a namespace. The specification of the aliased namespace is external to the source code of the program and applies also to nested namespaces of the aliased namespace.

```
extern_alias_directive
    : 'extern' 'alias' identifier ';'
    ;
```

The scope of an *extern_alias_directive* extends over the *using_directive*s, *global_attributes* and *namespace_member_declaration*s of its immediately containing compilation unit or namespace body.

Within a compilation unit or namespace body that contains an *extern_alias_directive*, the identifier introduced by the *extern_alias_directive* can be used to reference the aliased namespace. It is a compile-time error for the *identifier* to be the word `global`.

An *extern_alias_directive* makes an alias available within a particular compilation unit or namespace body, but it does not contribute any new members to the underlying declaration space. In other words, an *extern_alias_directive* is not transitive, but, rather, affects only the compilation unit or namespace body in which it occurs.

The following program declares and uses two extern aliases, `X` and `Y`, each of which represent the root of a distinct namespace hierarchy:

```
extern alias X;
extern alias Y;

class Test
{
    X::N.A a;
    X::N.B b1;
    Y::N.B b2;
    Y::N.C c;
}
```

The program declares the existence of the extern aliases `X` and `Y`, but the actual definitions of the aliases are external to the program. The identically named `N.B` classes can now be referenced as `X.N.B` and `Y.N.B`, or, using the namespace alias qualifier, `X::N.B` and `Y::N.B`. An error occurs if a program declares an extern alias for which no external definition is provided.

Using directives

Using directives facilitate the use of namespaces and types defined in other namespaces. Using directives impact the name resolution process of *namespace_or_type_name*s (Namespace and type names) and *simple_name*s (Simple names), but unlike declarations, using directives do not contribute new members to the underlying declaration spaces of the compilation units or namespaces within which they are used.

```
using_directive
    : using_alias_directive
    | using_namespace_directive
    | using_static_directive
    ;
```

A *using_alias_directive* (Using alias directives) introduces an alias for a namespace or type.

A *using_namespace_directive* (Using namespace directives) imports the type members of a namespace.

A *using_static_directive* (Using static directives) imports the nested types and static members of a type.

The scope of a *using_directive* extends over the *namespace_member_declaration*s of its immediately containing compilation unit or namespace body. The scope of a *using_directive* specifically does not include its peer *using_directive*s. Thus, peer *using_directive*s do not affect each other, and the order in which they are written is insignificant.

Using alias directives

A *using_alias_directive* introduces an identifier that serves as an alias for a namespace or type within the immediately enclosing compilation unit or namespace body.

```
using_alias_directive
    : 'using' identifier '=' namespace_or_type_name ';'
    ;
```

Within member declarations in a compilation unit or namespace body that contains a *using_alias_directive*, the identifier introduced by the *using_alias_directive* can be used to reference the given namespace or type. For example:

```
namespace N1.N2
{
    class A {}
}

namespace N3
{
    using A = N1.N2.A;

    class B: A {}
}
```

Above, within member declarations in the `N3` namespace, `A` is an alias for `N1.N2.A`, and thus class `N3.B` derives from class `N1.N2.A`. The same effect can be obtained by creating an alias `R` for `N1.N2` and then referencing `R.A`:

```
namespace N3
{
    using R = N1.N2;

    class B: R.A {}
}
```

The *identifier* of a *using_alias_directive* must be unique within the declaration space of the compilation unit or namespace that immediately contains the *using_alias_directive*. For example:

```
namespace N3
{
    class A {}
}

namespace N3
{
    using A = N1.N2.A;        // Error, A already exists
}
```

Above, `N3` already contains a member `A`, so it is a compile-time error for a *using_alias_directive* to use that identifier. Likewise, it is a compile-time error for two or more *using_alias_directive*s in the same compilation unit or namespace body to declare aliases by the same name.

A *using_alias_directive* makes an alias available within a particular compilation unit or namespace body, but it does not contribute any new members to the underlying declaration space. In other words, a *using_alias_directive* is not transitive but rather affects only the compilation unit or namespace body in which it occurs. In the example

```
namespace N3
{
    using R = N1.N2;
}

namespace N3
{
    class B: R.A {}            // Error, R unknown
}
```

the scope of the *using_alias_directive* that introduces `R` only extends to member declarations in the namespace body in which it is contained, so `R` is unknown in the second namespace declaration. However, placing the *using_alias_directive* in the containing compilation unit causes the alias to become available within both namespace declarations:

```
using R = N1.N2;

namespace N3
{
    class B: R.A {}
}

namespace N3
{
    class C: R.A {}
}
```

Just like regular members, names introduced by *using_alias_directive*s are hidden by similarly named members in nested scopes. In the example

```
using R = N1.N2;

namespace N3
{
    class R {}

    class B: R.A {}        // Error, R has no member A
}
```

the reference to `R.A` in the declaration of `B` causes a compile-time error because `R` refers to `N3.R`, not `N1.N2`.

The order in which *using_alias_directive*s are written has no significance, and resolution of the *namespace_or_type_name* referenced by a *using_alias_directive* is not affected by the *using_alias_directive* itself or by other *using_directive*s in the immediately containing compilation unit or namespace body. In other words, the *namespace_or_type_name* of a *using_alias_directive* is resolved as if the immediately containing compilation unit or namespace body had no *using_directive*s. A *using_alias_directive* may however be affected by *extern_alias_directive*s in the immediately containing compilation unit or namespace body. In the example

```
namespace N1.N2 {}

namespace N3
{
    extern alias E;

    using R1 = E.N;        // OK

    using R2 = N1;         // OK

    using R3 = N1.N2;      // OK

    using R4 = R2.N2;      // Error, R2 unknown
}
```

the last *using_alias_directive* results in a compile-time error because it is not affected by the first *using_alias_directive*. The first *using_alias_directive* does not result in an error since the scope of the extern alias `E` includes the *using_alias_directive*.

A *using_alias_directive* can create an alias for any namespace or type, including the namespace within which it appears and any namespace or type nested within that namespace.

Accessing a namespace or type through an alias yields exactly the same result as accessing that namespace or type through its declared name. For example, given

```
namespace N1.N2
{
    class A {}
}

namespace N3
{
    using R1 = N1;
    using R2 = N1.N2;

    class B
    {
        N1.N2.A a;            // refers to N1.N2.A
        R1.N2.A b;            // refers to N1.N2.A
        R2.A c;               // refers to N1.N2.A
    }
}
```

the names `N1.N2.A`, `R1.N2.A`, and `R2.A` are equivalent and all refer to the class whose fully qualified name is `N1.N2.A`.

Using aliases can name a closed constructed type, but cannot name an unbound generic type declaration without supplying type arguments. For example:

```
namespace N1
{
    class A<T>
    {
        class B {}
    }
}

namespace N2
{
    using W = N1.A;          // Error, cannot name unbound generic type

    using X = N1.A.B;        // Error, cannot name unbound generic type

    using Y = N1.A<int>;     // Ok, can name closed constructed type

    using Z<T> = N1.A<T>;    // Error, using alias cannot have type parameters
}
```

Using namespace directives

A *using_namespace_directive* imports the types contained in a namespace into the immediately enclosing compilation unit or namespace body, enabling the identifier of each type to be used without qualification.

```
using_namespace_directive
    : 'using' namespace_name ';'
    ;
```

Within member declarations in a compilation unit or namespace body that contains a *using_namespace_directive*, the types contained in the given namespace can be referenced directly. For example:

```
namespace N1.N2
{
    class A {}
}

namespace N3
{
    using N1.N2;

    class B: A {}
}
```

Above, within member declarations in the `N3` namespace, the type members of `N1.N2` are directly available, and thus class `N3.B` derives from class `N1.N2.A`.

A *using_namespace_directive* imports the types contained in the given namespace, but specifically does not import nested namespaces. In the example

```
namespace N1.N2
{
    class A {}
}

namespace N3
{
    using N1;

    class B: N2.A {}        // Error, N2 unknown
}
```

the *using_namespace_directive* imports the types contained in `N1`, but not the namespaces nested in `N1`. Thus, the reference to `N2.A` in the declaration of `B` results in a compile-time error because no members named `N2` are in scope.

Unlike a *using_alias_directive*, a *using_namespace_directive* may import types whose identifiers are already defined within the enclosing compilation unit or namespace body. In effect, names imported by a *using_namespace_directive* are hidden by similarly named members in the enclosing compilation unit or namespace body. For example:

```
namespace N1.N2
{
    class A {}

    class B {}
}

namespace N3
{
    using N1.N2;

    class A {}
}
```

Here, within member declarations in the `N3` namespace, `A` refers to `N3.A` rather than `N1.N2.A`.

When more than one namespace or type imported by *using_namespace_directive*s or *using_static_directive*s in the same compilation unit or namespace body contain types by the same name, references to that name as a *type_name* are considered ambiguous. In the example

```
namespace N1
{
    class A {}
}

namespace N2
{
    class A {}
}

namespace N3
{
    using N1;

    using N2;

    class B: A {}               // Error, A is ambiguous
}
```

both `N1` and `N2` contain a member `A`, and because `N3` imports both, referencing `A` in `N3` is a compile-time error. In this situation, the conflict can be resolved either through qualification of references to `A`, or by introducing a *using_alias_directive* that picks a particular `A`. For example:

```
namespace N3
{
    using N1;

    using N2;

    using A = N1.A;

    class B: A {}               // A means N1.A
}
```

Furthermore, when more than one namespace or type imported by *using_namespace_directive*s or *using_static_directive*s in the same compilation unit or namespace body contain types or members by the same name, references to that name as a *simple_name* are considered ambiguous. In the example

```
namespace N1
{
 class A {}
}

class C
{
 public static int A;
}

namespace N2
{
 using N1;
 using static C;

 class B
 {
  void M()
  {
   A a = new A();   // Ok, A is unambiguous as a type-name
   A.Equals(2);     // Error, A is ambiguous as a simple-name
  }
 }
}
```

`N1` contains a type member `A`, and `C` contains a static field `A`, and because `N2` imports both, referencing `A` as a *simple_name* is ambiguous and a compile-time error.

Like a *using_alias_directive*, a *using_namespace_directive* does not contribute any new members to the underlying declaration space of the compilation unit or namespace, but rather affects only the compilation unit or namespace body in which it appears.

The *namespace_name* referenced by a *using_namespace_directive* is resolved in the same way as the *namespace_or_type_name* referenced by a *using_alias_directive*. Thus, *using_namespace_directive*s in the same compilation unit or namespace body do not affect each other and can be written in any order.

Using static directives

A *using_static_directive* imports the nested types and static members contained directly in a type declaration into the immediately enclosing compilation unit or namespace body, enabling the identifier of each member and type to be used without qualification.

```
using_static_directive
    : 'using' 'static' type_name ';'
    ;
```

Within member declarations in a compilation unit or namespace body that contains a *using_static_directive*, the accessible nested types and static members (except extension methods) contained directly in the declaration of the given type can be referenced directly. For example:

```
namespace N1
{
 class A
 {
  public class B{}
  public static B M(){ return new B(); }
 }
}

namespace N2
{
 using static N1.A;
 class C
 {
  void N() { B b = M(); }
 }
}
```

Above, within member declarations in the `N2` namespace, the static members and nested types of `N1.A` are directly available, and thus the method `N` is able to reference both the `B` and `M` members of `N1.A`.

A *using_static_directive* specifically does not import extension methods directly as static methods, but makes them available for extension method invocation (Extension method invocations). In the example

```
namespace N1
{
 static class A
 {
  public static void M(this string s){}
 }
}

namespace N2
{
 using static N1.A;

 class B
 {
  void N()
  {
   M("A");      // Error, M unknown
   "B".M();     // Ok, M known as extension method
   N1.A.M("C"); // Ok, fully qualified
  }
 }
}
```

the *using_static_directive* imports the extension method `M` contained in `N1.A`, but only as an extension method. Thus, the first reference to `M` in the body of `B.N` results in a compile-time error because no members named `M` are in scope.

A *using_static_directive* only imports members and types declared directly in the given type, not members and types declared in base classes.

TODO: Example

Ambiguities between multiple *using_namespace_directives* and *using_static_directives* are discussed in Using namespace directives.

Namespace members

A *namespace_member_declaration* is either a *namespace_declaration* (Namespace declarations) or a *type_declaration* (Type declarations).

```
namespace_member_declaration
    : namespace_declaration
    | type_declaration
    ;
```

A compilation unit or a namespace body can contain *namespace_member_declaration*s, and such declarations contribute new members to the underlying declaration space of the containing compilation unit or namespace body.

Type declarations

A *type_declaration* is a *class_declaration* (Class declarations), a *struct_declaration* (Struct declarations), an *interface_declaration* (Interface declarations), an *enum_declaration* (Enum declarations), or a *delegate_declaration* (Delegate declarations).

```
type_declaration
    : class_declaration
    | struct_declaration
    | interface_declaration
    | enum_declaration
    | delegate_declaration
    ;
```

A *type_declaration* can occur as a top-level declaration in a compilation unit or as a member declaration within a namespace, class, or struct.

When a type declaration for a type `T` occurs as a top-level declaration in a compilation unit, the fully qualified name of the newly declared type is simply `T`. When a type declaration for a type `T` occurs within a namespace, class, or struct, the fully qualified name of the newly declared type is `N.T`, where `N` is the fully qualified name of the containing namespace, class, or struct.

A type declared within a class or struct is called a nested type (Nested types).

The permitted access modifiers and the default access for a type declaration depend on the context in which the declaration takes place (Declared accessibility):

- Types declared in compilation units or namespaces can have `public` or `internal` access. The default is `internal` access.
- Types declared in classes can have `public`, `protected internal`, `protected`, `internal`, or `private` access. The default is `private` access.
- Types declared in structs can have `public`, `internal`, or `private` access. The default is `private` access.

Namespace alias qualifiers

The ***namespace alias qualifier*** `::` makes it possible to guarantee that type name lookups are unaffected by the introduction of new types and members. The namespace alias qualifier always appears between two identifiers referred to as the left-hand and right-hand identifiers. Unlike the regular `.` qualifier, the left-hand identifier of the `::` qualifier is looked up only as an extern or using alias.

A *qualified_alias_member* is defined as follows:

```
qualified_alias_member
    : identifier '::' identifier type_argument_list?
    ;
```

A *qualified_alias_member* can be used as a *namespace_or_type_name* (Namespace and type names) or as the left operand in a *member_access* (Member access).

A *qualified_alias_member* has one of two forms:

- `N::I<A1, ..., Ak>`, where `N` and `I` represent identifiers, and `<A1, ..., Ak>` is a type argument list. (`K` is always at least one.)
- `N::I`, where `N` and `I` represent identifiers. (In this case, `K` is considered to be zero.)

Using this notation, the meaning of a *qualified_alias_member* is determined as follows:

- If `N` is the identifier `global`, then the global namespace is searched for `I`:
  - If the global namespace contains a namespace named `I` and `K` is zero, then the *qualified_alias_member* refers to that namespace.
  - Otherwise, if the global namespace contains a non-generic type named `I` and `K` is zero, then the *qualified_alias_member* refers to that type.
  - Otherwise, if the global namespace contains a type named `I` that has `K` type parameters, then the *qualified_alias_member* refers to that type constructed with the given type arguments.
  - Otherwise, the *qualified_alias_member* is undefined and a compile-time error occurs.
- Otherwise, starting with the namespace declaration (Namespace declarations) immediately containing the *qualified_alias_member* (if any), continuing with each enclosing namespace declaration (if any), and ending with the compilation unit containing the *qualified_alias_member*, the following steps are evaluated until an entity is located:
  - If the namespace declaration or compilation unit contains a *using_alias_directive* that associates `N` with a type, then the *qualified_alias_member* is undefined and a compile-time error occurs.
  - Otherwise, if the namespace declaration or compilation unit contains an *extern_alias_directive* or *using_alias_directive* that associates `N` with a namespace, then:
    - If the namespace associated with `N` contains a namespace named `I` and `K` is zero, then the *qualified_alias_member* refers to that namespace.
    - Otherwise, if the namespace associated with `N` contains a non-generic type named `I` and `K` is zero, then the *qualified_alias_member* refers to that type.
    - Otherwise, if the namespace associated with `N` contains a type named `I` that has `K` type parameters, then the *qualified_alias_member* refers to that type constructed with the given type arguments.
    - Otherwise, the *qualified_alias_member* is undefined and a compile-time error occurs.
- Otherwise, the *qualified_alias_member* is undefined and a compile-time error occurs.

Note that using the namespace alias qualifier with an alias that references a type causes a compile-time error. Also note that if the identifier `N` is `global`, then lookup is performed in the global namespace, even if there is a using alias associating `global` with a type or namespace.

Uniqueness of aliases

Each compilation unit and namespace body has a separate declaration space for extern aliases and using aliases. Thus, while the name of an extern alias or using alias must be unique within the set of extern aliases and using aliases declared in the immediately containing compilation unit or namespace body, an alias is permitted to have the same name as a type or namespace as long as it is used only with the `::` qualifier.

In the example

```
namespace N
{
    public class A {}

    public class B {}
}

namespace N
{
    using A = System.IO;

    class X
    {
        A.Stream s1;            // Error, A is ambiguous

        A::Stream s2;           // Ok
    }
}
```

the name `A` has two possible meanings in the second namespace body because both the class `A` and the using alias `A` are in scope. For this reason, use of `A` in the qualified name `A.Stream` is ambiguous and causes a compile-time error to occur. However, use of `A` with the `::` qualifier is not an error because `A` is looked up only as a namespace alias.

Classes

18.12.2021 • 170 minutes to read

A class is a data structure that may contain data members (constants and fields), function members (methods, properties, events, indexers, operators, instance constructors, destructors and static constructors), and nested types. Class types support inheritance, a mechanism whereby a derived class can extend and specialize a base class.

Class declarations

A *class_declaration* is a *type_declaration* (Type declarations) that declares a new class.

```
class_declaration
    : attributes? class_modifier* 'partial'? 'class' identifier type_parameter_list?
      class_base? type_parameter_constraints_clause* class_body ';'?
    ;
```

A *class_declaration* consists of an optional set of *attributes* (Attributes), followed by an optional set of *class_modifier*s (Class modifiers), followed by an optional `partial` modifier, followed by the keyword `class` and an *identifier* that names the class, followed by an optional *type_parameter_list* (Type parameters), followed by an optional *class_base* specification (Class base specification) , followed by an optional set of *type_parameter_constraints_clause*s (Type parameter constraints), followed by a *class_body* (Class body), optionally followed by a semicolon.

A class declaration cannot supply *type_parameter_constraints_clause*s unless it also supplies a *type_parameter_list*.

A class declaration that supplies a *type_parameter_list* is a ***generic class declaration***. Additionally, any class nested inside a generic class declaration or a generic struct declaration is itself a generic class declaration, since type parameters for the containing type must be supplied to create a constructed type.

Class modifiers

A *class_declaration* may optionally include a sequence of class modifiers:

```
class_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'abstract'
    | 'sealed'
    | 'static'
    | class_modifier_unsafe
    ;
```

It is a compile-time error for the same modifier to appear multiple times in a class declaration.

The `new` modifier is permitted on nested classes. It specifies that the class hides an inherited member by the same name, as described in The new modifier. It is a compile-time error for the `new` modifier to appear on a class declaration that is not a nested class declaration.

The `public`, `protected`, `internal`, and `private` modifiers control the accessibility of the class. Depending on

the context in which the class declaration occurs, some of these modifiers may not be permitted (Declared accessibility).

The `abstract`, `sealed` and `static` modifiers are discussed in the following sections.

Abstract classes

The `abstract` modifier is used to indicate that a class is incomplete and that it is intended to be used only as a base class. An abstract class differs from a non-abstract class in the following ways:

- An abstract class cannot be instantiated directly, and it is a compile-time error to use the `new` operator on an abstract class. While it is possible to have variables and values whose compile-time types are abstract, such variables and values will necessarily either be `null` or contain references to instances of non-abstract classes derived from the abstract types.
- An abstract class is permitted (but not required) to contain abstract members.
- An abstract class cannot be sealed.

When a non-abstract class is derived from an abstract class, the non-abstract class must include actual implementations of all inherited abstract members, thereby overriding those abstract members. In the example

```
abstract class A
{
    public abstract void F();
}

abstract class B: A
{
    public void G() {}
}

class C: B
{
    public override void F() {
        // actual implementation of F
    }
}
```

the abstract class `A` introduces an abstract method `F`. Class `B` introduces an additional method `G`, but since it doesn't provide an implementation of `F`, `B` must also be declared abstract. Class `C` overrides `F` and provides an actual implementation. Since there are no abstract members in `C`, `C` is permitted (but not required) to be non-abstract.

Sealed classes

The `sealed` modifier is used to prevent derivation from a class. A compile-time error occurs if a sealed class is specified as the base class of another class.

A sealed class cannot also be an abstract class.

The `sealed` modifier is primarily used to prevent unintended derivation, but it also enables certain run-time optimizations. In particular, because a sealed class is known to never have any derived classes, it is possible to transform virtual function member invocations on sealed class instances into non-virtual invocations.

Static classes

The `static` modifier is used to mark the class being declared as a ***static class***. A static class cannot be instantiated, cannot be used as a type and can contain only static members. Only a static class can contain declarations of extension methods (Extension methods).

A static class declaration is subject to the following restrictions:

- A static class may not include a `sealed` or `abstract` modifier. Note, however, that since a static class cannot

be instantiated or derived from, it behaves as if it was both sealed and abstract.

- A static class may not include a *class_base* specification (Class base specification) and cannot explicitly specify a base class or a list of implemented interfaces. A static class implicitly inherits from type `object`.
- A static class can only contain static members (Static and instance members). Note that constants and nested types are classified as static members.
- A static class cannot have members with `protected` or `protected internal` declared accessibility.

It is a compile-time error to violate any of these restrictions.

A static class has no instance constructors. It is not possible to declare an instance constructor in a static class, and no default instance constructor (Default constructors) is provided for a static class.

The members of a static class are not automatically static, and the member declarations must explicitly include a `static` modifier (except for constants and nested types). When a class is nested within a static outer class, the nested class is not a static class unless it explicitly includes a `static` modifier.

Referencing static class types

A *namespace_or_type_name* (Namespace and type names) is permitted to reference a static class if

- The *namespace_or_type_name* is the `T` in a *namespace_or_type_name* of the form `T.I`, or
- The *namespace_or_type_name* is the `T` in a *typeof_expression* (Argument lists1) of the form `typeof(T)`.

A *primary_expression* (Function members) is permitted to reference a static class if

- The *primary_expression* is the `E` in a *member_access* (Compile-time checking of dynamic overload resolution) of the form `E.I`.

In any other context it is a compile-time error to reference a static class. For example, it is an error for a static class to be used as a base class, a constituent type (Nested types) of a member, a generic type argument, or a type parameter constraint. Likewise, a static class cannot be used in an array type, a pointer type, a `new` expression, a cast expression, an `is` expression, an `as` expression, a `sizeof` expression, or a default value expression.

Partial modifier

The `partial` modifier is used to indicate that this *class_declaration* is a partial type declaration. Multiple partial type declarations with the same name within an enclosing namespace or type declaration combine to form one type declaration, following the rules specified in Partial types.

Having the declaration of a class distributed over separate segments of program text can be useful if these segments are produced or maintained in different contexts. For instance, one part of a class declaration may be machine generated, whereas the other is manually authored. Textual separation of the two prevents updates by one from conflicting with updates by the other.

Type parameters

A type parameter is a simple identifier that denotes a placeholder for a type argument supplied to create a constructed type. A type parameter is a formal placeholder for a type that will be supplied later. By contrast, a type argument (Type arguments) is the actual type that is substituted for the type parameter when a constructed type is created.

```
type_parameter_list
    : '<' type_parameters '>'
    ;

type_parameters
    : attributes? type_parameter
    | type_parameters ',' attributes? type_parameter
    ;

type_parameter
    : identifier
    ;
```

Each type parameter in a class declaration defines a name in the declaration space (Declarations) of that class. Thus, it cannot have the same name as another type parameter or a member declared in that class. A type parameter cannot have the same name as the type itself.

Class base specification

A class declaration may include a *class_base* specification, which defines the direct base class of the class and the interfaces (Interfaces) directly implemented by the class.

```
class_base
    : ':' class_type
    | ':' interface_type_list
    | ':' class_type ',' interface_type_list
    ;

interface_type_list
    : interface_type (',' interface_type)*
    ;
```

The base class specified in a class declaration can be a constructed class type (Constructed types). A base class cannot be a type parameter on its own, though it can involve the type parameters that are in scope.

```
class Extend<V>: V {}           // Error, type parameter used as base class
```

Base classes

When a *class_type* is included in the *class_base*, it specifies the direct base class of the class being declared. If a class declaration has no *class_base*, or if the *class_base* lists only interface types, the direct base class is assumed to be `object`. A class inherits members from its direct base class, as described in Inheritance.

In the example

```
class A {}

class B: A {}
```

class `A` is said to be the direct base class of `B`, and `B` is said to be derived from `A`. Since `A` does not explicitly specify a direct base class, its direct base class is implicitly `object`.

For a constructed class type, if a base class is specified in the generic class declaration, the base class of the constructed type is obtained by substituting, for each *type_parameter* in the base class declaration, the corresponding *type_argument* of the constructed type. Given the generic class declarations

```
class B<U,V> {...}

class G<T>: B<string,T[]> {...}
```

the base class of the constructed type `G<int>` would be `B<string,int[]>`.

The direct base class of a class type must be at least as accessible as the class type itself (Accessibility domains). For example, it is a compile-time error for a `public` class to derive from a `private` or `internal` class.

The direct base class of a class type must not be any of the following types: `System.Array`, `System.Delegate`, `System.MulticastDelegate`, `System.Enum`, or `System.ValueType`. Furthermore, a generic class declaration cannot use `System.Attribute` as a direct or indirect base class.

While determining the meaning of the direct base class specification `A` of a class `B`, the direct base class of `B` is temporarily assumed to be `object`. Intuitively this ensures that the meaning of a base class specification cannot recursively depend on itself. The example:

```
class A<T> {
   public class B {}
}

class C : A<C.B> {}
```

is in error since in the base class specification `A<C.B>` the direct base class of `C` is considered to be `object`, and hence (by the rules of Namespace and type names) `C` is not considered to have a member `B`.

The base classes of a class type are the direct base class and its base classes. In other words, the set of base classes is the transitive closure of the direct base class relationship. Referring to the example above, the base classes of `B` are `A` and `object`. In the example

```
class A {...}

class B<T>: A {...}

class C<T>: B<IComparable<T>> {...}

class D<T>: C<T[]> {...}
```

the base classes of `D<int>` are `C<int[]>`, `B<IComparable<int[]>>`, `A`, and `object`.

Except for class `object`, every class type has exactly one direct base class. The `object` class has no direct base class and is the ultimate base class of all other classes.

When a class `B` derives from a class `A`, it is a compile-time error for `A` to depend on `B`. A class ***directly depends on*** its direct base class (if any) and ***directly depends on*** the class within which it is immediately nested (if any). Given this definition, the complete set of classes upon which a class depends is the reflexive and transitive closure of the ***directly depends on*** relationship.

The example

```
class A: A {}
```

is erroneous because the class depends on itself. Likewise, the example

```
class A: B {}
class B: C {}
class C: A {}
```

is in error because the classes circularly depend on themselves. Finally, the example

```
class A: B.C {}

class B: A
{
    public class C {}
}
```

results in a compile-time error because `A` depends on `B.C` (its direct base class), which depends on `B` (its immediately enclosing class), which circularly depends on `A`.

Note that a class does not depend on the classes that are nested within it. In the example

```
class A
{
    class B: A {}
}
```

`B` depends on `A` (because `A` is both its direct base class and its immediately enclosing class), but `A` does not depend on `B` (since `B` is neither a base class nor an enclosing class of `A`). Thus, the example is valid.

It is not possible to derive from a `sealed` class. In the example

```
sealed class A {}

class B: A {}           // Error, cannot derive from a sealed class
```

class `B` is in error because it attempts to derive from the `sealed` class `A`.

Interface implementations

A *class_base* specification may include a list of interface types, in which case the class is said to directly implement the given interface types. Interface implementations are discussed further in Interface implementations.

Type parameter constraints

Generic type and method declarations can optionally specify type parameter constraints by including *type_parameter_constraints_clause*s.

```
type_parameter_constraints_clause
    : 'where' type_parameter ':' type_parameter_constraints
    ;

type_parameter_constraints
    : primary_constraint
    | secondary_constraints
    | constructor_constraint
    | primary_constraint ',' secondary_constraints
    | primary_constraint ',' constructor_constraint
    | secondary_constraints ',' constructor_constraint
    | primary_constraint ',' secondary_constraints ',' constructor_constraint
    ;

primary_constraint
    : class_type
    | 'class'
    | 'struct'
    ;

secondary_constraints
    : interface_type
    | type_parameter
    | secondary_constraints ',' interface_type
    | secondary_constraints ',' type_parameter
    ;

constructor_constraint
    : 'new' '(' ')'
    ;
```

Each *type_parameter_constraints_clause* consists of the token `where`, followed by the name of a type parameter, followed by a colon and the list of constraints for that type parameter. There can be at most one `where` clause for each type parameter, and the `where` clauses can be listed in any order. Like the `get` and `set` tokens in a property accessor, the `where` token is not a keyword.

The list of constraints given in a `where` clause can include any of the following components, in this order: a single primary constraint, one or more secondary constraints, and the constructor constraint, `new()`.

A primary constraint can be a class type or the ***reference type constraint*** `class` or the ***value type constraint*** `struct`. A secondary constraint can be a *type_parameter* or *interface_type*.

The reference type constraint specifies that a type argument used for the type parameter must be a reference type. All class types, interface types, delegate types, array types, and type parameters known to be a reference type (as defined below) satisfy this constraint.

The value type constraint specifies that a type argument used for the type parameter must be a non-nullable value type. All non-nullable struct types, enum types, and type parameters having the value type constraint satisfy this constraint. Note that although classified as a value type, a nullable type (Nullable types) does not satisfy the value type constraint. A type parameter having the value type constraint cannot also have the *constructor_constraint*.

Pointer types are never allowed to be type arguments and are not considered to satisfy either the reference type or value type constraints.

If a constraint is a class type, an interface type, or a type parameter, that type specifies a minimal "base type" that every type argument used for that type parameter must support. Whenever a constructed type or generic method is used, the type argument is checked against the constraints on the type parameter at compile-time. The type argument supplied must satisfy the conditions described in Satisfying constraints.

A *class_type* constraint must satisfy the following rules:

- The type must be a class type.
- The type must not be `sealed`.
- The type must not be one of the following types: `System.Array`, `System.Delegate`, `System.Enum`, or `System.ValueType`.
- The type must not be `object`. Because all types derive from `object`, such a constraint would have no effect if it were permitted.
- At most one constraint for a given type parameter can be a class type.

A type specified as an *interface_type* constraint must satisfy the following rules:

- The type must be an interface type.
- A type must not be specified more than once in a given `where` clause.

In either case, the constraint can involve any of the type parameters of the associated type or method declaration as part of a constructed type, and can involve the type being declared.

Any class or interface type specified as a type parameter constraint must be at least as accessible (Accessibility constraints) as the generic type or method being declared.

A type specified as a *type_parameter* constraint must satisfy the following rules:

- The type must be a type parameter.
- A type must not be specified more than once in a given `where` clause.

In addition there must be no cycles in the dependency graph of type parameters, where dependency is a transitive relation defined by:

- If a type parameter `T` is used as a constraint for type parameter `S` then `S` ***depends on*** `T`.
- If a type parameter `S` depends on a type parameter `T` and `T` depends on a type parameter `U` then `S` ***depends on*** `U`.

Given this relation, it is a compile-time error for a type parameter to depend on itself (directly or indirectly).

Any constraints must be consistent among dependent type parameters. If type parameter `S` depends on type parameter `T` then:

- `T` must not have the value type constraint. Otherwise, `T` is effectively sealed so `S` would be forced to be the same type as `T`, eliminating the need for two type parameters.
- If `S` has the value type constraint then `T` must not have a *class_type* constraint.
- If `S` has a *class_type* constraint `A` and `T` has a *class_type* constraint `B` then there must be an identity conversion or implicit reference conversion from `A` to `B` or an implicit reference conversion from `B` to `A`.
- If `S` also depends on type parameter `U` and `U` has a *class_type* constraint `A` and `T` has a *class_type* constraint `B` then there must be an identity conversion or implicit reference conversion from `A` to `B` or an implicit reference conversion from `B` to `A`.

It is valid for `S` to have the value type constraint and `T` to have the reference type constraint. Effectively this limits `T` to the types `System.Object`, `System.ValueType`, `System.Enum`, and any interface type.

If the `where` clause for a type parameter includes a constructor constraint (which has the form `new()`), it is possible to use the `new` operator to create instances of the type (Object creation expressions). Any type argument used for a type parameter with a constructor constraint must have a public parameterless constructor (this constructor implicitly exists for any value type) or be a type parameter having the value type constraint or constructor constraint (see Type parameter constraints for details).

The following are examples of constraints:

```
interface IPrintable
{
    void Print();
}

interface IComparable<T>
{
    int CompareTo(T value);
}

interface IKeyProvider<T>
{
    T GetKey();
}

class Printer<T> where T: IPrintable {...}

class SortedList<T> where T: IComparable<T> {...}

class Dictionary<K,V>
    where K: IComparable<K>
    where V: IPrintable, IKeyProvider<K>, new()
{
    ...
}
```

The following example is in error because it causes a circularity in the dependency graph of the type parameters:

```
class Circular<S,T>
    where S: T
    where T: S                // Error, circularity in dependency graph
{
    ...
}
```

The following examples illustrate additional invalid situations:

```
class Sealed<S,T>
    where S: T
    where T: struct        // Error, T is sealed
{
    ...
}

class A {...}

class B {...}

class Incompat<S,T>
    where S: A, T
    where T: B                // Error, incompatible class-type constraints
{
    ...
}

class StructWithClass<S,T,U>
    where S: struct, T
    where T: U
    where U: A                // Error, A incompatible with struct
{
    ...
}
```

The ***effective base class*** of a type parameter `T` is defined as follows:

- If `T` has no primary constraints or type parameter constraints, its effective base class is `object`.
- If `T` has the value type constraint, its effective base class is `System.ValueType`.
- If `T` has a *class_type* constraint `C` but no *type_parameter* constraints, its effective base class is `C`.
- If `T` has no *class_type* constraint but has one or more *type_parameter* constraints, its effective base class is the most encompassed type (Lifted conversion operators) in the set of effective base classes of its *type_parameter* constraints. The consistency rules ensure that such a most encompassed type exists.
- If `T` has both a *class_type* constraint and one or more *type_parameter* constraints, its effective base class is the most encompassed type (Lifted conversion operators) in the set consisting of the *class_type* constraint of `T` and the effective base classes of its *type_parameter* constraints. The consistency rules ensure that such a most encompassed type exists.
- If `T` has the reference type constraint but no *class_type* constraints, its effective base class is `object`.

For the purpose of these rules, if T has a constraint `V` that is a *value_type*, use instead the most specific base type of `V` that is a *class_type*. This can never happen in an explicitly given constraint, but may occur when the constraints of a generic method are implicitly inherited by an overriding method declaration or an explicit implementation of an interface method.

These rules ensure that the effective base class is always a *class_type*.

The ***effective interface set*** of a type parameter `T` is defined as follows:

- If `T` has no *secondary_constraints*, its effective interface set is empty.
- If `T` has *interface_type* constraints but no *type_parameter* constraints, its effective interface set is its set of *interface_type* constraints.
- If `T` has no *interface_type* constraints but has *type_parameter* constraints, its effective interface set is the union of the effective interface sets of its *type_parameter* constraints.
- If `T` has both *interface_type* constraints and *type_parameter* constraints, its effective interface set is the union of its set of *interface_type* constraints and the effective interface sets of its *type_parameter* constraints.

A type parameter is ***known to be a reference type*** if it has the reference type constraint or its effective base class is not `object` or `System.ValueType`.

Values of a constrained type parameter type can be used to access the instance members implied by the constraints. In the example

```
interface IPrintable
{
    void Print();
}

class Printer<T> where T: IPrintable
{
    void PrintOne(T x) {
        x.Print();
    }
}
```

the methods of `IPrintable` can be invoked directly on `x` because `T` is constrained to always implement `IPrintable`.

Class body

The *class_body* of a class defines the members of that class.

```
class_body
    : '{' class_member_declaration* '}'
    ;
```

Partial types

A type declaration can be split across multiple ***partial type declarations***. The type declaration is constructed from its parts by following the rules in this section, whereupon it is treated as a single declaration during the remainder of the compile-time and run-time processing of the program.

A *class_declaration*, *struct_declaration* or *interface_declaration* represents a partial type declaration if it includes a `partial` modifier. `partial` is not a keyword, and only acts as a modifier if it appears immediately before one of the keywords `class`, `struct` or `interface` in a type declaration, or before the type `void` in a method declaration. In other contexts it can be used as a normal identifier.

Each part of a partial type declaration must include a `partial` modifier. It must have the same name and be declared in the same namespace or type declaration as the other parts. The `partial` modifier indicates that additional parts of the type declaration may exist elsewhere, but the existence of such additional parts is not a requirement; it is valid for a type with a single declaration to include the `partial` modifier.

All parts of a partial type must be compiled together such that the parts can be merged at compile-time into a single type declaration. Partial types specifically do not allow already compiled types to be extended.

Nested types may be declared in multiple parts by using the `partial` modifier. Typically, the containing type is declared using `partial` as well, and each part of the nested type is declared in a different part of the containing type.

The `partial` modifier is not permitted on delegate or enum declarations.

Attributes

The attributes of a partial type are determined by combining, in an unspecified order, the attributes of each of the parts. If an attribute is placed on multiple parts, it is equivalent to specifying the attribute multiple times on the type. For example, the two parts:

```
[Attr1, Attr2("hello")]
partial class A {}

[Attr3, Attr2("goodbye")]
partial class A {}
```

are equivalent to a declaration such as:

```
[Attr1, Attr2("hello"), Attr3, Attr2("goodbye")]
class A {}
```

Attributes on type parameters combine in a similar fashion.

Modifiers

When a partial type declaration includes an accessibility specification (the `public`, `protected`, `internal`, and `private` modifiers) it must agree with all other parts that include an accessibility specification. If no part of a partial type includes an accessibility specification, the type is given the appropriate default accessibility (Declared accessibility).

If one or more partial declarations of a nested type include a `new` modifier, no warning is reported if the nested

type hides an inherited member (Hiding through inheritance).

If one or more partial declarations of a class include an `abstract` modifier, the class is considered abstract (Abstract classes). Otherwise, the class is considered non-abstract.

If one or more partial declarations of a class include a `sealed` modifier, the class is considered sealed (Sealed classes). Otherwise, the class is considered unsealed.

Note that a class cannot be both abstract and sealed.

When the `unsafe` modifier is used on a partial type declaration, only that particular part is considered an unsafe context (Unsafe contexts).

Type parameters and constraints

If a generic type is declared in multiple parts, each part must state the type parameters. Each part must have the same number of type parameters, and the same name for each type parameter, in order.

When a partial generic type declaration includes constraints (`where` clauses), the constraints must agree with all other parts that include constraints. Specifically, each part that includes constraints must have constraints for the same set of type parameters, and for each type parameter the sets of primary, secondary, and constructor constraints must be equivalent. Two sets of constraints are equivalent if they contain the same members. If no part of a partial generic type specifies type parameter constraints, the type parameters are considered unconstrained.

The example

```
partial class Dictionary<K,V>
    where K: IComparable<K>
    where V: IKeyProvider<K>, IPersistable
{
    ...
}

partial class Dictionary<K,V>
    where V: IPersistable, IKeyProvider<K>
    where K: IComparable<K>
{
    ...
}

partial class Dictionary<K,V>
{
    ...
}
```

is correct because those parts that include constraints (the first two) effectively specify the same set of primary, secondary, and constructor constraints for the same set of type parameters, respectively.

Base class

When a partial class declaration includes a base class specification it must agree with all other parts that include a base class specification. If no part of a partial class includes a base class specification, the base class becomes `System.Object` (Base classes).

Base interfaces

The set of base interfaces for a type declared in multiple parts is the union of the base interfaces specified on each part. A particular base interface may only be named once on each part, but it is permitted for multiple parts to name the same base interface(s). There must only be one implementation of the members of any given base interface.

In the example

```
partial class C: IA, IB {...}

partial class C: IC {...}

partial class C: IA, IB {...}
```

the set of base interfaces for class `C` is `IA`, `IB`, and `IC`.

Typically, each part provides an implementation of the interface(s) declared on that part; however, this is not a requirement. A part may provide the implementation for an interface declared on a different part:

```
partial class X
{
    int IComparable.CompareTo(object o) {...}
}

partial class X: IComparable
{
    ...
}
```

Members

With the exception of partial methods (Partial methods), the set of members of a type declared in multiple parts is simply the union of the set of members declared in each part. The bodies of all parts of the type declaration share the same declaration space (Declarations), and the scope of each member (Scopes) extends to the bodies of all the parts. The accessibility domain of any member always includes all the parts of the enclosing type; a `private` member declared in one part is freely accessible from another part. It is a compile-time error to declare the same member in more than one part of the type, unless that member is a type with the `partial` modifier.

```
partial class A
{
    int x;                    // Error, cannot declare x more than once

    partial class Inner       // Ok, Inner is a partial type
    {
        int y;
    }
}

partial class A
{
    int x;                    // Error, cannot declare x more than once

    partial class Inner       // Ok, Inner is a partial type
    {
        int z;
    }
}
```

The ordering of members within a type is rarely significant to C# code, but may be significant when interfacing with other languages and environments. In these cases, the ordering of members within a type declared in multiple parts is undefined.

Partial methods

Partial methods can be defined in one part of a type declaration and implemented in another. The

implementation is optional; if no part implements the partial method, the partial method declaration and all calls to it are removed from the type declaration resulting from the combination of the parts.

Partial methods cannot define access modifiers, but are implicitly `private`. Their return type must be `void`, and their parameters cannot have the `out` modifier. The identifier `partial` is recognized as a special keyword in a method declaration only if it appears right before the `void` type; otherwise it can be used as a normal identifier. A partial method cannot explicitly implement interface methods.

There are two kinds of partial method declarations: If the body of the method declaration is a semicolon, the declaration is said to be a ***defining partial method declaration***. If the body is given as a *block*, the declaration is said to be an ***implementing partial method declaration***. Across the parts of a type declaration there can be only one defining partial method declaration with a given signature, and there can be only one implementing partial method declaration with a given signature. If an implementing partial method declaration is given, a corresponding defining partial method declaration must exist, and the declarations must match as specified in the following:

- The declarations must have the same modifiers (although not necessarily in the same order), method name, number of type parameters and number of parameters.
- Corresponding parameters in the declarations must have the same modifiers (although not necessarily in the same order) and the same types (modulo differences in type parameter names).
- Corresponding type parameters in the declarations must have the same constraints (modulo differences in type parameter names).

An implementing partial method declaration can appear in the same part as the corresponding defining partial method declaration.

Only a defining partial method participates in overload resolution. Thus, whether or not an implementing declaration is given, invocation expressions may resolve to invocations of the partial method. Because a partial method always returns `void`, such invocation expressions will always be expression statements. Furthermore, because a partial method is implicitly `private`, such statements will always occur within one of the parts of the type declaration within which the partial method is declared.

If no part of a partial type declaration contains an implementing declaration for a given partial method, any expression statement invoking it is simply removed from the combined type declaration. Thus the invocation expression, including any constituent expressions, has no effect at run-time. The partial method itself is also removed and will not be a member of the combined type declaration.

If an implementing declaration exist for a given partial method, the invocations of the partial methods are retained. The partial method gives rise to a method declaration similar to the implementing partial method declaration except for the following:

- The `partial` modifier is not included
- The attributes in the resulting method declaration are the combined attributes of the defining and the implementing partial method declaration in unspecified order. Duplicates are not removed.
- The attributes on the parameters of the resulting method declaration are the combined attributes of the corresponding parameters of the defining and the implementing partial method declaration in unspecified order. Duplicates are not removed.

If a defining declaration but not an implementing declaration is given for a partial method M, the following restrictions apply:

- It is a compile-time error to create a delegate to method (Delegate creation expressions).
- It is a compile-time error to refer to `M` inside an anonymous function that is converted to an expression tree type (Evaluation of anonymous function conversions to expression tree types).
- Expressions occurring as part of an invocation of `M` do not affect the definite assignment state (Definite

assignment), which can potentially lead to compile-time errors.

- `M` cannot be the entry point for an application (Application Startup).

Partial methods are useful for allowing one part of a type declaration to customize the behavior of another part, e.g., one that is generated by a tool. Consider the following partial class declaration:

```
partial class Customer
{
    string name;

    public string Name {
        get { return name; }
        set {
            OnNameChanging(value);
            name = value;
            OnNameChanged();
        }

    }

    partial void OnNameChanging(string newName);

    partial void OnNameChanged();
}
```

If this class is compiled without any other parts, the defining partial method declarations and their invocations will be removed, and the resulting combined class declaration will be equivalent to the following:

```
class Customer
{
    string name;

    public string Name {
        get { return name; }
        set { name = value; }
    }
}
```

Assume that another part is given, however, which provides implementing declarations of the partial methods:

```
partial class Customer
{
    partial void OnNameChanging(string newName)
    {
        Console.WriteLine("Changing " + name + " to " + newName);
    }

    partial void OnNameChanged()
    {
        Console.WriteLine("Changed to " + name);
    }
}
```

Then the resulting combined class declaration will be equivalent to the following:

```
class Customer
{
    string name;

    public string Name {
        get { return name; }
        set {
            OnNameChanging(value);
            name = value;
            OnNameChanged();
        }

    }

    void OnNameChanging(string newName)
    {
        Console.WriteLine("Changing " + name + " to " + newName);
    }

    void OnNameChanged()
    {
        Console.WriteLine("Changed to " + name);
    }
}
```

Name binding

Although each part of an extensible type must be declared within the same namespace, the parts are typically written within different namespace declarations. Thus, different `using` directives (Using directives) may be present for each part. When interpreting simple names (Type inference) within one part, only the `using` directives of the namespace declaration(s) enclosing that part are considered. This may result in the same identifier having different meanings in different parts:

```
namespace N
{
    using List = System.Collections.ArrayList;

    partial class A
    {
        List x;                // x has type System.Collections.ArrayList
    }
}

namespace N
{
    using List = Widgets.LinkedList;

    partial class A
    {
        List y;                // y has type Widgets.LinkedList
    }
}
```

Class members

The members of a class consist of the members introduced by its *class_member_declaration*s and the members inherited from the direct base class.

```
class_member_declaration
    : constant_declaration
    | field_declaration
    | method_declaration
    | property_declaration
    | event_declaration
    | indexer_declaration
    | operator_declaration
    | constructor_declaration
    | destructor_declaration
    | static_constructor_declaration
    | type_declaration
    ;
```

The members of a class type are divided into the following categories:

- Constants, which represent constant values associated with the class (Constants).
- Fields, which are the variables of the class (Fields).
- Methods, which implement the computations and actions that can be performed by the class (Methods).
- Properties, which define named characteristics and the actions associated with reading and writing those characteristics (Properties).
- Events, which define notifications that can be generated by the class (Events).
- Indexers, which permit instances of the class to be indexed in the same way (syntactically) as arrays (Indexers).
- Operators, which define the expression operators that can be applied to instances of the class (Operators).
- Instance constructors, which implement the actions required to initialize instances of the class (Instance constructors)
- Destructors, which implement the actions to be performed before instances of the class are permanently discarded (Destructors).
- Static constructors, which implement the actions required to initialize the class itself (Static constructors).
- Types, which represent the types that are local to the class (Nested types).

Members that can contain executable code are collectively known as the *function members* of the class type. The function members of a class type are the methods, properties, events, indexers, operators, instance constructors, destructors, and static constructors of that class type.

A *class_declaration* creates a new declaration space (Declarations), and the *class_member_declaration*s immediately contained by the *class_declaration* introduce new members into this declaration space. The following rules apply to *class_member_declaration*s:

- Instance constructors, destructors and static constructors must have the same name as the immediately enclosing class. All other members must have names that differ from the name of the immediately enclosing class.
- The name of a constant, field, property, event, or type must differ from the names of all other members declared in the same class.
- The name of a method must differ from the names of all other non-methods declared in the same class. In addition, the signature (Signatures and overloading) of a method must differ from the signatures of all other methods declared in the same class, and two methods declared in the same class may not have signatures that differ solely by `ref` and `out`.
- The signature of an instance constructor must differ from the signatures of all other instance constructors declared in the same class, and two constructors declared in the same class may not have signatures that differ solely by `ref` and `out`.
- The signature of an indexer must differ from the signatures of all other indexers declared in the same class.
- The signature of an operator must differ from the signatures of all other operators declared in the same

class.

The inherited members of a class type (Inheritance) are not part of the declaration space of a class. Thus, a derived class is allowed to declare a member with the same name or signature as an inherited member (which in effect hides the inherited member).

The instance type

Each class declaration has an associated bound type (Bound and unbound types), the ***instance type***. For a generic class declaration, the instance type is formed by creating a constructed type (Constructed types) from the type declaration, with each of the supplied type arguments being the corresponding type parameter. Since the instance type uses the type parameters, it can only be used where the type parameters are in scope; that is, inside the class declaration. The instance type is the type of `this` for code written inside the class declaration. For non-generic classes, the instance type is simply the declared class. The following shows several class declarations along with their instance types:

```
class A<T>                          // instance type: A<T>
{
    class B {}                      // instance type: A<T>.B
    class C<U> {}                   // instance type: A<T>.C<U>
}

class D {}                          // instance type: D
```

Members of constructed types

The non-inherited members of a constructed type are obtained by substituting, for each *type_parameter* in the member declaration, the corresponding *type_argument* of the constructed type. The substitution process is based on the semantic meaning of type declarations, and is not simply textual substitution.

For example, given the generic class declaration

```
class Gen<T,U>
{
    public T[,] a;
    public void G(int i, T t, Gen<U,T> gt) {...}
    public U Prop { get {...} set {...} }
    public int H(double d) {...}
}
```

the constructed type `Gen<int[],IComparable<string>>` has the following members:

```
public int[,][] a;
public void G(int i, int[] t, Gen<IComparable<string>,int[]> gt) {...}
public IComparable<string> Prop { get {...} set {...} }
public int H(double d) {...}
```

The type of the member `a` in the generic class declaration `Gen` is "two-dimensional array of `T`", so the type of the member `a` in the constructed type above is "two-dimensional array of one-dimensional array of `int`", or `int[,][]`.

Within instance function members, the type of `this` is the instance type (The instance type) of the containing declaration.

All members of a generic class can use type parameters from any enclosing class, either directly or as part of a constructed type. When a particular closed constructed type (Open and closed types) is used at run-time, each use of a type parameter is replaced with the actual type argument supplied to the constructed type. For example:

```
class C<V>
{
    public V f1;
    public C<V> f2 = null;

    public C(V x) {
        this.f1 = x;
        this.f2 = this;
    }
}

class Application
{
    static void Main() {
        C<int> x1 = new C<int>(1);
        Console.WriteLine(x1.f1);        // Prints 1

        C<double> x2 = new C<double>(3.1415);
        Console.WriteLine(x2.f1);        // Prints 3.1415
    }
}
```

Inheritance

A class ***inherits*** the members of its direct base class type. Inheritance means that a class implicitly contains all members of its direct base class type, except for the instance constructors, destructors and static constructors of the base class. Some important aspects of inheritance are:

- Inheritance is transitive. If `C` is derived from `B`, and `B` is derived from `A`, then `C` inherits the members declared in `B` as well as the members declared in `A`.
- A derived class extends its direct base class. A derived class can add new members to those it inherits, but it cannot remove the definition of an inherited member.
- Instance constructors, destructors, and static constructors are not inherited, but all other members are, regardless of their declared accessibility (Member access). However, depending on their declared accessibility, inherited members might not be accessible in a derived class.
- A derived class can ***hide*** (Hiding through inheritance) inherited members by declaring new members with the same name or signature. Note however that hiding an inherited member does not remove that member—it merely makes that member inaccessible directly through the derived class.
- An instance of a class contains a set of all instance fields declared in the class and its base classes, and an implicit conversion (Implicit reference conversions) exists from a derived class type to any of its base class types. Thus, a reference to an instance of some derived class can be treated as a reference to an instance of any of its base classes.
- A class can declare virtual methods, properties, and indexers, and derived classes can override the implementation of these function members. This enables classes to exhibit polymorphic behavior wherein the actions performed by a function member invocation varies depending on the run-time type of the instance through which that function member is invoked.

The inherited member of a constructed class type are the members of the immediate base class type (Base classes), which is found by substituting the type arguments of the constructed type for each occurrence of the corresponding type parameters in the *class_base* specification. These members, in turn, are transformed by substituting, for each *type_parameter* in the member declaration, the corresponding *type_argument* of the *class_base* specification.

```
class B<U>
{
    public U F(long index) {...}
}

class D<T>: B<T[]>
{
    public T G(string s) {...}
}
```

In the above example, the constructed type `D<int>` has a non-inherited member `public int G(string s)` obtained by substituting the type argument `int` for the type parameter `T`. `D<int>` also has an inherited member from the class declaration `B`. This inherited member is determined by first determining the base class type `B<int[]>` of `D<int>` by substituting `int` for `T` in the base class specification `B<T[]>`. Then, as a type argument to `B`, `int[]` is substituted for `U` in `public U F(long index)`, yielding the inherited member `public int[] F(long index)`.

The new modifier

A *class_member_declaration* is permitted to declare a member with the same name or signature as an inherited member. When this occurs, the derived class member is said to ***hide*** the base class member. Hiding an inherited member is not considered an error, but it does cause the compiler to issue a warning. To suppress the warning, the declaration of the derived class member can include a `new` modifier to indicate that the derived member is intended to hide the base member. This topic is discussed further in Hiding through inheritance.

If a `new` modifier is included in a declaration that doesn't hide an inherited member, a warning to that effect is issued. This warning is suppressed by removing the `new` modifier.

Access modifiers

A *class_member_declaration* can have any one of the five possible kinds of declared accessibility (Declared accessibility): `public`, `protected internal`, `protected`, `internal`, or `private`. Except for the `protected internal` combination, it is a compile-time error to specify more than one access modifier. When a *class_member_declaration* does not include any access modifiers, `private` is assumed.

Constituent types

Types that are used in the declaration of a member are called the constituent types of that member. Possible constituent types are the type of a constant, field, property, event, or indexer, the return type of a method or operator, and the parameter types of a method, indexer, operator, or instance constructor. The constituent types of a member must be at least as accessible as that member itself (Accessibility constraints).

Static and instance members

Members of a class are either ***static members*** or ***instance members***. Generally speaking, it is useful to think of static members as belonging to class types and instance members as belonging to objects (instances of class types).

When a field, method, property, event, operator, or constructor declaration includes a `static` modifier, it declares a static member. In addition, a constant or type declaration implicitly declares a static member. Static members have the following characteristics:

- When a static member `M` is referenced in a *member_access* (Member access) of the form `E.M`, `E` must denote a type containing `M`. It is a compile-time error for `E` to denote an instance.
- A static field identifies exactly one storage location to be shared by all instances of a given closed class type. No matter how many instances of a given closed class type are created, there is only ever one copy of a static field.
- A static function member (method, property, event, operator, or constructor) does not operate on a specific

instance, and it is a compile-time error to refer to `this` in such a function member.

When a field, method, property, event, indexer, constructor, or destructor declaration does not include a `static` modifier, it declares an instance member. (An instance member is sometimes called a non-static member.) Instance members have the following characteristics:

- When an instance member `M` is referenced in a *member_access* (Member access) of the form `E.M`, `E` must denote an instance of a type containing `M`. It is a binding-time error for `E` to denote a type.
- Every instance of a class contains a separate set of all instance fields of the class.
- An instance function member (method, property, indexer, instance constructor, or destructor) operates on a given instance of the class, and this instance can be accessed as `this` (This access).

The following example illustrates the rules for accessing static and instance members:

```
class Test
{
    int x;
    static int y;

    void F() {
        x = 1;            // Ok, same as this.x = 1
        y = 1;            // Ok, same as Test.y = 1
    }

    static void G() {
        x = 1;            // Error, cannot access this.x
        y = 1;            // Ok, same as Test.y = 1
    }

    static void Main() {
        Test t = new Test();
        t.x = 1;          // Ok
        t.y = 1;          // Error, cannot access static member through instance
        Test.x = 1;       // Error, cannot access instance member through type
        Test.y = 1;       // Ok
    }
}
```

The `F` method shows that in an instance function member, a *simple_name* (Simple names) can be used to access both instance members and static members. The `G` method shows that in a static function member, it is a compile-time error to access an instance member through a *simple_name*. The `Main` method shows that in a *member_access* (Member access), instance members must be accessed through instances, and static members must be accessed through types.

Nested types

A type declared within a class or struct declaration is called a ***nested type***. A type that is declared within a compilation unit or namespace is called a ***non-nested type***.

In the example

```
using System;

class A
{
    class B
    {
        static void F() {
            Console.WriteLine("A.B.F");
        }
    }
}
```

class `B` is a nested type because it is declared within class `A`, and class `A` is a non-nested type because it is declared within a compilation unit.

Fully qualified name

The fully qualified name (Fully qualified names) for a nested type is `S.N` where `S` is the fully qualified name of the type in which type `N` is declared.

Declared accessibility

Non-nested types can have `public` or `internal` declared accessibility and have `internal` declared accessibility by default. Nested types can have these forms of declared accessibility too, plus one or more additional forms of declared accessibility, depending on whether the containing type is a class or struct:

- A nested type that is declared in a class can have any of five forms of declared accessibility (`public`, `protected internal`, `protected`, `internal`, or `private`) and, like other class members, defaults to `private` declared accessibility.
- A nested type that is declared in a struct can have any of three forms of declared accessibility (`public`, `internal`, or `private`) and, like other struct members, defaults to `private` declared accessibility.

The example

```
public class List
{
    // Private data structure
    private class Node
    {
        public object Data;
        public Node Next;

        public Node(object data, Node next) {
            this.Data = data;
            this.Next = next;
        }
    }

    private Node first = null;
    private Node last = null;

    // Public interface
    public void AddToFront(object o) {...}
    public void AddToBack(object o) {...}
    public object RemoveFromFront() {...}
    public object RemoveFromBack() {...}
    public int Count { get {...} }
}
```

declares a private nested class `Node`.

Hiding

A nested type may hide (Name hiding) a base member. The `new` modifier is permitted on nested type declarations so that hiding can be expressed explicitly. The example

```
using System;

class Base
{
    public static void M() {
        Console.WriteLine("Base.M");
    }
}

class Derived: Base
{
    new public class M
    {
        public static void F() {
            Console.WriteLine("Derived.M.F");
        }
    }
}

class Test
{
    static void Main() {
        Derived.M.F();
    }
}
```

shows a nested class `M` that hides the method `M` defined in `Base`.

this access

A nested type and its containing type do not have a special relationship with regard to *this_access* (This access). Specifically, `this` within a nested type cannot be used to refer to instance members of the containing type. In cases where a nested type needs access to the instance members of its containing type, access can be provided by providing the `this` for the instance of the containing type as a constructor argument for the nested type. The following example

```
using System;

class C
{
    int i = 123;

    public void F() {
        Nested n = new Nested(this);
        n.G();
    }

    public class Nested
    {
        C this_c;

        public Nested(C c) {
            this_c = c;
        }

        public void G() {
            Console.WriteLine(this_c.i);
        }
    }
}

class Test
{
    static void Main() {
        C c = new C();
        c.F();
    }
}
```

shows this technique. An instance of `C` creates an instance of `Nested` and passes its own `this` to `Nested`'s constructor in order to provide subsequent access to `C`'s instance members.

Access to private and protected members of the containing type

A nested type has access to all of the members that are accessible to its containing type, including members of the containing type that have `private` and `protected` declared accessibility. The example

```
using System;

class C
{
    private static void F() {
        Console.WriteLine("C.F");
    }

    public class Nested
    {
        public static void G() {
            F();
        }
    }
}

class Test
{
    static void Main() {
        C.Nested.G();
    }
}
```

shows a class `C` that contains a nested class `Nested`. Within `Nested`, the method `G` calls the static method `F` defined in `C`, and `F` has private declared accessibility.

A nested type also may access protected members defined in a base type of its containing type. In the example

```
using System;

class Base
{
    protected void F() {
        Console.WriteLine("Base.F");
    }
}

class Derived: Base
{
    public class Nested
    {
        public void G() {
            Derived d = new Derived();
            d.F();        // ok
        }
    }
}

class Test
{
    static void Main() {
        Derived.Nested n = new Derived.Nested();
        n.G();
    }
}
```

the nested class `Derived.Nested` accesses the protected method `F` defined in `Derived`'s base class, `Base`, by calling through an instance of `Derived`.

Nested types in generic classes

A generic class declaration can contain nested type declarations. The type parameters of the enclosing class can be used within the nested types. A nested type declaration can contain additional type parameters that apply only to the nested type.

Every type declaration contained within a generic class declaration is implicitly a generic type declaration. When writing a reference to a type nested within a generic type, the containing constructed type, including its type arguments, must be named. However, from within the outer class, the nested type can be used without qualification; the instance type of the outer class can be implicitly used when constructing the nested type. The following example shows three different correct ways to refer to a constructed type created from `Inner`; the first two are equivalent:

```
class Outer<T>
{
    class Inner<U>
    {
        public static void F(T t, U u) {...}
    }

    static void F(T t) {
        Outer<T>.Inner<string>.F(t, "abc");      // These two statements have
        Inner<string>.F(t, "abc");               // the same effect

        Outer<int>.Inner<string>.F(3, "abc");    // This type is different

        Outer.Inner<string>.F(t, "abc");         // Error, Outer needs type arg
    }
}
```

Although it is bad programming style, a type parameter in a nested type can hide a member or type parameter declared in the outer type:

```
class Outer<T>
{
    class Inner<T>        // Valid, hides Outer's T
    {
        public T t;       // Refers to Inner's T
    }
}
```

Reserved member names

To facilitate the underlying C# run-time implementation, for each source member declaration that is a property, event, or indexer, the implementation must reserve two method signatures based on the kind of the member declaration, its name, and its type. It is a compile-time error for a program to declare a member whose signature matches one of these reserved signatures, even if the underlying run-time implementation does not make use of these reservations.

The reserved names do not introduce declarations, thus they do not participate in member lookup. However, a declaration's associated reserved method signatures do participate in inheritance (Inheritance), and can be hidden with the `new` modifier (The new modifier).

The reservation of these names serves three purposes:

- To allow the underlying implementation to use an ordinary identifier as a method name for get or set access to the C# language feature.
- To allow other languages to interoperate using an ordinary identifier as a method name for get or set access to the C# language feature.
- To help ensure that the source accepted by one conforming compiler is accepted by another, by making the specifics of reserved member names consistent across all C# implementations.

The declaration of a destructor (Destructors) also causes a signature to be reserved (Member names reserved for destructors).

Member names reserved for properties

For a property `P` (Properties) of type `T`, the following signatures are reserved:

```
T get_P();
void set_P(T value);
```

Both signatures are reserved, even if the property is read-only or write-only.

In the example

```
using System;

class A
{
    public int P {
        get { return 123; }
    }
}

class B: A
{
    new public int get_P() {
        return 456;
    }

    new public void set_P(int value) {
    }
}

class Test
{
    static void Main() {
        B b = new B();
        A a = b;
        Console.WriteLine(a.P);
        Console.WriteLine(b.P);
        Console.WriteLine(b.get_P());
    }
}
```

a class `A` defines a read-only property `P`, thus reserving signatures for `get_P` and `set_P` methods. A class `B` derives from `A` and hides both of these reserved signatures. The example produces the output:

```
123
123
456
```

Member names reserved for events

For an event `E` (Events) of delegate type `T`, the following signatures are reserved:

```
void add_E(T handler);
void remove_E(T handler);
```

Member names reserved for indexers

For an indexer (Indexers) of type `T` with parameter-list `L`, the following signatures are reserved:

```
T get_Item(L);
void set_Item(L, T value);
```

Both signatures are reserved, even if the indexer is read-only or write-only.

Furthermore the member name `Item` is reserved.

Member names reserved for destructors

For a class containing a destructor (Destructors), the following signature is reserved:

```
void Finalize();
```

Constants

A ***constant*** is a class member that represents a constant value: a value that can be computed at compile-time. A *constant_declaration* introduces one or more constants of a given type.

```
constant_declaration
    : attributes? constant_modifier* 'const' type constant_declarators ';'
    ;

constant_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    ;

constant_declarators
    : constant_declarator (',' constant_declarator)*
    ;

constant_declarator
    : identifier '=' constant_expression
    ;
```

A *constant_declaration* may include a set of *attributes* (Attributes), a `new` modifier (The new modifier), and a valid combination of the four access modifiers (Access modifiers). The attributes and modifiers apply to all of the members declared by the *constant_declaration*. Even though constants are considered static members, a *constant_declaration* neither requires nor allows a `static` modifier. It is an error for the same modifier to appear multiple times in a constant declaration.

The *type* of a *constant_declaration* specifies the type of the members introduced by the declaration. The type is followed by a list of *constant_declarator*s, each of which introduces a new member. A *constant_declarator* consists of an *identifier* that names the member, followed by an "`=`" token, followed by a *constant_expression* (Constant expressions) that gives the value of the member.

The *type* specified in a constant declaration must be `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, `string`, an *enum_type*, or a *reference_type*. Each *constant_expression* must yield a value of the target type or of a type that can be converted to the target type by an implicit conversion (Implicit conversions).

The *type* of a constant must be at least as accessible as the constant itself (Accessibility constraints).

The value of a constant is obtained in an expression using a *simple_name* (Simple names) or a *member_access* (Member access).

A constant can itself participate in a *constant_expression*. Thus, a constant may be used in any construct that requires a *constant_expression*. Examples of such constructs include `case` labels, `goto case` statements, `enum` member declarations, attributes, and other constant declarations.

As described in Constant expressions, a *constant_expression* is an expression that can be fully evaluated at compile-time. Since the only way to create a non-null value of a *reference_type* other than `string` is to apply the `new` operator, and since the `new` operator is not permitted in a *constant_expression*, the only possible value for constants of *reference_type*s other than `string` is `null`.

When a symbolic name for a constant value is desired, but when the type of that value is not permitted in a constant declaration, or when the value cannot be computed at compile-time by a *constant_expression*, a `readonly` field (Readonly fields) may be used instead.

A constant declaration that declares multiple constants is equivalent to multiple declarations of single constants with the same attributes, modifiers, and type. For example

```
class A
{
    public const double X = 1.0, Y = 2.0, Z = 3.0;
}
```

is equivalent to

```
class A
{
    public const double X = 1.0;
    public const double Y = 2.0;
    public const double Z = 3.0;
}
```

Constants are permitted to depend on other constants within the same program as long as the dependencies are not of a circular nature. The compiler automatically arranges to evaluate the constant declarations in the appropriate order. In the example

```
class A
{
    public const int X = B.Z + 1;
    public const int Y = 10;
}

class B
{
    public const int Z = A.Y + 1;
}
```

the compiler first evaluates `A.Y`, then evaluates `B.Z`, and finally evaluates `A.X`, producing the values `10`, `11`, and `12`. Constant declarations may depend on constants from other programs, but such dependencies are only possible in one direction. Referring to the example above, if `A` and `B` were declared in separate programs, it would be possible for `A.X` to depend on `B.Z`, but `B.Z` could then not simultaneously depend on `A.Y`.

Fields

A ***field*** is a member that represents a variable associated with an object or class. A *field_declaration* introduces one or more fields of a given type.

```
field_declaration
    : attributes? field_modifier* type variable_declarators ';'
    ;

field_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'static'
    | 'readonly'
    | 'volatile'
    | field_modifier_unsafe
    ;

variable_declarators
    : variable_declarator (',' variable_declarator)*
    ;

variable_declarator
    : identifier ('=' variable_initializer)?
    ;

variable_initializer
    : expression
    | array_initializer
    ;
```

A *field_declaration* may include a set of *attributes* (Attributes), a `new` modifier (The new modifier), a valid combination of the four access modifiers (Access modifiers), and a `static` modifier (Static and instance fields). In addition, a *field_declaration* may include a `readonly` modifier (Readonly fields) or a `volatile` modifier (Volatile fields) but not both. The attributes and modifiers apply to all of the members declared by the *field_declaration*. It is an error for the same modifier to appear multiple times in a field declaration.

The *type* of a *field_declaration* specifies the type of the members introduced by the declaration. The type is followed by a list of *variable_declarator*s, each of which introduces a new member. A *variable_declarator* consists of an *identifier* that names that member, optionally followed by an "`=`" token and a *variable_initializer* (Variable initializers) that gives the initial value of that member.

The *type* of a field must be at least as accessible as the field itself (Accessibility constraints).

The value of a field is obtained in an expression using a *simple_name* (Simple names) or a *member_access* (Member access). The value of a non-readonly field is modified using an *assignment* (Assignment operators). The value of a non-readonly field can be both obtained and modified using postfix increment and decrement operators (Postfix increment and decrement operators) and prefix increment and decrement operators (Prefix increment and decrement operators).

A field declaration that declares multiple fields is equivalent to multiple declarations of single fields with the same attributes, modifiers, and type. For example

```
class A
{
    public static int X = 1, Y, Z = 100;
}
```

is equivalent to

```
class A
{
    public static int X = 1;
    public static int Y;
    public static int Z = 100;
}
```

Static and instance fields

When a field declaration includes a `static` modifier, the fields introduced by the declaration are ***static fields***. When no `static` modifier is present, the fields introduced by the declaration are ***instance fields***. Static fields and instance fields are two of the several kinds of variables (Variables) supported by C#, and at times they are referred to as ***static variables*** and ***instance variables***, respectively.

A static field is not part of a specific instance; instead, it is shared amongst all instances of a closed type (Open and closed types). No matter how many instances of a closed class type are created, there is only ever one copy of a static field for the associated application domain.

For example:

```
class C<V>
{
    static int count = 0;

    public C() {
        count++;
    }

    public static int Count {
        get { return count; }
    }
}

class Application
{
    static void Main() {
        C<int> x1 = new C<int>();
        Console.WriteLine(C<int>.Count);        // Prints 1

        C<double> x2 = new C<double>();
        Console.WriteLine(C<int>.Count);        // Prints 1

        C<int> x3 = new C<int>();
        Console.WriteLine(C<int>.Count);        // Prints 2
    }
}
```

An instance field belongs to an instance. Specifically, every instance of a class contains a separate set of all the instance fields of that class.

When a field is referenced in a *member_access* (Member access) of the form `E.M`, if `M` is a static field, `E` must denote a type containing `M`, and if `M` is an instance field, E must denote an instance of a type containing `M`.

The differences between static and instance members are discussed further in Static and instance members.

Readonly fields

When a *field_declaration* includes a `readonly` modifier, the fields introduced by the declaration are ***readonly fields***. Direct assignments to readonly fields can only occur as part of that declaration or in an instance constructor or static constructor in the same class. (A readonly field can be assigned to multiple times in these contexts.) Specifically, direct assignments to a `readonly` field are permitted only in the following contexts:

- In the *variable_declarator* that introduces the field (by including a *variable_initializer* in the declaration).
- For an instance field, in the instance constructors of the class that contains the field declaration; for a static field, in the static constructor of the class that contains the field declaration. These are also the only contexts in which it is valid to pass a `readonly` field as an `out` or `ref` parameter.

Attempting to assign to a `readonly` field or pass it as an `out` or `ref` parameter in any other context is a compile-time error.

Using static readonly fields for constants

A `static readonly` field is useful when a symbolic name for a constant value is desired, but when the type of the value is not permitted in a `const` declaration, or when the value cannot be computed at compile-time. In the example

```
public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);

    private byte red, green, blue;

    public Color(byte r, byte g, byte b) {
        red = r;
        green = g;
        blue = b;
    }
}
```

the `Black`, `White`, `Red`, `Green`, and `Blue` members cannot be declared as `const` members because their values cannot be computed at compile-time. However, declaring them `static readonly` instead has much the same effect.

Versioning of constants and static readonly fields

Constants and readonly fields have different binary versioning semantics. When an expression references a constant, the value of the constant is obtained at compile-time, but when an expression references a readonly field, the value of the field is not obtained until run-time. Consider an application that consists of two separate programs:

```
using System;

namespace Program1
{
    public class Utils
    {
        public static readonly int X = 1;
    }
}

namespace Program2
{
    class Test
    {
        static void Main() {
            Console.WriteLine(Program1.Utils.X);
        }
    }
}
```

The `Program1` and `Program2` namespaces denote two programs that are compiled separately. Because `Program1.Utils.X` is declared as a static readonly field, the value output by the `Console.WriteLine` statement is not known at compile-time, but rather is obtained at run-time. Thus, if the value of `X` is changed and `Program1` is recompiled, the `Console.WriteLine` statement will output the new value even if `Program2` isn't recompiled. However, had `X` been a constant, the value of `X` would have been obtained at the time `Program2` was compiled, and would remain unaffected by changes in `Program1` until `Program2` is recompiled.

Volatile fields

When a *field_declaration* includes a `volatile` modifier, the fields introduced by that declaration are ***volatile fields***.

For non-volatile fields, optimization techniques that reorder instructions can lead to unexpected and unpredictable results in multi-threaded programs that access fields without synchronization such as that provided by the *lock_statement* (The lock statement). These optimizations can be performed by the compiler, by the run-time system, or by hardware. For volatile fields, such reordering optimizations are restricted:

- A read of a volatile field is called a ***volatile read***. A volatile read has "acquire semantics"; that is, it is guaranteed to occur prior to any references to memory that occur after it in the instruction sequence.
- A write of a volatile field is called a ***volatile write***. A volatile write has "release semantics"; that is, it is guaranteed to happen after any memory references prior to the write instruction in the instruction sequence.

These restrictions ensure that all threads will observe volatile writes performed by any other thread in the order in which they were performed. A conforming implementation is not required to provide a single total ordering of volatile writes as seen from all threads of execution. The type of a volatile field must be one of the following:

- A *reference_type*.
- The type `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `char`, `float`, `bool`, `System.IntPtr`, or `System.UIntPtr`.
- An *enum_type* having an enum base type of `byte`, `sbyte`, `short`, `ushort`, `int`, or `uint`.

The example

```
using System;
using System.Threading;

class Test
{
    public static int result;
    public static volatile bool finished;

    static void Thread2() {
        result = 143;
        finished = true;
    }

    static void Main() {
        finished = false;

        // Run Thread2() in a new thread
        new Thread(new ThreadStart(Thread2)).Start();

        // Wait for Thread2 to signal that it has a result by setting
        // finished to true.
        for (;;) {
            if (finished) {
                Console.WriteLine("result = {0}", result);
                return;
            }
        }
    }
}
```

produces the output:

```
result = 143
```

In this example, the method `Main` starts a new thread that runs the method `Thread2`. This method stores a value into a non-volatile field called `result`, then stores `true` in the volatile field `finished`. The main thread waits for the field `finished` to be set to `true`, then reads the field `result`. Since `finished` has been declared `volatile`, the main thread must read the value `143` from the field `result`. If the field `finished` had not been declared `volatile`, then it would be permissible for the store to `result` to be visible to the main thread after the store to `finished`, and hence for the main thread to read the value `0` from the field `result`. Declaring `finished` as a `volatile` field prevents any such inconsistency.

Field initialization

The initial value of a field, whether it be a static field or an instance field, is the default value (Default values) of the field's type. It is not possible to observe the value of a field before this default initialization has occurred, and a field is thus never "uninitialized". The example

```
using System;

class Test
{
    static bool b;
    int i;

    static void Main() {
        Test t = new Test();
        Console.WriteLine("b = {0}, i = {1}", b, t.i);
    }
}
```

produces the output

```
b = False, i = 0
```

because `b` and `i` are both automatically initialized to default values.

Variable initializers

Field declarations may include *variable_initializer*s. For static fields, variable initializers correspond to assignment statements that are executed during class initialization. For instance fields, variable initializers correspond to assignment statements that are executed when an instance of the class is created.

The example

```
using System;

class Test
{
    static double x = Math.Sqrt(2.0);
    int i = 100;
    string s = "Hello";

    static void Main() {
        Test a = new Test();
        Console.WriteLine("x = {0}, i = {1}, s = {2}", x, a.i, a.s);
    }
}
```

produces the output

```
x = 1.4142135623731, i = 100, s = Hello
```

because an assignment to `x` occurs when static field initializers execute and assignments to `i` and `s` occur when the instance field initializers execute.

The default value initialization described in Field initialization occurs for all fields, including fields that have variable initializers. Thus, when a class is initialized, all static fields in that class are first initialized to their default values, and then the static field initializers are executed in textual order. Likewise, when an instance of a class is created, all instance fields in that instance are first initialized to their default values, and then the instance field initializers are executed in textual order.

It is possible for static fields with variable initializers to be observed in their default value state. However, this is strongly discouraged as a matter of style. The example

```
using System;

class Test
{
    static int a = b + 1;
    static int b = a + 1;

    static void Main() {
        Console.WriteLine("a = {0}, b = {1}", a, b);
    }
}
```

exhibits this behavior. Despite the circular definitions of a and b, the program is valid. It results in the output

```
a = 1, b = 2
```

because the static fields `a` and `b` are initialized to `0` (the default value for `int`) before their initializers are executed. When the initializer for `a` runs, the value of `b` is zero, and so `a` is initialized to `1`. When the initializer for `b` runs, the value of `a` is already `1`, and so `b` is initialized to `2`.

Static field initialization

The static field variable initializers of a class correspond to a sequence of assignments that are executed in the textual order in which they appear in the class declaration. If a static constructor (Static constructors) exists in the class, execution of the static field initializers occurs immediately prior to executing that static constructor. Otherwise, the static field initializers are executed at an implementation-dependent time prior to the first use of a static field of that class. The example

```
using System;

class Test
{
    static void Main() {
        Console.WriteLine("{0} {1}", B.Y, A.X);
    }

    public static int F(string s) {
        Console.WriteLine(s);
        return 1;
    }
}

class A
{
    public static int X = Test.F("Init A");
}

class B
{
    public static int Y = Test.F("Init B");
}
```

might produce either the output:

```
Init A
Init B
1 1
```

or the output:

```
Init B
Init A
1 1
```

because the execution of `X`'s initializer and `Y`'s initializer could occur in either order; they are only constrained to occur before the references to those fields. However, in the example:

```
using System;

class Test
{
    static void Main() {
        Console.WriteLine("{0} {1}", B.Y, A.X);
    }

    public static int F(string s) {
        Console.WriteLine(s);
        return 1;
    }
}

class A
{
    static A() {}

    public static int X = Test.F("Init A");
}

class B
{
    static B() {}

    public static int Y = Test.F("Init B");
}
```

the output must be:

```
Init B
Init A
1 1
```

because the rules for when static constructors execute (as defined in Static constructors) provide that `B`'s static constructor (and hence `B`'s static field initializers) must run before `A`'s static constructor and field initializers.

Instance field initialization

The instance field variable initializers of a class correspond to a sequence of assignments that are executed immediately upon entry to any one of the instance constructors (Constructor initializers) of that class. The variable initializers are executed in the textual order in which they appear in the class declaration. The class instance creation and initialization process is described further in Instance constructors.

A variable initializer for an instance field cannot reference the instance being created. Thus, it is a compile-time error to reference `this` in a variable initializer, as it is a compile-time error for a variable initializer to reference any instance member through a *simple_name*. In the example

```
class A
{
    int x = 1;
    int y = x + 1;        // Error, reference to instance member of this
}
```

the variable initializer for `y` results in a compile-time error because it references a member of the instance being created.

Methods

A ***method*** is a member that implements a computation or action that can be performed by an object or class.

Methods are declared using *method_declaration*s:

```
method_declaration
    : method_header method_body
    ;

method_header
    : attributes? method_modifier* 'partial'? return_type member_name type_parameter_list?
      '(' formal_parameter_list? ')' type_parameter_constraints_clause*
    ;

method_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'static'
    | 'virtual'
    | 'sealed'
    | 'override'
    | 'abstract'
    | 'extern'
    | 'async'
    | method_modifier_unsafe
    ;

return_type
    : type
    | 'void'
    ;

member_name
    : identifier
    | interface_type '.' identifier
    ;

method_body
    : block
    | '=>' expression ';'
    | ';'
    ;
```

A *method_declaration* may include a set of *attributes* (Attributes) and a valid combination of the four access modifiers (Access modifiers), the `new` (The new modifier), `static` (Static and instance methods), `virtual` (Virtual methods), `override` (Override methods), `sealed` (Sealed methods), `abstract` (Abstract methods), and `extern` (External methods) modifiers.

A declaration has a valid combination of modifiers if all of the following are true:

- The declaration includes a valid combination of access modifiers (Access modifiers).
- The declaration does not include the same modifier multiple times.
- The declaration includes at most one of the following modifiers: `static`, `virtual`, and `override`.
- The declaration includes at most one of the following modifiers: `new` and `override`.
- If the declaration includes the `abstract` modifier, then the declaration does not include any of the following modifiers: `static`, `virtual`, `sealed` or `extern`.
- If the declaration includes the `private` modifier, then the declaration does not include any of the following modifiers: `virtual`, `override`, or `abstract`.
- If the declaration includes the `sealed` modifier, then the declaration also includes the `override` modifier.
- If the declaration includes the `partial` modifier, then it does not include any of the following modifiers: `new`,

`public`, `protected`, `internal`, `private`, `virtual`, `sealed`, `override`, `abstract`, or `extern`.

A method that has the `async` modifier is an async function and follows the rules described in Async functions.

The *return_type* of a method declaration specifies the type of the value computed and returned by the method. The *return_type* is `void` if the method does not return a value. If the declaration includes the `partial` modifier, then the return type must be `void`.

The *member_name* specifies the name of the method. Unless the method is an explicit interface member implementation (Explicit interface member implementations), the *member_name* is simply an *identifier*. For an explicit interface member implementation, the *member_name* consists of an *interface_type* followed by a "`.`" and an *identifier*.

The optional *type_parameter_list* specifies the type parameters of the method (Type parameters). If a *type_parameter_list* is specified the method is a ***generic method***. If the method has an `extern` modifier, a *type_parameter_list* cannot be specified.

The optional *formal_parameter_list* specifies the parameters of the method (Method parameters).

The optional *type_parameter_constraints_clause*s specify constraints on individual type parameters (Type parameter constraints) and may only be specified if a *type_parameter_list* is also supplied, and the method does not have an `override` modifier.

The *return_type* and each of the types referenced in the *formal_parameter_list* of a method must be at least as accessible as the method itself (Accessibility constraints).

The *method_body* is either a semicolon, a ***statement body*** or an ***expression body***. A statement body consists of a *block*, which specifies the statements to execute when the method is invoked. An expression body consists of `=>` followed by an *expression* and a semicolon, and denotes a single expression to perform when the method is invoked.

For `abstract` and `extern` methods, the *method_body* consists simply of a semicolon. For `partial` methods the *method_body* may consist of either a semicolon, a block body or an expression body. For all other methods, the *method_body* is either a block body or an expression body.

If the *method_body* consists of a semicolon, then the declaration may not include the `async` modifier.

The name, the type parameter list and the formal parameter list of a method define the signature (Signatures and overloading) of the method. Specifically, the signature of a method consists of its name, the number of type parameters and the number, modifiers, and types of its formal parameters. For these purposes, any type parameter of the method that occurs in the type of a formal parameter is identified not by its name, but by its ordinal position in the type argument list of the method.The return type is not part of a method's signature, nor are the names of the type parameters or the formal parameters.

The name of a method must differ from the names of all other non-methods declared in the same class. In addition, the signature of a method must differ from the signatures of all other methods declared in the same class, and two methods declared in the same class may not have signatures that differ solely by `ref` and `out`.

The method's *type_parameter*s are in scope throughout the *method_declaration*, and can be used to form types throughout that scope in *return_type*, *method_body*, and *type_parameter_constraints_clause*s but not in *attributes*.

All formal parameters and type parameters must have different names.

Method parameters

The parameters of a method, if any, are declared by the method's *formal_parameter_list*.

```
formal_parameter_list
    : fixed_parameters
    | fixed_parameters ',' parameter_array
    | parameter_array
    ;

fixed_parameters
    : fixed_parameter (',' fixed_parameter)*
    ;

fixed_parameter
    : attributes? parameter_modifier? type identifier default_argument?
    ;

default_argument
    : '=' expression
    ;

parameter_modifier
    : 'ref'
    | 'out'
    | 'this'
    ;

parameter_array
    : attributes? 'params' array_type identifier
    ;
```

The formal parameter list consists of one or more comma-separated parameters of which only the last may be a *parameter_array*.

A *fixed_parameter* consists of an optional set of *attributes* (Attributes), an optional `ref`, `out` or `this` modifier, a *type*, an *identifier* and an optional *default_argument*. Each *fixed_parameter* declares a parameter of the given type with the given name. The `this` modifier designates the method as an extension method and is only allowed on the first parameter of a static method. Extension methods are further described in Extension methods.

A *fixed_parameter* with a *default_argument* is known as an ***optional parameter***, whereas a *fixed_parameter* without a *default_argument* is a ***required parameter***. A required parameter may not appear after an optional parameter in a *formal_parameter_list*.

A `ref` or `out` parameter cannot have a *default_argument*. The *expression* in a *default_argument* must be one of the following:

- a *constant_expression*
- an expression of the form `new S()` where `S` is a value type
- an expression of the form `default(S)` where `S` is a value type

The *expression* must be implicitly convertible by an identity or nullable conversion to the type of the parameter.

If optional parameters occur in an implementing partial method declaration (Partial methods) , an explicit interface member implementation (Explicit interface member implementations) or in a single-parameter indexer declaration (Indexers) the compiler should give a warning, since these members can never be invoked in a way that permits arguments to be omitted.

A *parameter_array* consists of an optional set of *attributes* (Attributes), a `params` modifier, an *array_type*, and an *identifier*. A parameter array declares a single parameter of the given array type with the given name. The *array_type* of a parameter array must be a single-dimensional array type (Array types). In a method invocation, a parameter array permits either a single argument of the given array type to be specified, or it permits zero or more arguments of the array element type to be specified. Parameter arrays are described further in Parameter

arrays.

A *parameter_array* may occur after an optional parameter, but cannot have a default value -- the omission of arguments for a *parameter_array* would instead result in the creation of an empty array.

The following example illustrates different kinds of parameters:

```
public void M(
    ref int      i,
    decimal      d,
    bool         b = false,
    bool?        n = false,
    string       s = "Hello",
    object       o = null,
    T            t = default(T),
    params int[] a
) { }
```

In the *formal_parameter_list* for `M`, `i` is a required ref parameter, `d` is a required value parameter, `b`, `s`, `o` and `t` are optional value parameters and `a` is a parameter array.

A method declaration creates a separate declaration space for parameters, type parameters and local variables. Names are introduced into this declaration space by the type parameter list and the formal parameter list of the method and by local variable declarations in the *block* of the method. It is an error for two members of a method declaration space to have the same name. It is an error for the method declaration space and the local variable declaration space of a nested declaration space to contain elements with the same name.

A method invocation (Method invocations) creates a copy, specific to that invocation, of the formal parameters and local variables of the method, and the argument list of the invocation assigns values or variable references to the newly created formal parameters. Within the *block* of a method, formal parameters can be referenced by their identifiers in *simple_name* expressions (Simple names).

There are four kinds of formal parameters:

- Value parameters, which are declared without any modifiers.
- Reference parameters, which are declared with the `ref` modifier.
- Output parameters, which are declared with the `out` modifier.
- Parameter arrays, which are declared with the `params` modifier.

As described in Signatures and overloading, the `ref` and `out` modifiers are part of a method's signature, but the `params` modifier is not.

Value parameters

A parameter declared with no modifiers is a value parameter. A value parameter corresponds to a local variable that gets its initial value from the corresponding argument supplied in the method invocation.

When a formal parameter is a value parameter, the corresponding argument in a method invocation must be an expression that is implicitly convertible (Implicit conversions) to the formal parameter type.

A method is permitted to assign new values to a value parameter. Such assignments only affect the local storage location represented by the value parameter—they have no effect on the actual argument given in the method invocation.

Reference parameters

A parameter declared with a `ref` modifier is a reference parameter. Unlike a value parameter, a reference parameter does not create a new storage location. Instead, a reference parameter represents the same storage location as the variable given as the argument in the method invocation.

When a formal parameter is a reference parameter, the corresponding argument in a method invocation must consist of the keyword `ref` followed by a *variable_reference* (Precise rules for determining definite assignment) of the same type as the formal parameter. A variable must be definitely assigned before it can be passed as a reference parameter.

Within a method, a reference parameter is always considered definitely assigned.

A method declared as an iterator (Iterators) cannot have reference parameters.

The example

```
using System;

class Test
{
    static void Swap(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
    }

    static void Main() {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine("i = {0}, j = {1}", i, j);
    }
}
```

produces the output

```
i = 2, j = 1
```

For the invocation of `Swap` in `Main`, `x` represents `i` and `y` represents `j`. Thus, the invocation has the effect of swapping the values of `i` and `j`.

In a method that takes reference parameters it is possible for multiple names to represent the same storage location. In the example

```
class A
{
    string s;

    void F(ref string a, ref string b) {
        s = "One";
        a = "Two";
        b = "Three";
    }

    void G() {
        F(ref s, ref s);
    }
}
```

the invocation of `F` in `G` passes a reference to `s` for both `a` and `b`. Thus, for that invocation, the names `s`, `a`, and `b` all refer to the same storage location, and the three assignments all modify the instance field `s`.

Output parameters

A parameter declared with an `out` modifier is an output parameter. Similar to a reference parameter, an output parameter does not create a new storage location. Instead, an output parameter represents the same storage location as the variable given as the argument in the method invocation.

When a formal parameter is an output parameter, the corresponding argument in a method invocation must consist of the keyword `out` followed by a *variable_reference* (Precise rules for determining definite assignment) of the same type as the formal parameter. A variable need not be definitely assigned before it can be passed as an output parameter, but following an invocation where a variable was passed as an output parameter, the variable is considered definitely assigned.

Within a method, just like a local variable, an output parameter is initially considered unassigned and must be definitely assigned before its value is used.

Every output parameter of a method must be definitely assigned before the method returns.

A method declared as a partial method (Partial methods) or an iterator (Iterators) cannot have output parameters.

Output parameters are typically used in methods that produce multiple return values. For example:

```
using System;

class Test
{
    static void SplitPath(string path, out string dir, out string name) {
        int i = path.Length;
        while (i > 0) {
            char ch = path[i - 1];
            if (ch == '\\' || ch == '/' || ch == ':') break;
            i--;
        }
        dir = path.Substring(0, i);
        name = path.Substring(i);
    }

    static void Main() {
        string dir, name;
        SplitPath("c:\\Windows\\System\\hello.txt", out dir, out name);
        Console.WriteLine(dir);
        Console.WriteLine(name);
    }
}
```

The example produces the output:

```
c:\Windows\System\
hello.txt
```

Note that the `dir` and `name` variables can be unassigned before they are passed to `SplitPath`, and that they are considered definitely assigned following the call.

Parameter arrays

A parameter declared with a `params` modifier is a parameter array. If a formal parameter list includes a parameter array, it must be the last parameter in the list and it must be of a single-dimensional array type. For example, the types `string[]` and `string[][]` can be used as the type of a parameter array, but the type `string[,]` can not. It is not possible to combine the `params` modifier with the modifiers `ref` and `out`.

A parameter array permits arguments to be specified in one of two ways in a method invocation:

- The argument given for a parameter array can be a single expression that is implicitly convertible (Implicit conversions) to the parameter array type. In this case, the parameter array acts precisely like a value parameter.
- Alternatively, the invocation can specify zero or more arguments for the parameter array, where each

argument is an expression that is implicitly convertible (Implicit conversions) to the element type of the parameter array. In this case, the invocation creates an instance of the parameter array type with a length corresponding to the number of arguments, initializes the elements of the array instance with the given argument values, and uses the newly created array instance as the actual argument.

Except for allowing a variable number of arguments in an invocation, a parameter array is precisely equivalent to a value parameter (Value parameters) of the same type.

The example

```
using System;

class Test
{
    static void F(params int[] args) {
        Console.Write("Array contains {0} elements:", args.Length);
        foreach (int i in args)
            Console.Write(" {0}", i);
        Console.WriteLine();
    }

    static void Main() {
        int[] arr = {1, 2, 3};
        F(arr);
        F(10, 20, 30, 40);
        F();
    }
}
```

produces the output

```
Array contains 3 elements: 1 2 3
Array contains 4 elements: 10 20 30 40
Array contains 0 elements:
```

The first invocation of `F` simply passes the array `a` as a value parameter. The second invocation of `F` automatically creates a four-element `int[]` with the given element values and passes that array instance as a value parameter. Likewise, the third invocation of `F` creates a zero-element `int[]` and passes that instance as a value parameter. The second and third invocations are precisely equivalent to writing:

```
F(new int[] {10, 20, 30, 40});
F(new int[] {});
```

When performing overload resolution, a method with a parameter array may be applicable either in its normal form or in its expanded form (Applicable function member). The expanded form of a method is available only if the normal form of the method is not applicable and only if an applicable method with the same signature as the expanded form is not already declared in the same type.

The example

```
using System;

class Test
{
    static void F(params object[] a) {
        Console.WriteLine("F(object[])");
    }

    static void F() {
        Console.WriteLine("F()");
    }

    static void F(object a0, object a1) {
        Console.WriteLine("F(object,object)");
    }

    static void Main() {
        F();
        F(1);
        F(1, 2);
        F(1, 2, 3);
        F(1, 2, 3, 4);
    }
}
```

produces the output

```
F();
F(object[]);
F(object,object);
F(object[]);
F(object[]);
```

In the example, two of the possible expanded forms of the method with a parameter array are already included in the class as regular methods. These expanded forms are therefore not considered when performing overload resolution, and the first and third method invocations thus select the regular methods. When a class declares a method with a parameter array, it is not uncommon to also include some of the expanded forms as regular methods. By doing so it is possible to avoid the allocation of an array instance that occurs when an expanded form of a method with a parameter array is invoked.

When the type of a parameter array is `object[]`, a potential ambiguity arises between the normal form of the method and the expended form for a single `object` parameter. The reason for the ambiguity is that an `object[]` is itself implicitly convertible to type `object`. The ambiguity presents no problem, however, since it can be resolved by inserting a cast if needed.

The example

```
using System;

class Test
{
    static void F(params object[] args) {
        foreach (object o in args) {
            Console.Write(o.GetType().FullName);
            Console.Write(" ");
        }
        Console.WriteLine();
    }

    static void Main() {
        object[] a = {1, "Hello", 123.456};
        object o = a;
        F(a);
        F((object)a);
        F(o);
        F((object[])o);
    }
}
```

produces the output

```
System.Int32 System.String System.Double
System.Object[]
System.Object[]
System.Int32 System.String System.Double
```

In the first and last invocations of `F`, the normal form of `F` is applicable because an implicit conversion exists from the argument type to the parameter type (both are of type `object[]`). Thus, overload resolution selects the normal form of `F`, and the argument is passed as a regular value parameter. In the second and third invocations, the normal form of `F` is not applicable because no implicit conversion exists from the argument type to the parameter type (type `object` cannot be implicitly converted to type `object[]`). However, the expanded form of `F` is applicable, so it is selected by overload resolution. As a result, a one-element `object[]` is created by the invocation, and the single element of the array is initialized with the given argument value (which itself is a reference to an `object[]`).

Static and instance methods

When a method declaration includes a `static` modifier, that method is said to be a static method. When no `static` modifier is present, the method is said to be an instance method.

A static method does not operate on a specific instance, and it is a compile-time error to refer to `this` in a static method.

An instance method operates on a given instance of a class, and that instance can be accessed as `this` (This access).

When a method is referenced in a *member_access* (Member access) of the form `E.M`, if `M` is a static method, `E` must denote a type containing `M`, and if `M` is an instance method, `E` must denote an instance of a type containing `M`.

The differences between static and instance members are discussed further in Static and instance members.

Virtual methods

When an instance method declaration includes a `virtual` modifier, that method is said to be a virtual method. When no `virtual` modifier is present, the method is said to be a non-virtual method.

The implementation of a non-virtual method is invariant: The implementation is the same whether the method is invoked on an instance of the class in which it is declared or an instance of a derived class. In contrast, the implementation of a virtual method can be superseded by derived classes. The process of superseding the implementation of an inherited virtual method is known as ***overriding*** that method (Override methods).

In a virtual method invocation, the ***run-time type*** of the instance for which that invocation takes place determines the actual method implementation to invoke. In a non-virtual method invocation, the ***compile-time type*** of the instance is the determining factor. In precise terms, when a method named `N` is invoked with an argument list `A` on an instance with a compile-time type `C` and a run-time type `R` (where `R` is either `C` or a class derived from `C`), the invocation is processed as follows:

- First, overload resolution is applied to `C`, `N`, and `A`, to select a specific method `M` from the set of methods declared in and inherited by `C`. This is described in Method invocations.
- Then, if `M` is a non-virtual method, `M` is invoked.
- Otherwise, `M` is a virtual method, and the most derived implementation of `M` with respect to `R` is invoked.

For every virtual method declared in or inherited by a class, there exists a ***most derived implementation*** of the method with respect to that class. The most derived implementation of a virtual method `M` with respect to a class `R` is determined as follows:

- If `R` contains the introducing `virtual` declaration of `M`, then this is the most derived implementation of `M`.
- Otherwise, if `R` contains an `override` of `M`, then this is the most derived implementation of `M`.
- Otherwise, the most derived implementation of `M` with respect to `R` is the same as the most derived implementation of `M` with respect to the direct base class of `R`.

The following example illustrates the differences between virtual and non-virtual methods:

```
using System;

class A
{
    public void F() { Console.WriteLine("A.F"); }

    public virtual void G() { Console.WriteLine("A.G"); }
}

class B: A
{
    new public void F() { Console.WriteLine("B.F"); }

    public override void G() { Console.WriteLine("B.G"); }
}

class Test
{
    static void Main() {
        B b = new B();
        A a = b;
        a.F();
        b.F();
        a.G();
        b.G();
    }
}
```

In the example, `A` introduces a non-virtual method `F` and a virtual method `G`. The class `B` introduces a new non-virtual method `F`, thus hiding the inherited `F`, and also overrides the inherited method `G`. The example produces the output:

```
A.F
B.F
B.G
B.G
```

Notice that the statement `a.G()` invokes `B.G`, not `A.G`. This is because the run-time type of the instance (which is `B`), not the compile-time type of the instance (which is `A`), determines the actual method implementation to invoke.

Because methods are allowed to hide inherited methods, it is possible for a class to contain several virtual methods with the same signature. This does not present an ambiguity problem, since all but the most derived method are hidden. In the example

```
using System;

class A
{
    public virtual void F() { Console.WriteLine("A.F"); }
}

class B: A
{
    public override void F() { Console.WriteLine("B.F"); }
}

class C: B
{
    new public virtual void F() { Console.WriteLine("C.F"); }
}

class D: C
{
    public override void F() { Console.WriteLine("D.F"); }
}

class Test
{
    static void Main() {
        D d = new D();
        A a = d;
        B b = d;
        C c = d;
        a.F();
        b.F();
        c.F();
        d.F();
    }
}
```

the `C` and `D` classes contain two virtual methods with the same signature: The one introduced by `A` and the one introduced by `C`. The method introduced by `C` hides the method inherited from `A`. Thus, the override declaration in `D` overrides the method introduced by `C`, and it is not possible for `D` to override the method introduced by `A`. The example produces the output:

```
B.F
B.F
D.F
D.F
```

Note that it is possible to invoke the hidden virtual method by accessing an instance of `D` through a less

derived type in which the method is not hidden.

Override methods

When an instance method declaration includes an `override` modifier, the method is said to be an ***override method***. An override method overrides an inherited virtual method with the same signature. Whereas a virtual method declaration introduces a new method, an override method declaration specializes an existing inherited virtual method by providing a new implementation of that method.

The method overridden by an `override` declaration is known as the ***overridden base method***. For an override method `M` declared in a class `C`, the overridden base method is determined by examining each base class type of `C`, starting with the direct base class type of `C` and continuing with each successive direct base class type, until in a given base class type at least one accessible method is located which has the same signature as `M` after substitution of type arguments. For the purposes of locating the overridden base method, a method is considered accessible if it is `public`, if it is `protected`, if it is `protected internal`, or if it is `internal` and declared in the same program as `C`.

A compile-time error occurs unless all of the following are true for an override declaration:

- An overridden base method can be located as described above.
- There is exactly one such overridden base method. This restriction has effect only if the base class type is a constructed type where the substitution of type arguments makes the signature of two methods the same.
- The overridden base method is a virtual, abstract, or override method. In other words, the overridden base method cannot be static or non-virtual.
- The overridden base method is not a sealed method.
- The override method and the overridden base method have the same return type.
- The override declaration and the overridden base method have the same declared accessibility. In other words, an override declaration cannot change the accessibility of the virtual method. However, if the overridden base method is protected internal and it is declared in a different assembly than the assembly containing the override method then the override method's declared accessibility must be protected.
- The override declaration does not specify type-parameter-constraints-clauses. Instead the constraints are inherited from the overridden base method. Note that constraints that are type parameters in the overridden method may be replaced by type arguments in the inherited constraint. This can lead to constraints that are not legal when explicitly specified, such as value types or sealed types.

The following example demonstrates how the overriding rules work for generic classes:

```
abstract class C<T>
{
    public virtual T F() {...}
    public virtual C<T> G() {...}
    public virtual void H(C<T> x) {...}
}

class D: C<string>
{
    public override string F() {...}            // Ok
    public override C<string> G() {...}         // Ok
    public override void H(C<T> x) {...}        // Error, should be C<string>
}

class E<T,U>: C<U>
{
    public override U F() {...}                 // Ok
    public override C<U> G() {...}              // Ok
    public override void H(C<T> x) {...}        // Error, should be C<U>
}
```

An override declaration can access the overridden base method using a *base_access* (Base access). In the example

```
class A
{
    int x;

    public virtual void PrintFields() {
        Console.WriteLine("x = {0}", x);
    }
}

class B: A
{
    int y;

    public override void PrintFields() {
        base.PrintFields();
        Console.WriteLine("y = {0}", y);
    }
}
```

the `base.PrintFields()` invocation in `B` invokes the `PrintFields` method declared in `A`. A *base_access* disables the virtual invocation mechanism and simply treats the base method as a non-virtual method. Had the invocation in `B` been written `((A)this).PrintFields()`, it would recursively invoke the `PrintFields` method declared in `B`, not the one declared in `A`, since `PrintFields` is virtual and the run-time type of `((A)this)` is `B`.

Only by including an `override` modifier can a method override another method. In all other cases, a method with the same signature as an inherited method simply hides the inherited method. In the example

```
class A
{
    public virtual void F() {}
}

class B: A
{
    public virtual void F() {}        // Warning, hiding inherited F()
}
```

the `F` method in `B` does not include an `override` modifier and therefore does not override the `F` method in `A`. Rather, the `F` method in `B` hides the method in `A`, and a warning is reported because the declaration does not include a `new` modifier.

In the example

```
class A
{
    public virtual void F() {}
}

class B: A
{
    new private void F() {}        // Hides A.F within body of B
}

class C: B
{
    public override void F() {}    // Ok, overrides A.F
}
```

the `F` method in `B` hides the virtual `F` method inherited from `A`. Since the new `F` in `B` has private access, its scope only includes the class body of `B` and does not extend to `C`. Therefore, the declaration of `F` in `C` is permitted to override the `F` inherited from `A`.

Sealed methods

When an instance method declaration includes a `sealed` modifier, that method is said to be a ***sealed method***. If an instance method declaration includes the `sealed` modifier, it must also include the `override` modifier. Use of the `sealed` modifier prevents a derived class from further overriding the method.

In the example

```
using System;

class A
{
    public virtual void F() {
        Console.WriteLine("A.F");
    }

    public virtual void G() {
        Console.WriteLine("A.G");
    }
}

class B: A
{
    sealed override public void F() {
        Console.WriteLine("B.F");
    }

    override public void G() {
        Console.WriteLine("B.G");
    }
}

class C: B
{
    override public void G() {
        Console.WriteLine("C.G");
    }
}
```

the class `B` provides two override methods: an `F` method that has the `sealed` modifier and a `G` method that does not. `B`'s use of the sealed `modifier` prevents `C` from further overriding `F`.

Abstract methods

When an instance method declaration includes an `abstract` modifier, that method is said to be an ***abstract method***. Although an abstract method is implicitly also a virtual method, it cannot have the modifier `virtual`.

An abstract method declaration introduces a new virtual method but does not provide an implementation of that method. Instead, non-abstract derived classes are required to provide their own implementation by overriding that method. Because an abstract method provides no actual implementation, the *method_body* of an abstract method simply consists of a semicolon.

Abstract method declarations are only permitted in abstract classes (Abstract classes).

In the example

```
public abstract class Shape
{
    public abstract void Paint(Graphics g, Rectangle r);
}

public class Ellipse: Shape
{
    public override void Paint(Graphics g, Rectangle r) {
        g.DrawEllipse(r);
    }
}

public class Box: Shape
{
    public override void Paint(Graphics g, Rectangle r) {
        g.DrawRect(r);
    }
}
```

the `Shape` class defines the abstract notion of a geometrical shape object that can paint itself. The `Paint` method is abstract because there is no meaningful default implementation. The `Ellipse` and `Box` classes are concrete `Shape` implementations. Because these classes are non-abstract, they are required to override the `Paint` method and provide an actual implementation.

It is a compile-time error for a *base_access* (Base access) to reference an abstract method. In the example

```
abstract class A
{
    public abstract void F();
}

class B: A
{
    public override void F() {
        base.F();                       // Error, base.F is abstract
    }
}
```

a compile-time error is reported for the `base.F()` invocation because it references an abstract method.

An abstract method declaration is permitted to override a virtual method. This allows an abstract class to force re-implementation of the method in derived classes, and makes the original implementation of the method unavailable. In the example

```
using System;

class A
{
    public virtual void F() {
        Console.WriteLine("A.F");
    }
}

abstract class B: A
{
    public abstract override void F();
}

class C: B
{
    public override void F() {
        Console.WriteLine("C.F");
    }
}
```

class `A` declares a virtual method, class `B` overrides this method with an abstract method, and class `C` overrides the abstract method to provide its own implementation.

External methods

When a method declaration includes an `extern` modifier, that method is said to be an ***external method***. External methods are implemented externally, typically using a language other than C#. Because an external method declaration provides no actual implementation, the *method_body* of an external method simply consists of a semicolon. An external method may not be generic.

The `extern` modifier is typically used in conjunction with a `DllImport` attribute (Interoperation with COM and Win32 components), allowing external methods to be implemented by DLLs (Dynamic Link Libraries). The execution environment may support other mechanisms whereby implementations of external methods can be provided.

When an external method includes a `DllImport` attribute, the method declaration must also include a `static` modifier. This example demonstrates the use of the `extern` modifier and the `DllImport` attribute:

```
using System.Text;
using System.Security.Permissions;
using System.Runtime.InteropServices;

class Path
{
    [DllImport("kernel32", SetLastError=true)]
    static extern bool CreateDirectory(string name, SecurityAttribute sa);

    [DllImport("kernel32", SetLastError=true)]
    static extern bool RemoveDirectory(string name);

    [DllImport("kernel32", SetLastError=true)]
    static extern int GetCurrentDirectory(int bufSize, StringBuilder buf);

    [DllImport("kernel32", SetLastError=true)]
    static extern bool SetCurrentDirectory(string name);
}
```

Partial methods (recap)

When a method declaration includes a `partial` modifier, that method is said to be a ***partial method***. Partial methods can only be declared as members of partial types (Partial types), and are subject to a number of

restrictions. Partial methods are further described in Partial methods.

Extension methods

When the first parameter of a method includes the `this` modifier, that method is said to be an ***extension method***. Extension methods can only be declared in non-generic, non-nested static classes. The first parameter of an extension method can have no modifiers other than `this`, and the parameter type cannot be a pointer type.

The following is an example of a static class that declares two extension methods:

```
public static class Extensions
{
    public static int ToInt32(this string s) {
        return Int32.Parse(s);
    }

    public static T[] Slice<T>(this T[] source, int index, int count) {
        if (index < 0 || count < 0 || source.Length - index < count)
            throw new ArgumentException();
        T[] result = new T[count];
        Array.Copy(source, index, result, 0, count);
        return result;
    }
}
```

An extension method is a regular static method. In addition, where its enclosing static class is in scope, an extension method can be invoked using instance method invocation syntax (Extension method invocations), using the receiver expression as the first argument.

The following program uses the extension methods declared above:

```
static class Program
{
    static void Main() {
        string[] strings = { "1", "22", "333", "4444" };
        foreach (string s in strings.Slice(1, 2)) {
            Console.WriteLine(s.ToInt32());
        }
    }
}
```

The `Slice` method is available on the `string[]`, and the `ToInt32` method is available on `string`, because they have been declared as extension methods. The meaning of the program is the same as the following, using ordinary static method calls:

```
static class Program
{
    static void Main() {
        string[] strings = { "1", "22", "333", "4444" };
        foreach (string s in Extensions.Slice(strings, 1, 2)) {
            Console.WriteLine(Extensions.ToInt32(s));
        }
    }
}
```

Method body

The *method_body* of a method declaration consists of either a block body, an expression body or a semicolon.

The ***result type*** of a method is `void` if the return type is `void`, or if the method is async and the return type is

`System.Threading.Tasks.Task`. Otherwise, the result type of a non-async method is its return type, and the result type of an async method with return type `System.Threading.Tasks.Task<T>` is `T`.

When a method has a `void` result type and a block body, `return` statements (The return statement) in the block are not permitted to specify an expression. If execution of the block of a void method completes normally (that is, control flows off the end of the method body), that method simply returns to its current caller.

When a method has a `void` result and an expression body, the expression `E` must be a *statement_expression*, and the body is exactly equivalent to a block body of the form `{ E; }`.

When a method has a non-void result type and a block body, each `return` statement in the block must specify an expression that is implicitly convertible to the result type. The endpoint of a block body of a value-returning method must not be reachable. In other words, in a value-returning method with a block body, control is not permitted to flow off the end of the method body.

When a method has a non-void result type and an expression body, the expression must be implicitly convertible to the result type, and the body is exactly equivalent to a block body of the form `{ return E; }`.

In the example

```
class A
{
    public int F() {}           // Error, return value required

    public int G() {
        return 1;
    }

    public int H(bool b) {
        if (b) {
            return 1;
        }
        else {
            return 0;
        }
    }

    public int I(bool b) => b ? 1 : 0;
}
```

the value-returning `F` method results in a compile-time error because control can flow off the end of the method body. The `G` and `H` methods are correct because all possible execution paths end in a return statement that specifies a return value. The `I` method is correct, because its body is equivalent to a statement block with just a single return statement in it.

Method overloading

The method overload resolution rules are described in Type inference.

Properties

A ***property*** is a member that provides access to a characteristic of an object or a class. Examples of properties include the length of a string, the size of a font, the caption of a window, the name of a customer, and so on. Properties are a natural extension of fields—both are named members with associated types, and the syntax for accessing fields and properties is the same. However, unlike fields, properties do not denote storage locations. Instead, properties have ***accessors*** that specify the statements to be executed when their values are read or written. Properties thus provide a mechanism for associating actions with the reading and writing of an object's attributes; furthermore, they permit such attributes to be computed.

Properties are declared using *property_declaration*s:

```
property_declaration
    : attributes? property_modifier* type member_name property_body
    ;

property_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'static'
    | 'virtual'
    | 'sealed'
    | 'override'
    | 'abstract'
    | 'extern'
    | property_modifier_unsafe
    ;

property_body
    : '{' accessor_declarations '}' property_initializer?
    | '=>' expression ';'
    ;

property_initializer
    : '=' variable_initializer ';'
    ;
```

A *property_declaration* may include a set of *attributes* (Attributes) and a valid combination of the four access modifiers (Access modifiers), the `new` (The new modifier), `static` (Static and instance methods), `virtual` (Virtual methods), `override` (Override methods), `sealed` (Sealed methods), `abstract` (Abstract methods), and `extern` (External methods) modifiers.

Property declarations are subject to the same rules as method declarations (Methods) with regard to valid combinations of modifiers.

The *type* of a property declaration specifies the type of the property introduced by the declaration, and the *member_name* specifies the name of the property. Unless the property is an explicit interface member implementation, the *member_name* is simply an *identifier*. For an explicit interface member implementation (Explicit interface member implementations), the *member_name* consists of an *interface_type* followed by a "`.`" and an *identifier*.

The *type* of a property must be at least as accessible as the property itself (Accessibility constraints).

A *property_body* may either consist of an ***accessor body*** or an ***expression body***. In an accessor body, *accessor_declarations*, which must be enclosed in "`{`" and "`}`" tokens, declare the accessors (Accessors) of the property. The accessors specify the executable statements associated with reading and writing the property.

An expression body consisting of `=>` followed by an *expression* `E` and a semicolon is exactly equivalent to the statement body `{ get { return E; } }`, and can therefore only be used to specify getter-only properties where the result of the getter is given by a single expression.

A *property_initializer* may only be given for an automatically implemented property (Automatically implemented properties), and causes the initialization of the underlying field of such properties with the value given by the *expression*.

Even though the syntax for accessing a property is the same as that for a field, a property is not classified as a variable. Thus, it is not possible to pass a property as a `ref` or `out` argument.

When a property declaration includes an `extern` modifier, the property is said to be an ***external property***.

Because an external property declaration provides no actual implementation, each of its *accessor_declarations* consists of a semicolon.

Static and instance properties

When a property declaration includes a `static` modifier, the property is said to be a ***static property***. When no `static` modifier is present, the property is said to be an ***instance property***.

A static property is not associated with a specific instance, and it is a compile-time error to refer to `this` in the accessors of a static property.

An instance property is associated with a given instance of a class, and that instance can be accessed as `this` (This access) in the accessors of that property.

When a property is referenced in a *member_access* (Member access) of the form `E.M`, if `M` is a static property, `E` must denote a type containing `M`, and if `M` is an instance property, E must denote an instance of a type containing `M`.

The differences between static and instance members are discussed further in Static and instance members.

Accessors

The *accessor_declarations* of a property specify the executable statements associated with reading and writing that property.

```
accessor_declarations
    : get_accessor_declaration set_accessor_declaration?
    | set_accessor_declaration get_accessor_declaration?
    ;

get_accessor_declaration
    : attributes? accessor_modifier? 'get' accessor_body
    ;

set_accessor_declaration
    : attributes? accessor_modifier? 'set' accessor_body
    ;

accessor_modifier
    : 'protected'
    | 'internal'
    | 'private'
    | 'protected' 'internal'
    | 'internal' 'protected'
    ;

accessor_body
    : block
    | ';'
    ;
```

The accessor declarations consist of a *get_accessor_declaration*, a *set_accessor_declaration*, or both. Each accessor declaration consists of the token `get` or `set` followed by an optional *accessor_modifier* and an *accessor_body*.

The use of *accessor_modifier*s is governed by the following restrictions:

- An *accessor_modifier* may not be used in an interface or in an explicit interface member implementation.
- For a property or indexer that has no `override` modifier, an *accessor_modifier* is permitted only if the property or indexer has both a `get` and `set` accessor, and then is permitted only on one of those accessors.
- For a property or indexer that includes an `override` modifier, an accessor must match the *accessor_modifier*, if any, of the accessor being overridden.

- The *accessor_modifier* must declare an accessibility that is strictly more restrictive than the declared accessibility of the property or indexer itself. To be precise:
  - If the property or indexer has a declared accessibility of `public`, the *accessor_modifier* may be either `protected internal`, `internal`, `protected`, or `private`.
  - If the property or indexer has a declared accessibility of `protected internal`, the *accessor_modifier* may be either `internal`, `protected`, or `private`.
  - If the property or indexer has a declared accessibility of `internal` or `protected`, the *accessor_modifier* must be `private`.
  - If the property or indexer has a declared accessibility of `private`, no *accessor_modifier* may be used.

For `abstract` and `extern` properties, the *accessor_body* for each accessor specified is simply a semicolon. A non-abstract, non-extern property may have each *accessor_body* be a semicolon, in which case it is an ***automatically implemented property*** (Automatically implemented properties). An automatically implemented property must have at least a get accessor. For the accessors of any other non-abstract, non-extern property, the *accessor_body* is a *block* which specifies the statements to be executed when the corresponding accessor is invoked.

A `get` accessor corresponds to a parameterless method with a return value of the property type. Except as the target of an assignment, when a property is referenced in an expression, the `get` accessor of the property is invoked to compute the value of the property (Values of expressions). The body of a `get` accessor must conform to the rules for value-returning methods described in Method body. In particular, all `return` statements in the body of a `get` accessor must specify an expression that is implicitly convertible to the property type. Furthermore, the endpoint of a `get` accessor must not be reachable.

A `set` accessor corresponds to a method with a single value parameter of the property type and a `void` return type. The implicit parameter of a `set` accessor is always named `value`. When a property is referenced as the target of an assignment (Assignment operators), or as the operand of `++` or `--` (Postfix increment and decrement operators, Prefix increment and decrement operators), the `set` accessor is invoked with an argument (whose value is that of the right-hand side of the assignment or the operand of the `++` or `--` operator) that provides the new value (Simple assignment). The body of a `set` accessor must conform to the rules for `void` methods described in Method body. In particular, `return` statements in the `set` accessor body are not permitted to specify an expression. Since a `set` accessor implicitly has a parameter named `value`, it is a compile-time error for a local variable or constant declaration in a `set` accessor to have that name.

Based on the presence or absence of the `get` and `set` accessors, a property is classified as follows:

- A property that includes both a `get` accessor and a `set` accessor is said to be a ***read-write*** property.
- A property that has only a `get` accessor is said to be a ***read-only*** property. It is a compile-time error for a read-only property to be the target of an assignment.
- A property that has only a `set` accessor is said to be a ***write-only*** property. Except as the target of an assignment, it is a compile-time error to reference a write-only property in an expression.

In the example

```
public class Button: Control
{
    private string caption;

    public string Caption {
        get {
            return caption;
        }
        set {
            if (caption != value) {
                caption = value;
                Repaint();
            }
        }
    }

    public override void Paint(Graphics g, Rectangle r) {
        // Painting code goes here
    }
}
```

the `Button` control declares a public `Caption` property. The `get` accessor of the `Caption` property returns the string stored in the private `caption` field. The `set` accessor checks if the new value is different from the current value, and if so, it stores the new value and repaints the control. Properties often follow the pattern shown above: The `get` accessor simply returns a value stored in a private field, and the `set` accessor modifies that private field and then performs any additional actions required to fully update the state of the object.

Given the `Button` class above, the following is an example of use of the `Caption` property:

```
Button okButton = new Button();
okButton.Caption = "OK";            // Invokes set accessor
string s = okButton.Caption;        // Invokes get accessor
```

Here, the `set` accessor is invoked by assigning a value to the property, and the `get` accessor is invoked by referencing the property in an expression.

The `get` and `set` accessors of a property are not distinct members, and it is not possible to declare the accessors of a property separately. As such, it is not possible for the two accessors of a read-write property to have different accessibility. The example

```
class A
{
    private string name;

    public string Name {                // Error, duplicate member name
        get { return name; }
    }

    public string Name {                // Error, duplicate member name
        set { name = value; }
    }
}
```

does not declare a single read-write property. Rather, it declares two properties with the same name, one read-only and one write-only. Since two members declared in the same class cannot have the same name, the example causes a compile-time error to occur.

When a derived class declares a property by the same name as an inherited property, the derived property hides the inherited property with respect to both reading and writing. In the example

```
class A
{
    public int P {
        set {...}
    }
}

class B: A
{
    new public int P {
        get {...}
    }
}
```

the `P` property in `B` hides the `P` property in `A` with respect to both reading and writing. Thus, in the statements

```
B b = new B();
b.P = 1;          // Error, B.P is read-only
((A)b).P = 1;     // Ok, reference to A.P
```

the assignment to `b.P` causes a compile-time error to be reported, since the read-only `P` property in `B` hides the write-only `P` property in `A`. Note, however, that a cast can be used to access the hidden `P` property.

Unlike public fields, properties provide a separation between an object's internal state and its public interface. Consider the example:

```
class Label
{
    private int x, y;
    private string caption;

    public Label(int x, int y, string caption) {
        this.x = x;
        this.y = y;
        this.caption = caption;
    }

    public int X {
        get { return x; }
    }

    public int Y {
        get { return y; }
    }

    public Point Location {
        get { return new Point(x, y); }
    }

    public string Caption {
        get { return caption; }
    }
}
```

Here, the `Label` class uses two `int` fields, `x` and `y`, to store its location. The location is publicly exposed both as an `X` and a `Y` property and as a `Location` property of type `Point`. If, in a future version of `Label`, it becomes more convenient to store the location as a `Point` internally, the change can be made without affecting the public interface of the class:

```
class Label
{
    private Point location;
    private string caption;

    public Label(int x, int y, string caption) {
        this.location = new Point(x, y);
        this.caption = caption;
    }

    public int X {
        get { return location.x; }
    }

    public int Y {
        get { return location.y; }
    }

    public Point Location {
        get { return location; }
    }

    public string Caption {
        get { return caption; }
    }
}
```

Had `x` and `y` instead been `public readonly` fields, it would have been impossible to make such a change to the `Label` class.

Exposing state through properties is not necessarily any less efficient than exposing fields directly. In particular, when a property is non-virtual and contains only a small amount of code, the execution environment may replace calls to accessors with the actual code of the accessors. This process is known as ***inlining***, and it makes property access as efficient as field access, yet preserves the increased flexibility of properties.

Since invoking a `get` accessor is conceptually equivalent to reading the value of a field, it is considered bad programming style for `get` accessors to have observable side-effects. In the example

```
class Counter
{
    private int next;

    public int Next {
        get { return next++; }
    }
}
```

the value of the `Next` property depends on the number of times the property has previously been accessed. Thus, accessing the property produces an observable side-effect, and the property should be implemented as a method instead.

The "no side-effects" convention for `get` accessors doesn't mean that `get` accessors should always be written to simply return values stored in fields. Indeed, `get` accessors often compute the value of a property by accessing multiple fields or invoking methods. However, a properly designed `get` accessor performs no actions that cause observable changes in the state of the object.

Properties can be used to delay initialization of a resource until the moment it is first referenced. For example:

```
using System.IO;

public class Console
{
    private static TextReader reader;
    private static TextWriter writer;
    private static TextWriter error;

    public static TextReader In {
        get {
            if (reader == null) {
                reader = new StreamReader(Console.OpenStandardInput());
            }
            return reader;
        }
    }

    public static TextWriter Out {
        get {
            if (writer == null) {
                writer = new StreamWriter(Console.OpenStandardOutput());
            }
            return writer;
        }
    }

    public static TextWriter Error {
        get {
            if (error == null) {
                error = new StreamWriter(Console.OpenStandardError());
            }
            return error;
        }
    }
}
```

The `Console` class contains three properties, `In`, `Out`, and `Error`, that represent the standard input, output, and error devices, respectively. By exposing these members as properties, the `Console` class can delay their initialization until they are actually used. For example, upon first referencing the `Out` property, as in

```
Console.Out.WriteLine("hello, world");
```

the underlying `TextWriter` for the output device is created. But if the application makes no reference to the `In` and `Error` properties, then no objects are created for those devices.

Automatically implemented properties

An automatically implemented property (or ***auto-property*** for short), is a non-abstract non-extern property with semicolon-only accessor bodies. Auto-properties must have a get accessor and can optionally have a set accessor.

When a property is specified as an automatically implemented property, a hidden backing field is automatically available for the property, and the accessors are implemented to read from and write to that backing field. If the auto-property has no set accessor, the backing field is considered `readonly` (Readonly fields). Just like a `readonly` field, a getter-only auto-property can also be assigned to in the body of a constructor of the enclosing class. Such an assignment assigns directly to the readonly backing field of the property.

An auto-property may optionally have a *property_initializer*, which is applied directly to the backing field as a *variable_initializer* (Variable initializers).

The following example:

```
public class Point {
    public int X { get; set; } = 0;
    public int Y { get; set; } = 0;
}
```

is equivalent to the following declaration:

```
public class Point {
    private int __x = 0;
    private int __y = 0;
    public int X { get { return __x; } set { __x = value; } }
    public int Y { get { return __y; } set { __y = value; } }
}
```

The following example:

```
public class ReadOnlyPoint
{
 public int X { get; }
 public int Y { get; }
 public ReadOnlyPoint(int x, int y) { X = x; Y = y; }
}
```

is equivalent to the following declaration:

```
public class ReadOnlyPoint
{
 private readonly int __x;
 private readonly int __y;
 public int X { get { return __x; } }
 public int Y { get { return __y; } }
    public ReadOnlyPoint(int x, int y) { __x = x; __y = y; }
}
```

Notice that the assignments to the readonly field are legal, because they occur within the constructor.

Accessibility

If an accessor has an *accessor_modifier*, the accessibility domain (Accessibility domains) of the accessor is determined using the declared accessibility of the *accessor_modifier*. If an accessor does not have an *accessor_modifier*, the accessibility domain of the accessor is determined from the declared accessibility of the property or indexer.

The presence of an *accessor_modifier* never affects member lookup (Operators) or overload resolution (Overload resolution). The modifiers on the property or indexer always determine which property or indexer is bound to, regardless of the context of the access.

Once a particular property or indexer has been selected, the accessibility domains of the specific accessors involved are used to determine if that usage is valid:

- If the usage is as a value (Values of expressions), the `get` accessor must exist and be accessible.
- If the usage is as the target of a simple assignment (Simple assignment), the `set` accessor must exist and be accessible.
- If the usage is as the target of compound assignment (Compound assignment), or as the target of the `++` or `--` operators (Function members.9, Invocation expressions), both the `get` accessors and the `set` accessor must exist and be accessible.

In the following example, the property `A.Text` is hidden by the property `B.Text`, even in contexts where only the `set` accessor is called. In contrast, the property `B.Count` is not accessible to class `M`, so the accessible property `A.Count` is used instead.

```
class A
{
    public string Text {
        get { return "hello"; }
        set { }
    }

    public int Count {
        get { return 5; }
        set { }
    }
}

class B: A
{
    private string text = "goodbye";
    private int count = 0;

    new public string Text {
        get { return text; }
        protected set { text = value; }
    }

    new protected int Count {
        get { return count; }
        set { count = value; }
    }
}

class M
{
    static void Main() {
        B b = new B();
        b.Count = 12;             // Calls A.Count set accessor
        int i = b.Count;          // Calls A.Count get accessor
        b.Text = "howdy";         // Error, B.Text set accessor not accessible
        string s = b.Text;        // Calls B.Text get accessor
    }
}
```

An accessor that is used to implement an interface may not have an *accessor_modifier*. If only one accessor is used to implement an interface, the other accessor may be declared with an *accessor_modifier*.

```
public interface I
{
    string Prop { get; }
}

public class C: I
{
    public string Prop {
        get { return "April"; }       // Must not have a modifier here
        internal set {...}            // Ok, because I.Prop has no set accessor
    }
}
```

Virtual, sealed, override, and abstract property accessors

A `virtual` property declaration specifies that the accessors of the property are virtual. The `virtual` modifier

applies to both accessors of a read-write property—it is not possible for only one accessor of a read-write property to be virtual.

An `abstract` property declaration specifies that the accessors of the property are virtual, but does not provide an actual implementation of the accessors. Instead, non-abstract derived classes are required to provide their own implementation for the accessors by overriding the property. Because an accessor for an abstract property declaration provides no actual implementation, its *accessor_body* simply consists of a semicolon.

A property declaration that includes both the `abstract` and `override` modifiers specifies that the property is abstract and overrides a base property. The accessors of such a property are also abstract.

Abstract property declarations are only permitted in abstract classes (Abstract classes).The accessors of an inherited virtual property can be overridden in a derived class by including a property declaration that specifies an `override` directive. This is known as an ***overriding property declaration***. An overriding property declaration does not declare a new property. Instead, it simply specializes the implementations of the accessors of an existing virtual property.

An overriding property declaration must specify the exact same accessibility modifiers, type, and name as the inherited property. If the inherited property has only a single accessor (i.e., if the inherited property is read-only or write-only), the overriding property must include only that accessor. If the inherited property includes both accessors (i.e., if the inherited property is read-write), the overriding property can include either a single accessor or both accessors.

An overriding property declaration may include the `sealed` modifier. Use of this modifier prevents a derived class from further overriding the property. The accessors of a sealed property are also sealed.

Except for differences in declaration and invocation syntax, virtual, sealed, override, and abstract accessors behave exactly like virtual, sealed, override and abstract methods. Specifically, the rules described in Virtual methods, Override methods, Sealed methods, and Abstract methods apply as if accessors were methods of a corresponding form:

- A `get` accessor corresponds to a parameterless method with a return value of the property type and the same modifiers as the containing property.
- A `set` accessor corresponds to a method with a single value parameter of the property type, a `void` return type, and the same modifiers as the containing property.

In the example

```
abstract class A
{
    int y;

    public virtual int X {
        get { return 0; }
    }

    public virtual int Y {
        get { return y; }
        set { y = value; }
    }

    public abstract int Z { get; set; }
}
```

`X` is a virtual read-only property, `Y` is a virtual read-write property, and `Z` is an abstract read-write property. Because `Z` is abstract, the containing class `A` must also be declared abstract.

A class that derives from `A` is show below:

```
class B: A
{
    int z;

    public override int X {
        get { return base.X + 1; }
    }

    public override int Y {
        set { base.Y = value < 0? 0: value; }
    }

    public override int Z {
        get { return z; }
        set { z = value; }
    }
}
```

Here, the declarations of `X`, `Y`, and `Z` are overriding property declarations. Each property declaration exactly matches the accessibility modifiers, type, and name of the corresponding inherited property. The `get` accessor of `X` and the `set` accessor of `Y` use the `base` keyword to access the inherited accessors. The declaration of `Z` overrides both abstract accessors—thus, there are no outstanding abstract function members in `B`, and `B` is permitted to be a non-abstract class.

When a property is declared as an `override`, any overridden accessors must be accessible to the overriding code. In addition, the declared accessibility of both the property or indexer itself, and of the accessors, must match that of the overridden member and accessors. For example:

```
public class B
{
    public virtual int P {
        protected set {...}
        get {...}
    }
}

public class D: B
{
    public override int P {
        protected set {...}            // Must specify protected here
        get {...}                      // Must not have a modifier here
    }
}
```

Events

An ***event*** is a member that enables an object or class to provide notifications. Clients can attach executable code for events by supplying ***event handlers***.

Events are declared using *event_declaration*s:

```
event_declaration
    : attributes? event_modifier* 'event' type variable_declarators ';'
    | attributes? event_modifier* 'event' type member_name '{' event_accessor_declarations '}'
    ;

event_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'static'
    | 'virtual'
    | 'sealed'
    | 'override'
    | 'abstract'
    | 'extern'
    | event_modifier_unsafe
    ;

event_accessor_declarations
    : add_accessor_declaration remove_accessor_declaration
    | remove_accessor_declaration add_accessor_declaration
    ;

add_accessor_declaration
    : attributes? 'add' block
    ;

remove_accessor_declaration
    : attributes? 'remove' block
    ;
```

An *event_declaration* may include a set of *attributes* (Attributes) and a valid combination of the four access modifiers (Access modifiers), the `new` (The new modifier), `static` (Static and instance methods), `virtual` (Virtual methods), `override` (Override methods), `sealed` (Sealed methods), `abstract` (Abstract methods), and `extern` (External methods) modifiers.

Event declarations are subject to the same rules as method declarations (Methods) with regard to valid combinations of modifiers.

The *type* of an event declaration must be a *delegate_type* (Reference types), and that *delegate_type* must be at least as accessible as the event itself (Accessibility constraints).

An event declaration may include *event_accessor_declarations*. However, if it does not, for non-extern, non-abstract events, the compiler supplies them automatically (Field-like events); for extern events, the accessors are provided externally.

An event declaration that omits *event_accessor_declarations* defines one or more events—one for each of the *variable_declarator*s. The attributes and modifiers apply to all of the members declared by such an *event_declaration*.

It is a compile-time error for an *event_declaration* to include both the `abstract` modifier and brace-delimited *event_accessor_declarations*.

When an event declaration includes an `extern` modifier, the event is said to be an ***external event***. Because an external event declaration provides no actual implementation, it is an error for it to include both the `extern` modifier and *event_accessor_declarations*.

It is a compile-time error for a *variable_declarator* of an event declaration with an `abstract` or `external` modifier to include a *variable_initializer*.

An event can be used as the left-hand operand of the `+=` and `-=` operators (Event assignment). These operators are used, respectively, to attach event handlers to or to remove event handlers from an event, and the access modifiers of the event control the contexts in which such operations are permitted.

Since `+=` and `-=` are the only operations that are permitted on an event outside the type that declares the event, external code can add and remove handlers for an event, but cannot in any other way obtain or modify the underlying list of event handlers.

In an operation of the form `x += y` or `x -= y`, when `x` is an event and the reference takes place outside the type that contains the declaration of `x`, the result of the operation has type `void` (as opposed to having the type of `x`, with the value of `x` after the assignment). This rule prohibits external code from indirectly examining the underlying delegate of an event.

The following example shows how event handlers are attached to instances of the `Button` class:

```
public delegate void EventHandler(object sender, EventArgs e);

public class Button: Control
{
    public event EventHandler Click;
}

public class LoginDialog: Form
{
    Button OkButton;
    Button CancelButton;

    public LoginDialog() {
        OkButton = new Button(...);
        OkButton.Click += new EventHandler(OkButtonClick);
        CancelButton = new Button(...);
        CancelButton.Click += new EventHandler(CancelButtonClick);
    }

    void OkButtonClick(object sender, EventArgs e) {
        // Handle OkButton.Click event
    }

    void CancelButtonClick(object sender, EventArgs e) {
        // Handle CancelButton.Click event
    }
}
```

Here, the `LoginDialog` instance constructor creates two `Button` instances and attaches event handlers to the `Click` events.

Field-like events

Within the program text of the class or struct that contains the declaration of an event, certain events can be used like fields. To be used in this way, an event must not be `abstract` or `extern`, and must not explicitly include *event_accessor_declarations*. Such an event can be used in any context that permits a field. The field contains a delegate (Delegates) which refers to the list of event handlers that have been added to the event. If no event handlers have been added, the field contains `null`.

In the example

```
public delegate void EventHandler(object sender, EventArgs e);

public class Button: Control
{
    public event EventHandler Click;

    protected void OnClick(EventArgs e) {
        if (Click != null) Click(this, e);
    }

    public void Reset() {
        Click = null;
    }
}
```

`Click` is used as a field within the `Button` class. As the example demonstrates, the field can be examined, modified, and used in delegate invocation expressions. The `OnClick` method in the `Button` class "raises" the `Click` event. The notion of raising an event is precisely equivalent to invoking the delegate represented by the event—thus, there are no special language constructs for raising events. Note that the delegate invocation is preceded by a check that ensures the delegate is non-null.

Outside the declaration of the `Button` class, the `Click` member can only be used on the left-hand side of the `+=` and `-=` operators, as in

```
b.Click += new EventHandler(...);
```

which appends a delegate to the invocation list of the `Click` event, and

```
b.Click -= new EventHandler(...);
```

which removes a delegate from the invocation list of the `Click` event.

When compiling a field-like event, the compiler automatically creates storage to hold the delegate, and creates accessors for the event that add or remove event handlers to the delegate field. The addition and removal operations are thread safe, and may (but are not required to) be done while holding the lock (The lock statement) on the containing object for an instance event, or the type object (Anonymous object creation expressions) for a static event.

Thus, an instance event declaration of the form:

```
class X
{
    public event D Ev;
}
```

will be compiled to something equivalent to:

```
class X
{
    private D __Ev;  // field to hold the delegate

    public event D Ev {
        add {
            /* add the delegate in a thread safe way */
        }

        remove {
            /* remove the delegate in a thread safe way */
        }
    }
}
```

Within the class `X`, references to `Ev` on the left-hand side of the `+=` and `-=` operators cause the add and remove accessors to be invoked. All other references to `Ev` are compiled to reference the hidden field `__Ev` instead (Member access). The name "`__Ev`" is arbitrary; the hidden field could have any name or no name at all.

Event accessors

Event declarations typically omit *event_accessor_declarations*, as in the `Button` example above. One situation for doing so involves the case in which the storage cost of one field per event is not acceptable. In such cases, a class can include *event_accessor_declarations* and use a private mechanism for storing the list of event handlers.

The *event_accessor_declarations* of an event specify the executable statements associated with adding and removing event handlers.

The accessor declarations consist of an *add_accessor_declaration* and a *remove_accessor_declaration*. Each accessor declaration consists of the token `add` or `remove` followed by a *block*. The *block* associated with an *add_accessor_declaration* specifies the statements to execute when an event handler is added, and the *block* associated with a *remove_accessor_declaration* specifies the statements to execute when an event handler is removed.

Each *add_accessor_declaration* and *remove_accessor_declaration* corresponds to a method with a single value parameter of the event type and a `void` return type. The implicit parameter of an event accessor is named `value`. When an event is used in an event assignment, the appropriate event accessor is used. Specifically, if the assignment operator is `+=` then the add accessor is used, and if the assignment operator is `-=` then the remove accessor is used. In either case, the right-hand operand of the assignment operator is used as the argument to the event accessor. The block of an *add_accessor_declaration* or a *remove_accessor_declaration* must conform to the rules for `void` methods described in Method body. In particular, `return` statements in such a block are not permitted to specify an expression.

Since an event accessor implicitly has a parameter named `value`, it is a compile-time error for a local variable or constant declared in an event accessor to have that name.

In the example

```
class Control: Component
{
    // Unique keys for events
    static readonly object mouseDownEventKey = new object();
    static readonly object mouseUpEventKey = new object();

    // Return event handler associated with key
    protected Delegate GetEventHandler(object key) {...}

    // Add event handler associated with key
    protected void AddEventHandler(object key, Delegate handler) {...}

    // Remove event handler associated with key
    protected void RemoveEventHandler(object key, Delegate handler) {...}

    // MouseDown event
    public event MouseEventHandler MouseDown {
        add { AddEventHandler(mouseDownEventKey, value); }
        remove { RemoveEventHandler(mouseDownEventKey, value); }
    }

    // MouseUp event
    public event MouseEventHandler MouseUp {
        add { AddEventHandler(mouseUpEventKey, value); }
        remove { RemoveEventHandler(mouseUpEventKey, value); }
    }

    // Invoke the MouseUp event
    protected void OnMouseUp(MouseEventArgs args) {
        MouseEventHandler handler;
        handler = (MouseEventHandler)GetEventHandler(mouseUpEventKey);
        if (handler != null)
            handler(this, args);
    }
}
```

the `Control` class implements an internal storage mechanism for events. The `AddEventHandler` method associates a delegate value with a key, the `GetEventHandler` method returns the delegate currently associated with a key, and the `RemoveEventHandler` method removes a delegate as an event handler for the specified event. Presumably, the underlying storage mechanism is designed such that there is no cost for associating a `null` delegate value with a key, and thus unhandled events consume no storage.

Static and instance events

When an event declaration includes a `static` modifier, the event is said to be a ***static event***. When no `static` modifier is present, the event is said to be an ***instance event***.

A static event is not associated with a specific instance, and it is a compile-time error to refer to `this` in the accessors of a static event.

An instance event is associated with a given instance of a class, and this instance can be accessed as `this` (This access) in the accessors of that event.

When an event is referenced in a *member_access* (Member access) of the form `E.M`, if `M` is a static event, `E` must denote a type containing `M`, and if `M` is an instance event, E must denote an instance of a type containing `M`.

The differences between static and instance members are discussed further in Static and instance members.

Virtual, sealed, override, and abstract event accessors

A `virtual` event declaration specifies that the accessors of that event are virtual. The `virtual` modifier applies to both accessors of an event.

An `abstract` event declaration specifies that the accessors of the event are virtual, but does not provide an actual implementation of the accessors. Instead, non-abstract derived classes are required to provide their own implementation for the accessors by overriding the event. Because an abstract event declaration provides no actual implementation, it cannot provide brace-delimited *event_accessor_declarations*.

An event declaration that includes both the `abstract` and `override` modifiers specifies that the event is abstract and overrides a base event. The accessors of such an event are also abstract.

Abstract event declarations are only permitted in abstract classes (Abstract classes).

The accessors of an inherited virtual event can be overridden in a derived class by including an event declaration that specifies an `override` modifier. This is known as an ***overriding event declaration***. An overriding event declaration does not declare a new event. Instead, it simply specializes the implementations of the accessors of an existing virtual event.

An overriding event declaration must specify the exact same accessibility modifiers, type, and name as the overridden event.

An overriding event declaration may include the `sealed` modifier. Use of this modifier prevents a derived class from further overriding the event. The accessors of a sealed event are also sealed.

It is a compile-time error for an overriding event declaration to include a `new` modifier.

Except for differences in declaration and invocation syntax, virtual, sealed, override, and abstract accessors behave exactly like virtual, sealed, override and abstract methods. Specifically, the rules described in Virtual methods, Override methods, Sealed methods, and Abstract methods apply as if accessors were methods of a corresponding form. Each accessor corresponds to a method with a single value parameter of the event type, a `void` return type, and the same modifiers as the containing event.

Indexers

An ***indexer*** is a member that enables an object to be indexed in the same way as an array. Indexers are declared using *indexer_declaration*s:

```
indexer_declaration
    : attributes? indexer_modifier* indexer_declarator indexer_body
    ;

indexer_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'virtual'
    | 'sealed'
    | 'override'
    | 'abstract'
    | 'extern'
    | indexer_modifier_unsafe
    ;

indexer_declarator
    : type 'this' '[' formal_parameter_list ']'
    | type interface_type '.' 'this' '[' formal_parameter_list ']'
    ;

indexer_body
    : '{' accessor_declarations '}'
    | '=>' expression ';'
    ;
```

An *indexer_declaration* may include a set of *attributes* (Attributes) and a valid combination of the four access modifiers (Access modifiers), the `new` (The new modifier), `virtual` (Virtual methods), `override` (Override methods), `sealed` (Sealed methods), `abstract` (Abstract methods), and `extern` (External methods) modifiers.

Indexer declarations are subject to the same rules as method declarations (Methods) with regard to valid combinations of modifiers, with the one exception being that the static modifier is not permitted on an indexer declaration.

The modifiers `virtual`, `override`, and `abstract` are mutually exclusive except in one case. The `abstract` and `override` modifiers may be used together so that an abstract indexer can override a virtual one.

The *type* of an indexer declaration specifies the element type of the indexer introduced by the declaration. Unless the indexer is an explicit interface member implementation, the *type* is followed by the keyword `this`. For an explicit interface member implementation, the *type* is followed by an *interface_type*, a "`.`", and the keyword `this`. Unlike other members, indexers do not have user-defined names.

The *formal_parameter_list* specifies the parameters of the indexer. The formal parameter list of an indexer corresponds to that of a method (Method parameters), except that at least one parameter must be specified, and that the `ref` and `out` parameter modifiers are not permitted.

The *type* of an indexer and each of the types referenced in the *formal_parameter_list* must be at least as accessible as the indexer itself (Accessibility constraints).

An *indexer_body* may either consist of an ***accessor body*** or an ***expression body***. In an accessor body, *accessor_declarations*, which must be enclosed in "`{`" and "`}`" tokens, declare the accessors (Accessors) of the property. The accessors specify the executable statements associated with reading and writing the property.

An expression body consisting of "`=>`" followed by an expression `E` and a semicolon is exactly equivalent to the statement body `{ get { return E; } }`, and can therefore only be used to specify getter-only indexers where the result of the getter is given by a single expression.

Even though the syntax for accessing an indexer element is the same as that for an array element, an indexer

element is not classified as a variable. Thus, it is not possible to pass an indexer element as a `ref` or `out` argument.

The formal parameter list of an indexer defines the signature (Signatures and overloading) of the indexer. Specifically, the signature of an indexer consists of the number and types of its formal parameters. The element type and names of the formal parameters are not part of an indexer's signature.

The signature of an indexer must differ from the signatures of all other indexers declared in the same class.

Indexers and properties are very similar in concept, but differ in the following ways:

- A property is identified by its name, whereas an indexer is identified by its signature.
- A property is accessed through a *simple_name* (Simple names) or a *member_access* (Member access), whereas an indexer element is accessed through an *element_access* (Indexer access).
- A property can be a `static` member, whereas an indexer is always an instance member.
- A `get` accessor of a property corresponds to a method with no parameters, whereas a `get` accessor of an indexer corresponds to a method with the same formal parameter list as the indexer.
- A `set` accessor of a property corresponds to a method with a single parameter named `value`, whereas a `set` accessor of an indexer corresponds to a method with the same formal parameter list as the indexer, plus an additional parameter named `value`.
- It is a compile-time error for an indexer accessor to declare a local variable with the same name as an indexer parameter.
- In an overriding property declaration, the inherited property is accessed using the syntax `base.P`, where `P` is the property name. In an overriding indexer declaration, the inherited indexer is accessed using the syntax `base[E]`, where `E` is a comma separated list of expressions.
- There is no concept of an "automatically implemented indexer". It is an error to have a non-abstract, non-external indexer with semicolon accessors.

Aside from these differences, all rules defined in Accessors and Automatically implemented properties apply to indexer accessors as well as to property accessors.

When an indexer declaration includes an `extern` modifier, the indexer is said to be an ***external indexer***. Because an external indexer declaration provides no actual implementation, each of its *accessor_declarations* consists of a semicolon.

The example below declares a `BitArray` class that implements an indexer for accessing the individual bits in the bit array.

```
using System;

class BitArray
{
    int[] bits;
    int length;

    public BitArray(int length) {
        if (length < 0) throw new ArgumentException();
        bits = new int[((length - 1) >> 5) + 1];
        this.length = length;
    }

    public int Length {
        get { return length; }
    }

    public bool this[int index] {
        get {
            if (index < 0 || index >= length) {
                throw new IndexOutOfRangeException();
            }
            return (bits[index >> 5] & 1 << index) != 0;
        }
        set {
            if (index < 0 || index >= length) {
                throw new IndexOutOfRangeException();
            }
            if (value) {
                bits[index >> 5] |= 1 << index;
            }
            else {
                bits[index >> 5] &= ~(1 << index);
            }
        }
    }
}
```

An instance of the `BitArray` class consumes substantially less memory than a corresponding `bool[]` (since each value of the former occupies only one bit instead of the latter's one byte), but it permits the same operations as a `bool[]`.

The following `CountPrimes` class uses a `BitArray` and the classical "sieve" algorithm to compute the number of primes between 1 and a given maximum:

```
class CountPrimes
{
    static int Count(int max) {
        BitArray flags = new BitArray(max + 1);
        int count = 1;
        for (int i = 2; i <= max; i++) {
            if (!flags[i]) {
                for (int j = i * 2; j <= max; j += i) flags[j] = true;
                count++;
            }
        }
        return count;
    }

    static void Main(string[] args) {
        int max = int.Parse(args[0]);
        int count = Count(max);
        Console.WriteLine("Found {0} primes between 1 and {1}", count, max);
    }
}
```

Note that the syntax for accessing elements of the `BitArray` is precisely the same as for a `bool[]`.

The following example shows a 26 * 10 grid class that has an indexer with two parameters. The first parameter is required to be an upper- or lowercase letter in the range A-Z, and the second is required to be an integer in the range 0-9.

```
using System;

class Grid
{
    const int NumRows = 26;
    const int NumCols = 10;

    int[,] cells = new int[NumRows, NumCols];

    public int this[char c, int col] {
        get {
            c = Char.ToUpper(c);
            if (c < 'A' || c > 'Z') {
                throw new ArgumentException();
            }
            if (col < 0 || col >= NumCols) {
                throw new IndexOutOfRangeException();
            }
            return cells[c - 'A', col];
        }

        set {
            c = Char.ToUpper(c);
            if (c < 'A' || c > 'Z') {
                throw new ArgumentException();
            }
            if (col < 0 || col >= NumCols) {
                throw new IndexOutOfRangeException();
            }
            cells[c - 'A', col] = value;
        }
    }
}
```

Indexer overloading

The indexer overload resolution rules are described in Type inference.

Operators

An ***operator*** is a member that defines the meaning of an expression operator that can be applied to instances of the class. Operators are declared using *operator_declaration*s:

```
operator_declaration
    : attributes? operator_modifier+ operator_declarator operator_body
    ;

operator_modifier
    : 'public'
    | 'static'
    | 'extern'
    | operator_modifier_unsafe
    ;

operator_declarator
    : unary_operator_declarator
    | binary_operator_declarator
    | conversion_operator_declarator
    ;

unary_operator_declarator
    : type 'operator' overloadable_unary_operator '(' type identifier ')'
    ;

overloadable_unary_operator
    : '+' | '-' | '!' | '~' | '++' | '--' | 'true' | 'false'
    ;

binary_operator_declarator
    : type 'operator' overloadable_binary_operator '(' type identifier ',' type identifier ')'
    ;

overloadable_binary_operator
    : '+'   | '-'   | '*'   | '/'   | '%'   | '&'   | '|'   | '^'   | '<<'
    | right_shift | '=='  | '!='  | '>'   | '<'   | '>='  | '<='
    ;

conversion_operator_declarator
    : 'implicit' 'operator' type '(' type identifier ')'
    | 'explicit' 'operator' type '(' type identifier ')'
    ;

operator_body
    : block
    | '=>' expression ';'
    | ';'
    ;
```

There are three categories of overloadable operators: Unary operators (Unary operators), binary operators (Binary operators), and conversion operators (Conversion operators).

The *operator_body* is either a semicolon, a ***statement body*** or an ***expression body***. A statement body consists of a *block*, which specifies the statements to execute when the operator is invoked. The *block* must conform to the rules for value-returning methods described in Method body. An expression body consists of `=>` followed by an expression and a semicolon, and denotes a single expression to perform when the operator is invoked.

For `extern` operators, the *operator_body* consists simply of a semicolon. For all other operators, the *operator_body* is either a block body or an expression body.

The following rules apply to all operator declarations:

- An operator declaration must include both a `public` and a `static` modifier.
- The parameter(s) of an operator must be value parameters (Value parameters). It is a compile-time error for an operator declaration to specify `ref` or `out` parameters.
- The signature of an operator (Unary operators, Binary operators, Conversion operators) must differ from the signatures of all other operators declared in the same class.
- All types referenced in an operator declaration must be at least as accessible as the operator itself (Accessibility constraints).
- It is an error for the same modifier to appear multiple times in an operator declaration.

Each operator category imposes additional restrictions, as described in the following sections.

Like other members, operators declared in a base class are inherited by derived classes. Because operator declarations always require the class or struct in which the operator is declared to participate in the signature of the operator, it is not possible for an operator declared in a derived class to hide an operator declared in a base class. Thus, the `new` modifier is never required, and therefore never permitted, in an operator declaration.

Additional information on unary and binary operators can be found in Operators.

Additional information on conversion operators can be found in User-defined conversions.

Unary operators

The following rules apply to unary operator declarations, where `T` denotes the instance type of the class or struct that contains the operator declaration:

- A unary `+`, `-`, `!`, or `~` operator must take a single parameter of type `T` or `T?` and can return any type.
- A unary `++` or `--` operator must take a single parameter of type `T` or `T?` and must return that same type or a type derived from it.
- A unary `true` or `false` operator must take a single parameter of type `T` or `T?` and must return type `bool`.

The signature of a unary operator consists of the operator token (`+`, `-`, `!`, `~`, `++`, `--`, `true`, or `false`) and the type of the single formal parameter. The return type is not part of a unary operator's signature, nor is the name of the formal parameter.

The `true` and `false` unary operators require pair-wise declaration. A compile-time error occurs if a class declares one of these operators without also declaring the other. The `true` and `false` operators are described further in User-defined conditional logical operators and Boolean expressions.

The following example shows an implementation and subsequent usage of `operator ++` for an integer vector class:

```
public class IntVector
{
    public IntVector(int length) {...}

    public int Length {...}                // read-only property

    public int this[int index] {...}       // read-write indexer

    public static IntVector operator ++(IntVector iv) {
        IntVector temp = new IntVector(iv.Length);
        for (int i = 0; i < iv.Length; i++)
            temp[i] = iv[i] + 1;
        return temp;
    }
}

class Test
{
    static void Main() {
        IntVector iv1 = new IntVector(4);    // vector of 4 x 0
        IntVector iv2;

        iv2 = iv1++;    // iv2 contains 4 x 0, iv1 contains 4 x 1
        iv2 = ++iv1;    // iv2 contains 4 x 2, iv1 contains 4 x 2
    }
}
```

Note how the operator method returns the value produced by adding 1 to the operand, just like the postfix increment and decrement operators (Postfix increment and decrement operators), and the prefix increment and decrement operators (Prefix increment and decrement operators). Unlike in C++, this method need not modify the value of its operand directly. In fact, modifying the operand value would violate the standard semantics of the postfix increment operator.

Binary operators

The following rules apply to binary operator declarations, where `T` denotes the instance type of the class or struct that contains the operator declaration:

- A binary non-shift operator must take two parameters, at least one of which must have type `T` or `T?`, and can return any type.
- A binary `<<` or `>>` operator must take two parameters, the first of which must have type `T` or `T?` and the second of which must have type `int` or `int?`, and can return any type.

The signature of a binary operator consists of the operator token (`+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>`, `==`, `!=`, `>`, `<`, `>=`, or `<=`) and the types of the two formal parameters. The return type and the names of the formal parameters are not part of a binary operator's signature.

Certain binary operators require pair-wise declaration. For every declaration of either operator of a pair, there must be a matching declaration of the other operator of the pair. Two operator declarations match when they have the same return type and the same type for each parameter. The following operators require pair-wise declaration:

- `operator ==` and `operator !=`
- `operator >` and `operator <`
- `operator >=` and `operator <=`

Conversion operators

A conversion operator declaration introduces a ***user-defined conversion*** (User-defined conversions) which augments the pre-defined implicit and explicit conversions.

A conversion operator declaration that includes the `implicit` keyword introduces a user-defined implicit conversion. Implicit conversions can occur in a variety of situations, including function member invocations, cast expressions, and assignments. This is described further in Implicit conversions.

A conversion operator declaration that includes the `explicit` keyword introduces a user-defined explicit conversion. Explicit conversions can occur in cast expressions, and are described further in Explicit conversions.

A conversion operator converts from a source type, indicated by the parameter type of the conversion operator, to a target type, indicated by the return type of the conversion operator.

For a given source type `S` and target type `T`, if `S` or `T` are nullable types, let `S0` and `T0` refer to their underlying types, otherwise `S0` and `T0` are equal to `S` and `T` respectively. A class or struct is permitted to declare a conversion from a source type `S` to a target type `T` only if all of the following are true:

- `S0` and `T0` are different types.
- Either `S0` or `T0` is the class or struct type in which the operator declaration takes place.
- Neither `S0` nor `T0` is an *interface_type*.
- Excluding user-defined conversions, a conversion does not exist from `S` to `T` or from `T` to `S`.

For the purposes of these rules, any type parameters associated with `S` or `T` are considered to be unique types that have no inheritance relationship with other types, and any constraints on those type parameters are ignored.

In the example

```
class C<T> {...}

class D<T>: C<T>
{
    public static implicit operator C<int>(D<T> value) {...}      // Ok
    public static implicit operator C<string>(D<T> value) {...}   // Ok
    public static implicit operator C<T>(D<T> value) {...}        // Error
}
```

the first two operator declarations are permitted because, for the purposes of Indexers.3, `T` and `int` and `string` respectively are considered unique types with no relationship. However, the third operator is an error because `C<T>` is the base class of `D<T>`.

From the second rule it follows that a conversion operator must convert either to or from the class or struct type in which the operator is declared. For example, it is possible for a class or struct type `C` to define a conversion from `C` to `int` and from `int` to `C`, but not from `int` to `bool`.

It is not possible to directly redefine a pre-defined conversion. Thus, conversion operators are not allowed to convert from or to `object` because implicit and explicit conversions already exist between `object` and all other types. Likewise, neither the source nor the target types of a conversion can be a base type of the other, since a conversion would then already exist.

However, it is possible to declare operators on generic types that, for particular type arguments, specify conversions that already exist as pre-defined conversions. In the example

```
struct Convertible<T>
{
    public static implicit operator Convertible<T>(T value) {...}
    public static explicit operator T(Convertible<T> value) {...}
}
```

when type `object` is specified as a type argument for `T`, the second operator declares a conversion that

already exists (an implicit, and therefore also an explicit, conversion exists from any type to type `object`).

In cases where a pre-defined conversion exists between two types, any user-defined conversions between those types are ignored. Specifically:

- If a pre-defined implicit conversion (Implicit conversions) exists from type `S` to type `T`, all user-defined conversions (implicit or explicit) from `S` to `T` are ignored.
- If a pre-defined explicit conversion (Explicit conversions) exists from type `S` to type `T`, any user-defined explicit conversions from `S` to `T` are ignored. Furthermore:

If `T` is an interface type, user-defined implicit conversions from `S` to `T` are ignored.

Otherwise, user-defined implicit conversions from `S` to `T` are still considered.

For all types but `object`, the operators declared by the `Convertible<T>` type above do not conflict with pre-defined conversions. For example:

```
void F(int i, Convertible<int> n) {
    i = n;                          // Error
    i = (int)n;                     // User-defined explicit conversion
    n = i;                          // User-defined implicit conversion
    n = (Convertible<int>)i;        // User-defined implicit conversion
}
```

However, for type `object`, pre-defined conversions hide the user-defined conversions in all cases but one:

```
void F(object o, Convertible<object> n) {
    o = n;                          // Pre-defined boxing conversion
    o = (object)n;                  // Pre-defined boxing conversion
    n = o;                          // User-defined implicit conversion
    n = (Convertible<object>)o;     // Pre-defined unboxing conversion
}
```

User-defined conversions are not allowed to convert from or to *interface_type*s. In particular, this restriction ensures that no user-defined transformations occur when converting to an *interface_type*, and that a conversion to an *interface_type* succeeds only if the object being converted actually implements the specified *interface_type*.

The signature of a conversion operator consists of the source type and the target type. (Note that this is the only form of member for which the return type participates in the signature.) The `implicit` or `explicit` classification of a conversion operator is not part of the operator's signature. Thus, a class or struct cannot declare both an `implicit` and an `explicit` conversion operator with the same source and target types.

In general, user-defined implicit conversions should be designed to never throw exceptions and never lose information. If a user-defined conversion can give rise to exceptions (for example, because the source argument is out of range) or loss of information (such as discarding high-order bits), then that conversion should be defined as an explicit conversion.

In the example

```
using System;

public struct Digit
{
    byte value;

    public Digit(byte value) {
        if (value < 0 || value > 9) throw new ArgumentException();
        this.value = value;
    }

    public static implicit operator byte(Digit d) {
        return d.value;
    }

    public static explicit operator Digit(byte b) {
        return new Digit(b);
    }
}
```

the conversion from `Digit` to `byte` is implicit because it never throws exceptions or loses information, but the conversion from `byte` to `Digit` is explicit since `Digit` can only represent a subset of the possible values of a `byte`.

Instance constructors

An ***instance constructor*** is a member that implements the actions required to initialize an instance of a class. Instance constructors are declared using *constructor_declaration*s:

```
constructor_declaration
    : attributes? constructor_modifier* constructor_declarator constructor_body
    ;

constructor_modifier
    : 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'extern'
    | constructor_modifier_unsafe
    ;

constructor_declarator
    : identifier '(' formal_parameter_list? ')' constructor_initializer?
    ;

constructor_initializer
    : ':' 'base' '(' argument_list? ')'
    | ':' 'this' '(' argument_list? ')'
    ;

constructor_body
    : block
    | ';'
    ;
```

A *constructor_declaration* may include a set of *attributes* (Attributes), a valid combination of the four access modifiers (Access modifiers), and an `extern` (External methods) modifier. A constructor declaration is not permitted to include the same modifier multiple times.

The *identifier* of a *constructor_declarator* must name the class in which the instance constructor is declared. If

any other name is specified, a compile-time error occurs.

The optional *formal_parameter_list* of an instance constructor is subject to the same rules as the *formal_parameter_list* of a method (Methods). The formal parameter list defines the signature (Signatures and overloading) of an instance constructor and governs the process whereby overload resolution (Type inference) selects a particular instance constructor in an invocation.

Each of the types referenced in the *formal_parameter_list* of an instance constructor must be at least as accessible as the constructor itself (Accessibility constraints).

The optional *constructor_initializer* specifies another instance constructor to invoke before executing the statements given in the *constructor_body* of this instance constructor. This is described further in Constructor initializers.

When a constructor declaration includes an `extern` modifier, the constructor is said to be an ***external constructor***. Because an external constructor declaration provides no actual implementation, its *constructor_body* consists of a semicolon. For all other constructors, the *constructor_body* consists of a *block* which specifies the statements to initialize a new instance of the class. This corresponds exactly to the *block* of an instance method with a `void` return type (Method body).

Instance constructors are not inherited. Thus, a class has no instance constructors other than those actually declared in the class. If a class contains no instance constructor declarations, a default instance constructor is automatically provided (Default constructors).

Instance constructors are invoked by *object_creation_expression*s (Object creation expressions) and through *constructor_initializer*s.

Constructor initializers

All instance constructors (except those for class `object`) implicitly include an invocation of another instance constructor immediately before the *constructor_body*. The constructor to implicitly invoke is determined by the *constructor_initializer*:

- An instance constructor initializer of the form `base(argument_list)` or `base()` causes an instance constructor from the direct base class to be invoked. That constructor is selected using *argument_list* if present and the overload resolution rules of Overload resolution. The set of candidate instance constructors consists of all accessible instance constructors contained in the direct base class, or the default constructor (Default constructors), if no instance constructors are declared in the direct base class. If this set is empty, or if a single best instance constructor cannot be identified, a compile-time error occurs.
- An instance constructor initializer of the form `this(argument-list)` or `this()` causes an instance constructor from the class itself to be invoked. The constructor is selected using *argument_list* if present and the overload resolution rules of Overload resolution. The set of candidate instance constructors consists of all accessible instance constructors declared in the class itself. If this set is empty, or if a single best instance constructor cannot be identified, a compile-time error occurs. If an instance constructor declaration includes a constructor initializer that invokes the constructor itself, a compile-time error occurs.

If an instance constructor has no constructor initializer, a constructor initializer of the form `base()` is implicitly provided. Thus, an instance constructor declaration of the form

```
C(...) {...}
```

is exactly equivalent to

```
C(...): base() {...}
```

The scope of the parameters given by the *formal_parameter_list* of an instance constructor declaration includes

the constructor initializer of that declaration. Thus, a constructor initializer is permitted to access the parameters of the constructor. For example:

```
class A
{
    public A(int x, int y) {}
}

class B: A
{
    public B(int x, int y): base(x + y, x - y) {}
}
```

An instance constructor initializer cannot access the instance being created. Therefore it is a compile-time error to reference `this` in an argument expression of the constructor initializer, as is it a compile-time error for an argument expression to reference any instance member through a *simple_name*.

Instance variable initializers

When an instance constructor has no constructor initializer, or it has a constructor initializer of the form `base(...)`, that constructor implicitly performs the initializations specified by the *variable_initializer*s of the instance fields declared in its class. This corresponds to a sequence of assignments that are executed immediately upon entry to the constructor and before the implicit invocation of the direct base class constructor. The variable initializers are executed in the textual order in which they appear in the class declaration.

Constructor execution

Variable initializers are transformed into assignment statements, and these assignment statements are executed before the invocation of the base class instance constructor. This ordering ensures that all instance fields are initialized by their variable initializers before any statements that have access to that instance are executed.

Given the example

```
using System;

class A
{
    public A() {
        PrintFields();
    }

    public virtual void PrintFields() {}
}

class B: A
{
    int x = 1;
    int y;

    public B() {
        y = -1;
    }

    public override void PrintFields() {
        Console.WriteLine("x = {0}, y = {1}", x, y);
    }
}
```

when `new B()` is used to create an instance of `B`, the following output is produced:

```
x = 1, y = 0
```

The value of `x` is 1 because the variable initializer is executed before the base class instance constructor is invoked. However, the value of `y` is 0 (the default value of an `int`) because the assignment to `y` is not executed until after the base class constructor returns.

It is useful to think of instance variable initializers and constructor initializers as statements that are automatically inserted before the *constructor_body*. The example

```
using System;
using System.Collections;

class A
{
    int x = 1, y = -1, count;

    public A() {
        count = 0;
    }

    public A(int n) {
        count = n;
    }
}

class B: A
{
    double sqrt2 = Math.Sqrt(2.0);
    ArrayList items = new ArrayList(100);
    int max;

    public B(): this(100) {
        items.Add("default");
    }

    public B(int n): base(n - 1) {
        max = n;
    }
}
```

contains several variable initializers; it also contains constructor initializers of both forms (`base` and `this`). The example corresponds to the code shown below, where each comment indicates an automatically inserted statement (the syntax used for the automatically inserted constructor invocations isn't valid, but merely serves to illustrate the mechanism).

```
using System.Collections;

class A
{
    int x, y, count;

    public A() {
        x = 1;                      // Variable initializer
        y = -1;                     // Variable initializer
        object();                   // Invoke object() constructor
        count = 0;
    }

    public A(int n) {
        x = 1;                      // Variable initializer
        y = -1;                     // Variable initializer
        object();                   // Invoke object() constructor
        count = n;
    }
}

class B: A
{
    double sqrt2;
    ArrayList items;
    int max;

    public B(): this(100) {
        B(100);                     // Invoke B(int) constructor
        items.Add("default");
    }

    public B(int n): base(n - 1) {
        sqrt2 = Math.Sqrt(2.0);     // Variable initializer
        items = new ArrayList(100); // Variable initializer
        A(n - 1);                   // Invoke A(int) constructor
        max = n;
    }
}
```

Default constructors

If a class contains no instance constructor declarations, a default instance constructor is automatically provided. That default constructor simply invokes the parameterless constructor of the direct base class. If the class is abstract then the declared accessibility for the default constructor is protected. Otherwise, the declared accessibility for the default constructor is public. Thus, the default constructor is always of the form

```
protected C(): base() {}
```

or

```
public C(): base() {}
```

where `C` is the name of the class. If overload resolution is unable to determine a unique best candidate for the base class constructor initializer then a compile-time error occurs.

In the example

```
class Message
{
    object sender;
    string text;
}
```

a default constructor is provided because the class contains no instance constructor declarations. Thus, the example is precisely equivalent to

```
class Message
{
    object sender;
    string text;

    public Message(): base() {}
}
```

Private constructors

When a class `T` declares only private instance constructors, it is not possible for classes outside the program text of `T` to derive from `T` or to directly create instances of `T`. Thus, if a class contains only static members and isn't intended to be instantiated, adding an empty private instance constructor will prevent instantiation. For example:

```
public class Trig
{
    private Trig() {}        // Prevent instantiation

    public const double PI = 3.14159265358979323846;

    public static double Sin(double x) {...}
    public static double Cos(double x) {...}
    public static double Tan(double x) {...}
}
```

The `Trig` class groups related methods and constants, but is not intended to be instantiated. Therefore it declares a single empty private instance constructor. At least one instance constructor must be declared to suppress the automatic generation of a default constructor.

Optional instance constructor parameters

The `this(...)` form of constructor initializer is commonly used in conjunction with overloading to implement optional instance constructor parameters. In the example

```
class Text
{
    public Text(): this(0, 0, null) {}

    public Text(int x, int y): this(x, y, null) {}

    public Text(int x, int y, string s) {
        // Actual constructor implementation
    }
}
```

the first two instance constructors merely provide the default values for the missing arguments. Both use a `this(...)` constructor initializer to invoke the third instance constructor, which actually does the work of initializing the new instance. The effect is that of optional constructor parameters:

```
Text t1 = new Text();                   // Same as Text(0, 0, null)
Text t2 = new Text(5, 10);              // Same as Text(5, 10, null)
Text t3 = new Text(5, 20, "Hello");
```

Static constructors

A ***static constructor*** is a member that implements the actions required to initialize a closed class type. Static constructors are declared using *static_constructor_declaration*s:

```
static_constructor_declaration
    : attributes? static_constructor_modifiers identifier '(' ')' static_constructor_body
    ;

static_constructor_modifiers
    : 'extern'? 'static'
    | 'static' 'extern'?
    | static_constructor_modifiers_unsafe
    ;

static_constructor_body
    : block
    | ';'
    ;
```

A *static_constructor_declaration* may include a set of *attributes* (Attributes) and an `extern` modifier (External methods).

The *identifier* of a *static_constructor_declaration* must name the class in which the static constructor is declared. If any other name is specified, a compile-time error occurs.

When a static constructor declaration includes an `extern` modifier, the static constructor is said to be an ***external static constructor***. Because an external static constructor declaration provides no actual implementation, its *static_constructor_body* consists of a semicolon. For all other static constructor declarations, the *static_constructor_body* consists of a *block* which specifies the statements to execute in order to initialize the class. This corresponds exactly to the *method_body* of a static method with a `void` return type (Method body).

Static constructors are not inherited, and cannot be called directly.

The static constructor for a closed class type executes at most once in a given application domain. The execution of a static constructor is triggered by the first of the following events to occur within an application domain:

- An instance of the class type is created.
- Any of the static members of the class type are referenced.

If a class contains the `Main` method (Application Startup) in which execution begins, the static constructor for that class executes before the `Main` method is called.

To initialize a new closed class type, first a new set of static fields (Static and instance fields) for that particular closed type is created. Each of the static fields is initialized to its default value (Default values). Next, the static field initializers (Static field initialization) are executed for those static fields. Finally, the static constructor is executed.

The example

```
using System;

class Test
{
    static void Main() {
        A.F();
        B.F();
    }
}

class A
{
    static A() {
        Console.WriteLine("Init A");
    }
    public static void F() {
        Console.WriteLine("A.F");
    }
}

class B
{
    static B() {
        Console.WriteLine("Init B");
    }
    public static void F() {
        Console.WriteLine("B.F");
    }
}
```

must produce the output:

```
Init A
A.F
Init B
B.F
```

because the execution of `A`'s static constructor is triggered by the call to `A.F`, and the execution of `B`'s static constructor is triggered by the call to `B.F`.

It is possible to construct circular dependencies that allow static fields with variable initializers to be observed in their default value state.

The example

```
using System;

class A
{
    public static int X;

    static A() {
        X = B.Y + 1;
    }
}

class B
{
    public static int Y = A.X + 1;

    static B() {}

    static void Main() {
        Console.WriteLine("X = {0}, Y = {1}", A.X, B.Y);
    }
}
```

produces the output

```
X = 1, Y = 2
```

To execute the `Main` method, the system first runs the initializer for `B.Y`, prior to class `B`'s static constructor. `Y`'s initializer causes `A`'s static constructor to be run because the value of `A.X` is referenced. The static constructor of `A` in turn proceeds to compute the value of `X`, and in doing so fetches the default value of `Y`, which is zero. `A.X` is thus initialized to 1. The process of running `A`'s static field initializers and static constructor then completes, returning to the calculation of the initial value of `Y`, the result of which becomes 2.

Because the static constructor is executed exactly once for each closed constructed class type, it is a convenient place to enforce run-time checks on the type parameter that cannot be checked at compile-time via constraints (Type parameter constraints). For example, the following type uses a static constructor to enforce that the type argument is an enum:

```
class Gen<T> where T: struct
{
    static Gen() {
        if (!typeof(T).IsEnum) {
            throw new ArgumentException("T must be an enum");
        }
    }
}
```

Destructors

A ***destructor*** is a member that implements the actions required to destruct an instance of a class. A destructor is declared using a *destructor_declaration*:

```
destructor_declaration
    : attributes? 'extern'? '~' identifier '(' ')' destructor_body
    | destructor_declaration_unsafe
    ;

destructor_body
    : block
    | ';'
    ;
```

A *destructor_declaration* may include a set of *attributes* (Attributes).

The *identifier* of a *destructor_declaration* must name the class in which the destructor is declared. If any other name is specified, a compile-time error occurs.

When a destructor declaration includes an `extern` modifier, the destructor is said to be an ***external destructor***. Because an external destructor declaration provides no actual implementation, its *destructor_body* consists of a semicolon. For all other destructors, the *destructor_body* consists of a *block* which specifies the statements to execute in order to destruct an instance of the class. A *destructor_body* corresponds exactly to the *method_body* of an instance method with a `void` return type (Method body).

Destructors are not inherited. Thus, a class has no destructors other than the one which may be declared in that class.

Since a destructor is required to have no parameters, it cannot be overloaded, so a class can have, at most, one destructor.

Destructors are invoked automatically, and cannot be invoked explicitly. An instance becomes eligible for destruction when it is no longer possible for any code to use that instance. Execution of the destructor for the instance may occur at any time after the instance becomes eligible for destruction. When an instance is destructed, the destructors in that instance's inheritance chain are called, in order, from most derived to least derived. A destructor may be executed on any thread. For further discussion of the rules that govern when and how a destructor is executed, see Automatic memory management.

The output of the example

```
using System;

class A
{
    ~A() {
        Console.WriteLine("A's destructor");
    }
}

class B: A
{
    ~B() {
        Console.WriteLine("B's destructor");
    }
}

class Test
{
   static void Main() {
        B b = new B();
        b = null;
        GC.Collect();
        GC.WaitForPendingFinalizers();
   }
}
```

is

```
B's destructor
A's destructor
```

since destructors in an inheritance chain are called in order, from most derived to least derived.

Destructors are implemented by overriding the virtual method `Finalize` on `System.Object`. C# programs are not permitted to override this method or call it (or overrides of it) directly. For instance, the program

```
class A
{
    override protected void Finalize() {}    // error

    public void F() {
        this.Finalize();                     // error
    }
}
```

contains two errors.

The compiler behaves as if this method, and overrides of it, do not exist at all. Thus, this program:

```
class A
{
    void Finalize() {}                           // permitted
}
```

is valid, and the method shown hides `System.Object`'s `Finalize` method.

For a discussion of the behavior when an exception is thrown from a destructor, see How exceptions are handled.

Iterators

A function member (Function members) implemented using an iterator block (Blocks) is called an ***iterator***.

An iterator block may be used as the body of a function member as long as the return type of the corresponding function member is one of the enumerator interfaces (Enumerator interfaces) or one of the enumerable interfaces (Enumerable interfaces). It can occur as a *method_body*, *operator_body* or *accessor_body*, whereas events, instance constructors, static constructors and destructors cannot be implemented as iterators.

When a function member is implemented using an iterator block, it is a compile-time error for the formal parameter list of the function member to specify any `ref` or `out` parameters.

Enumerator interfaces

The ***enumerator interfaces*** are the non-generic interface `System.Collections.IEnumerator` and all instantiations of the generic interface `System.Collections.Generic.IEnumerator<T>`. For the sake of brevity, in this chapter these interfaces are referenced as `IEnumerator` and `IEnumerator<T>`, respectively.

Enumerable interfaces

The ***enumerable interfaces*** are the non-generic interface `System.Collections.IEnumerable` and all instantiations of the generic interface `System.Collections.Generic.IEnumerable<T>`. For the sake of brevity, in this chapter these interfaces are referenced as `IEnumerable` and `IEnumerable<T>`, respectively.

Yield type

An iterator produces a sequence of values, all of the same type. This type is called the ***yield type*** of the iterator.

- The yield type of an iterator that returns `IEnumerator` or `IEnumerable` is `object`.
- The yield type of an iterator that returns `IEnumerator<T>` or `IEnumerable<T>` is `T`.

Enumerator objects

When a function member returning an enumerator interface type is implemented using an iterator block, invoking the function member does not immediately execute the code in the iterator block. Instead, an ***enumerator object*** is created and returned. This object encapsulates the code specified in the iterator block, and execution of the code in the iterator block occurs when the enumerator object's `MoveNext` method is invoked. An enumerator object has the following characteristics:

- It implements `IEnumerator` and `IEnumerator<T>`, where `T` is the yield type of the iterator.
- It implements `System.IDisposable`.
- It is initialized with a copy of the argument values (if any) and instance value passed to the function member.
- It has four potential states, ***before***, ***running***, ***suspended***, and ***after***, and is initially in the ***before*** state.

An enumerator object is typically an instance of a compiler-generated enumerator class that encapsulates the code in the iterator block and implements the enumerator interfaces, but other methods of implementation are possible. If an enumerator class is generated by the compiler, that class will be nested, directly or indirectly, in the class containing the function member, it will have private accessibility, and it will have a name reserved for compiler use (Identifiers).

An enumerator object may implement more interfaces than those specified above.

The following sections describe the exact behavior of the `MoveNext`, `Current`, and `Dispose` members of the `IEnumerable` and `IEnumerable<T>` interface implementations provided by an enumerator object.

Note that enumerator objects do not support the `IEnumerator.Reset` method. Invoking this method causes a `System.NotSupportedException` to be thrown.

The MoveNext method

The `MoveNext` method of an enumerator object encapsulates the code of an iterator block. Invoking the

`MoveNext` method executes code in the iterator block and sets the `Current` property of the enumerator object as appropriate. The precise action performed by `MoveNext` depends on the state of the enumerator object when `MoveNext` is invoked:

- If the state of the enumerator object is ***before***, invoking `MoveNext`:
  - Changes the state to ***running***.
  - Initializes the parameters (including `this`) of the iterator block to the argument values and instance value saved when the enumerator object was initialized.
  - Executes the iterator block from the beginning until execution is interrupted (as described below).
- If the state of the enumerator object is ***running***, the result of invoking `MoveNext` is unspecified.
- If the state of the enumerator object is ***suspended***, invoking `MoveNext`:
  - Changes the state to ***running***.
  - Restores the values of all local variables and parameters (including this) to the values saved when execution of the iterator block was last suspended. Note that the contents of any objects referenced by these variables may have changed since the previous call to MoveNext.
  - Resumes execution of the iterator block immediately following the `yield return` statement that caused the suspension of execution and continues until execution is interrupted (as described below).
- If the state of the enumerator object is ***after***, invoking `MoveNext` returns `false`.

When `MoveNext` executes the iterator block, execution can be interrupted in four ways: By a `yield return` statement, by a `yield break` statement, by encountering the end of the iterator block, and by an exception being thrown and propagated out of the iterator block.

- When a `yield return` statement is encountered (The yield statement):
  - The expression given in the statement is evaluated, implicitly converted to the yield type, and assigned to the `Current` property of the enumerator object.
  - Execution of the iterator body is suspended. The values of all local variables and parameters (including `this`) are saved, as is the location of this `yield return` statement. If the `yield return` statement is within one or more `try` blocks, the associated `finally` blocks are not executed at this time.
  - The state of the enumerator object is changed to ***suspended***.
  - The `MoveNext` method returns `true` to its caller, indicating that the iteration successfully advanced to the next value.
- When a `yield break` statement is encountered (The yield statement):
  - If the `yield break` statement is within one or more `try` blocks, the associated `finally` blocks are executed.
  - The state of the enumerator object is changed to ***after***.
  - The `MoveNext` method returns `false` to its caller, indicating that the iteration is complete.
- When the end of the iterator body is encountered:
  - The state of the enumerator object is changed to ***after***.
  - The `MoveNext` method returns `false` to its caller, indicating that the iteration is complete.
- When an exception is thrown and propagated out of the iterator block:
  - Appropriate `finally` blocks in the iterator body will have been executed by the exception propagation.
  - The state of the enumerator object is changed to ***after***.
  - The exception propagation continues to the caller of the `MoveNext` method.

The Current property

An enumerator object's `Current` property is affected by `yield return` statements in the iterator block.

When an enumerator object is in the ***suspended*** state, the value of `Current` is the value set by the previous call

to `MoveNext`. When an enumerator object is in the ***before***, ***running***, or ***after*** states, the result of accessing `Current` is unspecified.

For an iterator with a yield type other than `object`, the result of accessing `Current` through the enumerator object's `IEnumerable` implementation corresponds to accessing `Current` through the enumerator object's `IEnumerator<T>` implementation and casting the result to `object`.

The Dispose method

The `Dispose` method is used to clean up the iteration by bringing the enumerator object to the ***after*** state.

- If the state of the enumerator object is ***before***, invoking `Dispose` changes the state to ***after***.
- If the state of the enumerator object is ***running***, the result of invoking `Dispose` is unspecified.
- If the state of the enumerator object is ***suspended***, invoking `Dispose`:
  - Changes the state to ***running***.
  - Executes any finally blocks as if the last executed `yield return` statement were a `yield break` statement. If this causes an exception to be thrown and propagated out of the iterator body, the state of the enumerator object is set to ***after*** and the exception is propagated to the caller of the `Dispose` method.
  - Changes the state to ***after***.
- If the state of the enumerator object is ***after***, invoking `Dispose` has no affect.

Enumerable objects

When a function member returning an enumerable interface type is implemented using an iterator block, invoking the function member does not immediately execute the code in the iterator block. Instead, an ***enumerable object*** is created and returned. The enumerable object's `GetEnumerator` method returns an enumerator object that encapsulates the code specified in the iterator block, and execution of the code in the iterator block occurs when the enumerator object's `MoveNext` method is invoked. An enumerable object has the following characteristics:

- It implements `IEnumerable` and `IEnumerable<T>`, where `T` is the yield type of the iterator.
- It is initialized with a copy of the argument values (if any) and instance value passed to the function member.

An enumerable object is typically an instance of a compiler-generated enumerable class that encapsulates the code in the iterator block and implements the enumerable interfaces, but other methods of implementation are possible. If an enumerable class is generated by the compiler, that class will be nested, directly or indirectly, in the class containing the function member, it will have private accessibility, and it will have a name reserved for compiler use (Identifiers).

An enumerable object may implement more interfaces than those specified above. In particular, an enumerable object may also implement `IEnumerator` and `IEnumerator<T>`, enabling it to serve as both an enumerable and an enumerator. In that type of implementation, the first time an enumerable object's `GetEnumerator` method is invoked, the enumerable object itself is returned. Subsequent invocations of the enumerable object's `GetEnumerator`, if any, return a copy of the enumerable object. Thus, each returned enumerator has its own state and changes in one enumerator will not affect another.

The GetEnumerator method

An enumerable object provides an implementation of the `GetEnumerator` methods of the `IEnumerable` and `IEnumerable<T>` interfaces. The two `GetEnumerator` methods share a common implementation that acquires and returns an available enumerator object. The enumerator object is initialized with the argument values and instance value saved when the enumerable object was initialized, but otherwise the enumerator object functions as described in Enumerator objects.

Implementation example

This section describes a possible implementation of iterators in terms of standard C# constructs. The

implementation described here is based on the same principles used by the Microsoft C# compiler, but it is by no means a mandated implementation or the only one possible.

The following `Stack<T>` class implements its `GetEnumerator` method using an iterator. The iterator enumerates the elements of the stack in top to bottom order.

```
using System;
using System.Collections;
using System.Collections.Generic;

class Stack<T>: IEnumerable<T>
{
    T[] items;
    int count;

    public void Push(T item) {
        if (items == null) {
            items = new T[4];
        }
        else if (items.Length == count) {
            T[] newItems = new T[count * 2];
            Array.Copy(items, 0, newItems, 0, count);
            items = newItems;
        }
        items[count++] = item;
    }

    public T Pop() {
        T result = items[--count];
        items[count] = default(T);
        return result;
    }

    public IEnumerator<T> GetEnumerator() {
        for (int i = count - 1; i >= 0; --i) yield return items[i];
    }
}
```

The `GetEnumerator` method can be translated into an instantiation of a compiler-generated enumerator class that encapsulates the code in the iterator block, as shown in the following.

```
class Stack<T>: IEnumerable<T>
{
    ...

    public IEnumerator<T> GetEnumerator() {
        return new __Enumerator1(this);
    }

    class __Enumerator1: IEnumerator<T>, IEnumerator
    {
        int __state;
        T __current;
        Stack<T> __this;
        int i;

        public __Enumerator1(Stack<T> __this) {
            this.__this = __this;
        }

        public T Current {
            get { return __current; }
        }

        object IEnumerator.Current {
            get { return __current; }
        }

        public bool MoveNext() {
            switch (__state) {
                case 1: goto __state1;
                case 2: goto __state2;
            }
            i = __this.count - 1;
        __loop:
            if (i < 0) goto __state2;
            __current = __this.items[i];
            __state = 1;
            return true;
        __state1:
            --i;
            goto __loop;
        __state2:
            __state = 2;
            return false;
        }

        public void Dispose() {
            __state = 2;
        }

        void IEnumerator.Reset() {
            throw new NotSupportedException();
        }
    }
}
```

In the preceding translation, the code in the iterator block is turned into a state machine and placed in the `MoveNext` method of the enumerator class. Furthermore, the local variable `i` is turned into a field in the enumerator object so it can continue to exist across invocations of `MoveNext`.

The following example prints a simple multiplication table of the integers 1 through 10. The `FromTo` method in the example returns an enumerable object and is implemented using an iterator.

```
using System;
using System.Collections.Generic;

class Test
{
    static IEnumerable<int> FromTo(int from, int to) {
        while (from <= to) yield return from++;
    }

    static void Main() {
        IEnumerable<int> e = FromTo(1, 10);
        foreach (int x in e) {
            foreach (int y in e) {
                Console.Write("{0,3} ", x * y);
            }
            Console.WriteLine();
        }
    }
}
```

The `FromTo` method can be translated into an instantiation of a compiler-generated enumerable class that encapsulates the code in the iterator block, as shown in the following.

```
using System;
using System.Threading;
using System.Collections;
using System.Collections.Generic;

class Test
{
    ...

    static IEnumerable<int> FromTo(int from, int to) {
        return new __Enumerable1(from, to);
    }

    class __Enumerable1:
        IEnumerable<int>, IEnumerable,
        IEnumerator<int>, IEnumerator
    {
        int __state;
        int __current;
        int __from;
        int from;
        int to;
        int i;

        public __Enumerable1(int __from, int to) {
            this.__from = __from;
            this.to = to;
        }

        public IEnumerator<int> GetEnumerator() {
            __Enumerable1 result = this;
            if (Interlocked.CompareExchange(ref __state, 1, 0) != 0) {
                result = new __Enumerable1(__from, to);
                result.__state = 1;
            }
            result.from = result.__from;
            return result;
        }

        IEnumerator IEnumerable.GetEnumerator() {
            return (IEnumerator)GetEnumerator();
        }
```

```
        public int Current {
            get { return __current; }
        }

        object IEnumerator.Current {
            get { return __current; }
        }

        public bool MoveNext() {
            switch (__state) {
            case 1:
                if (from > to) goto case 2;
                __current = from++;
                __state = 1;
                return true;
            case 2:
                __state = 2;
                return false;
            default:
                throw new InvalidOperationException();
            }
        }

        public void Dispose() {
            __state = 2;
        }

        void IEnumerator.Reset() {
            throw new NotSupportedException();
        }
    }
}
```

The enumerable class implements both the enumerable interfaces and the enumerator interfaces, enabling it to serve as both an enumerable and an enumerator. The first time the `GetEnumerator` method is invoked, the enumerable object itself is returned. Subsequent invocations of the enumerable object's `GetEnumerator`, if any, return a copy of the enumerable object. Thus, each returned enumerator has its own state and changes in one enumerator will not affect another. The `Interlocked.CompareExchange` method is used to ensure thread-safe operation.

The `from` and `to` parameters are turned into fields in the enumerable class. Because `from` is modified in the iterator block, an additional `__from` field is introduced to hold the initial value given to `from` in each enumerator.

The `MoveNext` method throws an `InvalidOperationException` if it is called when `__state` is `0`. This protects against use of the enumerable object as an enumerator object without first calling `GetEnumerator`.

The following example shows a simple tree class. The `Tree<T>` class implements its `GetEnumerator` method using an iterator. The iterator enumerates the elements of the tree in infix order.

```
using System;
using System.Collections.Generic;

class Tree<T>: IEnumerable<T>
{
    T value;
    Tree<T> left;
    Tree<T> right;

    public Tree(T value, Tree<T> left, Tree<T> right) {
        this.value = value;
        this.left = left;
        this.right = right;
    }

    public IEnumerator<T> GetEnumerator() {
        if (left != null) foreach (T x in left) yield x;
        yield value;
        if (right != null) foreach (T x in right) yield x;
    }
}

class Program
{
    static Tree<T> MakeTree<T>(T[] items, int left, int right) {
        if (left > right) return null;
        int i = (left + right) / 2;
        return new Tree<T>(items[i],
            MakeTree(items, left, i - 1),
            MakeTree(items, i + 1, right));
    }

    static Tree<T> MakeTree<T>(params T[] items) {
        return MakeTree(items, 0, items.Length - 1);
    }

    // The output of the program is:
    // 1 2 3 4 5 6 7 8 9
    // Mon Tue Wed Thu Fri Sat Sun

    static void Main() {
        Tree<int> ints = MakeTree(1, 2, 3, 4, 5, 6, 7, 8, 9);
        foreach (int i in ints) Console.Write("{0} ", i);
        Console.WriteLine();

        Tree<string> strings = MakeTree(
            "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun");
        foreach (string s in strings) Console.Write("{0} ", s);
        Console.WriteLine();
    }
}
```

The `GetEnumerator` method can be translated into an instantiation of a compiler-generated enumerator class that encapsulates the code in the iterator block, as shown in the following.

```
class Tree<T>: IEnumerable<T>
{
    ...

    public IEnumerator<T> GetEnumerator() {
        return new __Enumerator1(this);
    }

    class __Enumerator1 : IEnumerator<T>, IEnumerator
    {
        Node<T> this;
```

```
        Node<T> __this;
        IEnumerator<T> __left, __right;
        int __state;
        T __current;

        public __Enumerator1(Node<T> __this) {
            this.__this = __this;
        }

        public T Current {
            get { return __current; }
        }

        object IEnumerator.Current {
            get { return __current; }
        }

        public bool MoveNext() {
            try {
                switch (__state) {

                case 0:
                    __state = -1;
                    if (__this.left == null) goto __yield_value;
                    __left = __this.left.GetEnumerator();
                    goto case 1;

                case 1:
                    __state = -2;
                    if (!__left.MoveNext()) goto __left_dispose;
                    __current = __left.Current;
                    __state = 1;
                    return true;

                __left_dispose:
                    __state = -1;
                    __left.Dispose();

                __yield_value:
                    __current = __this.value;
                    __state = 2;
                    return true;

                case 2:
                    __state = -1;
                    if (__this.right == null) goto __end;
                    __right = __this.right.GetEnumerator();
                    goto case 3;

                case 3:
                    __state = -3;
                    if (!__right.MoveNext()) goto __right_dispose;
                    __current = __right.Current;
                    __state = 3;
                    return true;

                __right_dispose:
                    __state = -1;
                    __right.Dispose();

                __end:
                    __state = 4;
                    break;

                }
            }
            finally {
                if (__state < 0) Dispose();
            }
            return false;
```

```
            return false;
        }

        public void Dispose() {
            try {
                switch (__state) {

                case 1:
                case -2:
                    __left.Dispose();
                    break;

                case 3:
                case -3:
                    __right.Dispose();
                    break;

                }
            }
            finally {
                __state = 4;
            }
        }

        void IEnumerator.Reset() {
            throw new NotSupportedException();
        }
    }
}
```

The compiler generated temporaries used in the `foreach` statements are lifted into the `__left` and `__right` fields of the enumerator object. The `__state` field of the enumerator object is carefully updated so that the correct `Dispose()` method will be called correctly if an exception is thrown. Note that it is not possible to write the translated code with simple `foreach` statements.

Async functions

A method (Methods) or anonymous function (Anonymous function expressions) with the `async` modifier is called an ***async function***. In general, the term ***async*** is used to describe any kind of function that has the `async` modifier.

It is a compile-time error for the formal parameter list of an async function to specify any `ref` or `out` parameters.

The *return_type* of an async method must be either `void` or a ***task type***. The task types are `System.Threading.Tasks.Task` and types constructed from `System.Threading.Tasks.Task<T>`. For the sake of brevity, in this chapter these types are referenced as `Task` and `Task<T>`, respectively. An async method returning a task type is said to be task-returning.

The exact definition of the task types is implementation defined, but from the language's point of view a task type is in one of the states incomplete, succeeded or faulted. A faulted task records a pertinent exception. A succeeded `Task<T>` records a result of type `T`. Task types are awaitable, and can therefore be the operands of await expressions (Await expressions).

An async function invocation has the ability to suspend evaluation by means of await expressions (Await expressions) in its body. Evaluation may later be resumed at the point of the suspending await expression by means of a ***resumption delegate***. The resumption delegate is of type `System.Action`, and when it is invoked, evaluation of the async function invocation will resume from the await expression where it left off. The ***current caller*** of an async function invocation is the original caller if the function invocation has never been suspended, or the most recent caller of the resumption delegate otherwise.

Evaluation of a task-returning async function

Invocation of a task-returning async function causes an instance of the returned task type to be generated. This is called the ***return task*** of the async function. The task is initially in an incomplete state.

The async function body is then evaluated until it is either suspended (by reaching an await expression) or terminates, at which point control is returned to the caller, along with the return task.

When the body of the async function terminates, the return task is moved out of the incomplete state:

- If the function body terminates as the result of reaching a return statement or the end of the body, any result value is recorded in the return task, which is put into a succeeded state.
- If the function body terminates as the result of an uncaught exception (The throw statement) the exception is recorded in the return task which is put into a faulted state.

Evaluation of a void-returning async function

If the return type of the async function is `void`, evaluation differs from the above in the following way: Because no task is returned, the function instead communicates completion and exceptions to the current thread's ***synchronization context***. The exact definition of synchronization context is implementation-dependent, but is a representation of "where" the current thread is running. The synchronization context is notified when evaluation of a void-returning async function commences, completes successfully, or causes an uncaught exception to be thrown.

This allows the context to keep track of how many void-returning async functions are running under it, and to decide how to propagate exceptions coming out of them.

Structs

18.12.2021 • 19 minutes to read

Structs are similar to classes in that they represent data structures that can contain data members and function members. However, unlike classes, structs are value types and do not require heap allocation. A variable of a struct type directly contains the data of the struct, whereas a variable of a class type contains a reference to the data, the latter known as an object.

Structs are particularly useful for small data structures that have value semantics. Complex numbers, points in a coordinate system, or key-value pairs in a dictionary are all good examples of structs. Key to these data structures is that they have few data members, that they do not require use of inheritance or referential identity, and that they can be conveniently implemented using value semantics where assignment copies the value instead of the reference.

As described in Simple types, the simple types provided by C#, such as `int`, `double`, and `bool`, are in fact all struct types. Just as these predefined types are structs, it is also possible to use structs and operator overloading to implement new "primitive" types in the C# language. Two examples of such types are given at the end of this chapter (Struct examples).

Struct declarations

A *struct_declaration* is a *type_declaration* (Type declarations) that declares a new struct:

```
struct_declaration
    : attributes? struct_modifier* 'partial'? 'struct' identifier type_parameter_list?
      struct_interfaces? type_parameter_constraints_clause* struct_body ';'?
    ;
```

A *struct_declaration* consists of an optional set of *attributes* (Attributes), followed by an optional set of *struct_modifier*s (Struct modifiers), followed by an optional `partial` modifier, followed by the keyword `struct` and an *identifier* that names the struct, followed by an optional *type_parameter_list* specification (Type parameters), followed by an optional *struct_interfaces* specification (Partial modifier)), followed by an optional *type_parameter_constraints_clause*s specification (Type parameter constraints), followed by a *struct_body* (Struct body), optionally followed by a semicolon.

Struct modifiers

A *struct_declaration* may optionally include a sequence of struct modifiers:

```
struct_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | struct_modifier_unsafe
    ;
```

It is a compile-time error for the same modifier to appear multiple times in a struct declaration.

The modifiers of a struct declaration have the same meaning as those of a class declaration (Class declarations).

Partial modifier

The `partial` modifier indicates that this *struct_declaration* is a partial type declaration. Multiple partial struct declarations with the same name within an enclosing namespace or type declaration combine to form one struct declaration, following the rules specified in Partial types.

Struct interfaces

A struct declaration may include a *struct_interfaces* specification, in which case the struct is said to directly implement the given interface types.

```
struct_interfaces
    : ':' interface_type_list
    ;
```

Interface implementations are discussed further in Interface implementations.

Struct body

The *struct_body* of a struct defines the members of the struct.

```
struct_body
    : '{' struct_member_declaration* '}'
    ;
```

Struct members

The members of a struct consist of the members introduced by its *struct_member_declaration*s and the members inherited from the type `System.ValueType`.

```
struct_member_declaration
    : constant_declaration
    | field_declaration
    | method_declaration
    | property_declaration
    | event_declaration
    | indexer_declaration
    | operator_declaration
    | constructor_declaration
    | static_constructor_declaration
    | type_declaration
    | struct_member_declaration_unsafe
    ;
```

Except for the differences noted in Class and struct differences, the descriptions of class members provided in Class members through Async functions apply to struct members as well.

Class and struct differences

Structs differ from classes in several important ways:

- Structs are value types (Value semantics).
- All struct types implicitly inherit from the class `System.ValueType` (Inheritance).
- Assignment to a variable of a struct type creates a copy of the value being assigned (Assignment).
- The default value of a struct is the value produced by setting all value type fields to their default value and all reference type fields to `null` (Default values).
- Boxing and unboxing operations are used to convert between a struct type and `object` (Boxing and unboxing).

- The meaning of `this` is different for structs (This access).
- Instance field declarations for a struct are not permitted to include variable initializers (Field initializers).
- A struct is not permitted to declare a parameterless instance constructor (Constructors).
- A struct is not permitted to declare a destructor (Destructors).

Value semantics

Structs are value types (Value types) and are said to have value semantics. Classes, on the other hand, are reference types (Reference types) and are said to have reference semantics.

A variable of a struct type directly contains the data of the struct, whereas a variable of a class type contains a reference to the data, the latter known as an object. When a struct `B` contains an instance field of type `A` and `A` is a struct type, it is a compile-time error for `A` to depend on `B` or a type constructed from `B`. A struct `X` ***directly depends on*** a struct `Y` if `X` contains an instance field of type `Y`. Given this definition, the complete set of structs upon which a struct depends is the transitive closure of the ***directly depends on*** relationship. For example

```
struct Node
{
    int data;
    Node next; // error, Node directly depends on itself
}
```

is an error because `Node` contains an instance field of its own type. Another example

```
struct A { B b; }

struct B { C c; }

struct C { A a; }
```

is an error because each of the types `A`, `B`, and `C` depend on each other.

With classes, it is possible for two variables to reference the same object, and thus possible for operations on one variable to affect the object referenced by the other variable. With structs, the variables each have their own copy of the data (except in the case of `ref` and `out` parameter variables), and it is not possible for operations on one to affect the other. Furthermore, because structs are not reference types, it is not possible for values of a struct type to be `null`.

Given the declaration

```
struct Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

the code fragment

```
Point a = new Point(10, 10);
Point b = a;
a.x = 100;
System.Console.WriteLine(b.x);
```

outputs the value `10`. The assignment of `a` to `b` creates a copy of the value, and `b` is thus unaffected by the assignment to `a.x`. Had `Point` instead been declared as a class, the output would be `100` because `a` and `b` would reference the same object.

Inheritance

All struct types implicitly inherit from the class `System.ValueType`, which, in turn, inherits from class `object`. A struct declaration may specify a list of implemented interfaces, but it is not possible for a struct declaration to specify a base class.

Struct types are never abstract and are always implicitly sealed. The `abstract` and `sealed` modifiers are therefore not permitted in a struct declaration.

Since inheritance isn't supported for structs, the declared accessibility of a struct member cannot be `protected` or `protected internal`.

Function members in a struct cannot be `abstract` or `virtual`, and the `override` modifier is allowed only to override methods inherited from `System.ValueType`.

Assignment

Assignment to a variable of a struct type creates a copy of the value being assigned. This differs from assignment to a variable of a class type, which copies the reference but not the object identified by the reference.

Similar to an assignment, when a struct is passed as a value parameter or returned as the result of a function member, a copy of the struct is created. A struct may be passed by reference to a function member using a `ref` or `out` parameter.

When a property or indexer of a struct is the target of an assignment, the instance expression associated with the property or indexer access must be classified as a variable. If the instance expression is classified as a value, a compile-time error occurs. This is described in further detail in Simple assignment.

Default values

As described in Default values, several kinds of variables are automatically initialized to their default value when they are created. For variables of class types and other reference types, this default value is `null`. However, since structs are value types that cannot be `null`, the default value of a struct is the value produced by setting all value type fields to their default value and all reference type fields to `null`.

Referring to the `Point` struct declared above, the example

```
Point[] a = new Point[100];
```

initializes each `Point` in the array to the value produced by setting the `x` and `y` fields to zero.

The default value of a struct corresponds to the value returned by the default constructor of the struct (Default constructors). Unlike a class, a struct is not permitted to declare a parameterless instance constructor. Instead, every struct implicitly has a parameterless instance constructor which always returns the value that results from setting all value type fields to their default value and all reference type fields to `null`.

Structs should be designed to consider the default initialization state a valid state. In the example

```
using System;

struct KeyValuePair
{
    string key;
    string value;

    public KeyValuePair(string key, string value) {
        if (key == null || value == null) throw new ArgumentException();
        this.key = key;
        this.value = value;
    }
}
```

the user-defined instance constructor protects against null values only where it is explicitly called. In cases where a `KeyValuePair` variable is subject to default value initialization, the `key` and `value` fields will be null, and the struct must be prepared to handle this state.

Boxing and unboxing

A value of a class type can be converted to type `object` or to an interface type that is implemented by the class simply by treating the reference as another type at compile-time. Likewise, a value of type `object` or a value of an interface type can be converted back to a class type without changing the reference (but of course a run-time type check is required in this case).

Since structs are not reference types, these operations are implemented differently for struct types. When a value of a struct type is converted to type `object` or to an interface type that is implemented by the struct, a boxing operation takes place. Likewise, when a value of type `object` or a value of an interface type is converted back to a struct type, an unboxing operation takes place. A key difference from the same operations on class types is that boxing and unboxing copies the struct value either into or out of the boxed instance. Thus, following a boxing or unboxing operation, changes made to the unboxed struct are not reflected in the boxed struct.

When a struct type overrides a virtual method inherited from `System.Object` (such as `Equals`, `GetHashCode`, or `ToString`), invocation of the virtual method through an instance of the struct type does not cause boxing to occur. This is true even when the struct is used as a type parameter and the invocation occurs through an instance of the type parameter type. For example:

```
using System;

struct Counter
{
    int value;

    public override string ToString() {
        value++;
        return value.ToString();
    }
}

class Program
{
    static void Test<T>() where T: new() {
        T x = new T();
        Console.WriteLine(x.ToString());
        Console.WriteLine(x.ToString());
        Console.WriteLine(x.ToString());
    }

    static void Main() {
        Test<Counter>();
    }
}
```

The output of the program is:

```
1
2
3
```

Although it is bad style for `ToString` to have side effects, the example demonstrates that no boxing occurred for the three invocations of `x.ToString()`.

Similarly, boxing never implicitly occurs when accessing a member on a constrained type parameter. For example, suppose an interface `ICounter` contains a method `Increment` which can be used to modify a value. If `ICounter` is used as a constraint, the implementation of the `Increment` method is called with a reference to the variable that `Increment` was called on, never a boxed copy.

```
using System;

interface ICounter
{
    void Increment();
}

struct Counter: ICounter
{
    int value;

    public override string ToString() {
        return value.ToString();
    }

    void ICounter.Increment() {
        value++;
    }
}

class Program
{
    static void Test<T>() where T: ICounter, new() {
        T x = new T();
        Console.WriteLine(x);
        x.Increment();                  // Modify x
        Console.WriteLine(x);
        ((ICounter)x).Increment();      // Modify boxed copy of x
        Console.WriteLine(x);
    }

    static void Main() {
        Test<Counter>();
    }
}
```

The first call to `Increment` modifies the value in the variable `x`. This is not equivalent to the second call to `Increment`, which modifies the value in a boxed copy of `x`. Thus, the output of the program is:

```
0
1
1
```

For further details on boxing and unboxing, see Boxing and unboxing.

Meaning of this

Within an instance constructor or instance function member of a class, `this` is classified as a value. Thus, while `this` can be used to refer to the instance for which the function member was invoked, it is not possible to assign to `this` in a function member of a class.

Within an instance constructor of a struct, `this` corresponds to an `out` parameter of the struct type, and within an instance function member of a struct, `this` corresponds to a `ref` parameter of the struct type. In both cases, `this` is classified as a variable, and it is possible to modify the entire struct for which the function member was invoked by assigning to `this` or by passing this as a `ref` or `out` parameter.

Field initializers

As described in Default values, the default value of a struct consists of the value that results from setting all value type fields to their default value and all reference type fields to `null`. For this reason, a struct does not permit instance field declarations to include variable initializers. This restriction applies only to instance fields. Static fields of a struct are permitted to include variable initializers.

The example

```
struct Point
{
    public int x = 1;  // Error, initializer not permitted
    public int y = 1;  // Error, initializer not permitted
}
```

is in error because the instance field declarations include variable initializers.

Constructors

Unlike a class, a struct is not permitted to declare a parameterless instance constructor. Instead, every struct implicitly has a parameterless instance constructor which always returns the value that results from setting all value type fields to their default value and all reference type fields to null (Default constructors). A struct can declare instance constructors having parameters. For example

```
struct Point
{
    int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Given the above declaration, the statements

```
Point p1 = new Point();
Point p2 = new Point(0, 0);
```

both create a `Point` with `x` and `y` initialized to zero.

A struct instance constructor is not permitted to include a constructor initializer of the form `base(...)`.

If the struct instance constructor doesn't specify a constructor initializer, the `this` variable corresponds to an `out` parameter of the struct type, and similar to an `out` parameter, `this` must be definitely assigned (Definite assignment) at every location where the constructor returns. If the struct instance constructor specifies a constructor initializer, the `this` variable corresponds to a `ref` parameter of the struct type, and similar to a `ref` parameter, `this` is considered definitely assigned on entry to the constructor body. Consider the instance constructor implementation below:

```
struct Point
{
    int x, y;

    public int X {
        set { x = value; }
    }

    public int Y {
        set { y = value; }
    }

    public Point(int x, int y) {
        X = x;        // error, this is not yet definitely assigned
        Y = y;        // error, this is not yet definitely assigned
    }
}
```

No instance member function (including the set accessors for the properties `X` and `Y`) can be called until all fields of the struct being constructed have been definitely assigned. The only exception involves automatically implemented properties (Automatically implemented properties). The definite assignment rules (Simple assignment expressions) specifically exempt assignment to an auto-property of a struct type within an instance constructor of that struct type: such an assignment is considered a definite assignment of the hidden backing field of the auto-property. Thus, the following is allowed:

```
struct Point
{
 public int X { get; set; }
 public int Y { get; set; }

 public Point(int x, int y) {
  X = x;  // allowed, definitely assigns backing field
  Y = y;  // allowed, definitely assigns backing field
 }
```

Destructors

A struct is not permitted to declare a destructor.

Static constructors

Static constructors for structs follow most of the same rules as for classes. The execution of a static constructor for a struct type is triggered by the first of the following events to occur within an application domain:

- A static member of the struct type is referenced.
- An explicitly declared constructor of the struct type is called.

The creation of default values (Default values) of struct types does not trigger the static constructor. (An example of this is the initial value of elements in an array.)

Struct examples

The following shows two significant examples of using `struct` types to create types that can be used similarly to the predefined types of the language, but with modified semantics.

Database integer type

The `DBInt` struct below implements an integer type that can represent the complete set of values of the `int` type, plus an additional state that indicates an unknown value. A type with these characteristics is commonly used in databases.

```
using System;

public struct DBInt
{
    // The Null member represents an unknown DBInt value.

    public static readonly DBInt Null = new DBInt();

    // When the defined field is true, this DBInt represents a known value
    // which is stored in the value field. When the defined field is false,
    // this DBInt represents an unknown value, and the value field is 0.

    int value;
    bool defined;

    // Private instance constructor. Creates a DBInt with a known value.

    DBInt(int value) {
        this.value = value;
        this.defined = true;
    }

    // The IsNull property is true if this DBInt represents an unknown value.

    public bool IsNull { get { return !defined; } }

    // The Value property is the known value of this DBInt, or 0 if this
    // DBInt represents an unknown value.

    public int Value { get { return value; } }

    // Implicit conversion from int to DBInt.

    public static implicit operator DBInt(int x) {
        return new DBInt(x);
    }

    // Explicit conversion from DBInt to int. Throws an exception if the
    // given DBInt represents an unknown value.

    public static explicit operator int(DBInt x) {
        if (!x.defined) throw new InvalidOperationException();
        return x.value;
    }

    public static DBInt operator +(DBInt x) {
        return x;
    }

    public static DBInt operator -(DBInt x) {
        return x.defined ? -x.value : Null;
    }

    public static DBInt operator +(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value + y.value: Null;
    }

    public static DBInt operator -(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value - y.value: Null;
    }

    public static DBInt operator *(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value * y.value: Null;
    }

    public static DBInt operator /(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value / y.value: Null;
    }
```

```
    public static DBInt operator %(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value % y.value: Null;
    }

    public static DBBool operator ==(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value == y.value: DBBool.Null;
    }

    public static DBBool operator !=(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value != y.value: DBBool.Null;
    }

    public static DBBool operator >(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value > y.value: DBBool.Null;
    }

    public static DBBool operator <(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value < y.value: DBBool.Null;
    }

    public static DBBool operator >=(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value >= y.value: DBBool.Null;
    }

    public static DBBool operator <=(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value <= y.value: DBBool.Null;
    }

    public override bool Equals(object obj) {
        if (!(obj is DBInt)) return false;
        DBInt x = (DBInt)obj;
        return value == x.value && defined == x.defined;
    }

    public override int GetHashCode() {
        return value;
    }

    public override string ToString() {
        return defined? value.ToString(): "DBInt.Null";
    }
}
```

Database boolean type

The `DBBool` struct below implements a three-valued logical type. The possible values of this type are `DBBool.True`, `DBBool.False`, and `DBBool.Null`, where the `Null` member indicates an unknown value. Such three-valued logical types are commonly used in databases.

```
using System;

public struct DBBool
{
    // The three possible DBBool values.

    public static readonly DBBool Null = new DBBool(0);
    public static readonly DBBool False = new DBBool(-1);
    public static readonly DBBool True = new DBBool(1);

    // Private field that stores -1, 0, 1 for False, Null, True.

    sbyte value;

    // Private instance constructor. The value parameter must be -1, 0, or 1.

    DBBool(int value) {
```

```
    this.value = (sbyte)value;
}

// Properties to examine the value of a DBBool. Return true if this
// DBBool has the given value, false otherwise.

public bool IsNull { get { return value == 0; } }

public bool IsFalse { get { return value < 0; } }

public bool IsTrue { get { return value > 0; } }

// Implicit conversion from bool to DBBool. Maps true to DBBool.True and
// false to DBBool.False.

public static implicit operator DBBool(bool x) {
    return x? True: False;
}

// Explicit conversion from DBBool to bool. Throws an exception if the
// given DBBool is Null, otherwise returns true or false.

public static explicit operator bool(DBBool x) {
    if (x.value == 0) throw new InvalidOperationException();
    return x.value > 0;
}

// Equality operator. Returns Null if either operand is Null, otherwise
// returns True or False.

public static DBBool operator ==(DBBool x, DBBool y) {
    if (x.value == 0 || y.value == 0) return Null;
    return x.value == y.value? True: False;
}

// Inequality operator. Returns Null if either operand is Null, otherwise
// returns True or False.

public static DBBool operator !=(DBBool x, DBBool y) {
    if (x.value == 0 || y.value == 0) return Null;
    return x.value != y.value? True: False;
}

// Logical negation operator. Returns True if the operand is False, Null
// if the operand is Null, or False if the operand is True.

public static DBBool operator !(DBBool x) {
    return new DBBool(-x.value);
}

// Logical AND operator. Returns False if either operand is False,
// otherwise Null if either operand is Null, otherwise True.

public static DBBool operator &(DBBool x, DBBool y) {
    return new DBBool(x.value < y.value? x.value: y.value);
}

// Logical OR operator. Returns True if either operand is True, otherwise
// Null if either operand is Null, otherwise False.

public static DBBool operator |(DBBool x, DBBool y) {
    return new DBBool(x.value > y.value? x.value: y.value);
}

// Definitely true operator. Returns true if the operand is True, false
// otherwise.

public static bool operator true(DBBool x) {
    return x.value > 0;
```

```
        return x.value > 0;
    }

    // Definitely false operator. Returns true if the operand is False, false
    // otherwise.

    public static bool operator false(DBBool x) {
        return x.value < 0;
    }

    public override bool Equals(object obj) {
        if (!(obj is DBBool)) return false;
        return value == ((DBBool)obj).value;
    }

    public override int GetHashCode() {
        return value;
    }

    public override string ToString() {
        if (value > 0) return "DBBool.True";
        if (value < 0) return "DBBool.False";
        return "DBBool.Null";
    }
}
```

Arrays

18.12.2021 • 9 minutes to read

An array is a data structure that contains a number of variables which are accessed through computed indices. The variables contained in an array, also called the elements of the array, are all of the same type, and this type is called the element type of the array.

An array has a rank which determines the number of indices associated with each array element. The rank of an array is also referred to as the dimensions of the array. An array with a rank of one is called a ***single-dimensional array***. An array with a rank greater than one is called a ***multi-dimensional array***. Specific sized multi-dimensional arrays are often referred to as two-dimensional arrays, three-dimensional arrays, and so on.

Each dimension of an array has an associated length which is an integral number greater than or equal to zero. The dimension lengths are not part of the type of the array, but rather are established when an instance of the array type is created at run-time. The length of a dimension determines the valid range of indices for that dimension: For a dimension of length `N`, indices can range from `0` to `N - 1` inclusive. The total number of elements in an array is the product of the lengths of each dimension in the array. If one or more of the dimensions of an array have a length of zero, the array is said to be empty.

The element type of an array can be any type, including an array type.

Array types

An array type is written as a *non_array_type* followed by one or more *rank_specifier*s:

```
array_type
    : non_array_type rank_specifier+
    ;

non_array_type
    : type
    ;

rank_specifier
    : '[' dim_separator* ']'
    ;

dim_separator
    : ','
    ;
```

A *non_array_type* is any *type* that is not itself an *array_type*.

The rank of an array type is given by the leftmost *rank_specifier* in the *array_type*: A *rank_specifier* indicates that the array is an array with a rank of one plus the number of "`,`" tokens in the *rank_specifier*.

The element type of an array type is the type that results from deleting the leftmost *rank_specifier*:

- An array type of the form `T[R]` is an array with rank `R` and a non-array element type `T`.
- An array type of the form `T[R][R1]...[Rn]` is an array with rank `R` and an element type `T[R1]...[Rn]`.

In effect, the *rank_specifier*s are read from left to right before the final non-array element type. The type `int[][,,][,]` is a single-dimensional array of three-dimensional arrays of two-dimensional arrays of `int`.

At run-time, a value of an array type can be `null` or a reference to an instance of that array type.

The System.Array type

The type `System.Array` is the abstract base type of all array types. An implicit reference conversion (Implicit reference conversions) exists from any array type to `System.Array`, and an explicit reference conversion (Explicit reference conversions) exists from `System.Array` to any array type. Note that `System.Array` is not itself an *array_type*. Rather, it is a *class_type* from which all *array_type*s are derived.

At run-time, a value of type `System.Array` can be `null` or a reference to an instance of any array type.

Arrays and the generic IList interface

A one-dimensional array `T[]` implements the interface `System.Collections.Generic.IList<T>` (`IList<T>` for short) and its base interfaces. Accordingly, there is an implicit conversion from `T[]` to `IList<T>` and its base interfaces. In addition, if there is an implicit reference conversion from `S` to `T` then `S[]` implements `IList<T>` and there is an implicit reference conversion from `S[]` to `IList<T>` and its base interfaces (Implicit reference conversions). If there is an explicit reference conversion from `S` to `T` then there is an explicit reference conversion from `S[]` to `IList<T>` and its base interfaces (Explicit reference conversions). For example:

```
using System.Collections.Generic;

class Test
{
    static void Main() {
        string[] sa = new string[5];
        object[] oa1 = new object[5];
        object[] oa2 = sa;

        IList<string> lst1 = sa;                    // Ok
        IList<string> lst2 = oa1;                   // Error, cast needed
        IList<object> lst3 = sa;                    // Ok
        IList<object> lst4 = oa1;                   // Ok

        IList<string> lst5 = (IList<string>)oa1;    // Exception
        IList<string> lst6 = (IList<string>)oa2;    // Ok
    }
}
```

The assignment `lst2 = oa1` generates a compile-time error since the conversion from `object[]` to `IList<string>` is an explicit conversion, not implicit. The cast `(IList<string>)oa1` will cause an exception to be thrown at run-time since `oa1` references an `object[]` and not a `string[]`. However the cast `(IList<string>)oa2` will not cause an exception to be thrown since `oa2` references a `string[]`.

Whenever there is an implicit or explicit reference conversion from `S[]` to `IList<T>`, there is also an explicit reference conversion from `IList<T>` and its base interfaces to `S[]` (Explicit reference conversions).

When an array type `S[]` implements `IList<T>`, some of the members of the implemented interface may throw exceptions. The precise behavior of the implementation of the interface is beyond the scope of this specification.

Array creation

Array instances are created by *array_creation_expression*s (Array creation expressions) or by field or local variable declarations that include an *array_initializer* (Array initializers).

When an array instance is created, the rank and length of each dimension are established and then remain constant for the entire lifetime of the instance. In other words, it is not possible to change the rank of an existing array instance, nor is it possible to resize its dimensions.

An array instance is always of an array type. The `System.Array` type is an abstract type that cannot be

instantiated.

Elements of arrays created by *array_creation_expression*s are always initialized to their default value (Default values).

Array element access

Array elements are accessed using *element_access* expressions (Array access) of the form `A[I1, I2, ..., In]`, where `A` is an expression of an array type and each `Ix` is an expression of type `int`, `uint`, `long`, `ulong`, or can be implicitly converted to one or more of these types. The result of an array element access is a variable, namely the array element selected by the indices.

The elements of an array can be enumerated using a `foreach` statement (The foreach statement).

Array members

Every array type inherits the members declared by the `System.Array` type.

Array covariance

For any two *reference_type*s `A` and `B`, if an implicit reference conversion (Implicit reference conversions) or explicit reference conversion (Explicit reference conversions) exists from `A` to `B`, then the same reference conversion also exists from the array type `A[R]` to the array type `B[R]`, where `R` is any given *rank_specifier* (but the same for both array types). This relationship is known as ***array covariance***. Array covariance in particular means that a value of an array type `A[R]` may actually be a reference to an instance of an array type `B[R]`, provided an implicit reference conversion exists from `B` to `A`.

Because of array covariance, assignments to elements of reference type arrays include a run-time check which ensures that the value being assigned to the array element is actually of a permitted type (Simple assignment). For example:

```
class Test
{
    static void Fill(object[] array, int index, int count, object value) {
        for (int i = index; i < index + count; i++) array[i] = value;
    }

    static void Main() {
        string[] strings = new string[100];
        Fill(strings, 0, 100, "Undefined");
        Fill(strings, 0, 10, null);
        Fill(strings, 90, 10, 0);
    }
}
```

The assignment to `array[i]` in the `Fill` method implicitly includes a run-time check which ensures that the object referenced by `value` is either `null` or an instance that is compatible with the actual element type of `array`. In `Main`, the first two invocations of `Fill` succeed, but the third invocation causes a `System.ArrayTypeMismatchException` to be thrown upon executing the first assignment to `array[i]`. The exception occurs because a boxed `int` cannot be stored in a `string` array.

Array covariance specifically does not extend to arrays of *value_type*s. For example, no conversion exists that permits an `int[]` to be treated as an `object[]`.

Array initializers

Array initializers may be specified in field declarations (Fields), local variable declarations (Local variable declarations), and array creation expressions (Array creation expressions):

```
array_initializer
    : '{' variable_initializer_list? '}'
    | '{' variable_initializer_list ',' '}'
    ;

variable_initializer_list
    : variable_initializer (',' variable_initializer)*
    ;

variable_initializer
    : expression
    | array_initializer
    ;
```

An array initializer consists of a sequence of variable initializers, enclosed by "`{`" and "`}`" tokens and separated by "`,`" tokens. Each variable initializer is an expression or, in the case of a multi-dimensional array, a nested array initializer.

The context in which an array initializer is used determines the type of the array being initialized. In an array creation expression, the array type immediately precedes the initializer, or is inferred from the expressions in the array initializer. In a field or variable declaration, the array type is the type of the field or variable being declared. When an array initializer is used in a field or variable declaration, such as:

```
int[] a = {0, 2, 4, 6, 8};
```

it is simply shorthand for an equivalent array creation expression:

```
int[] a = new int[] {0, 2, 4, 6, 8};
```

For a single-dimensional array, the array initializer must consist of a sequence of expressions that are assignment compatible with the element type of the array. The expressions initialize array elements in increasing order, starting with the element at index zero. The number of expressions in the array initializer determines the length of the array instance being created. For example, the array initializer above creates an `int[]` instance of length 5 and then initializes the instance with the following values:

```
a[0] = 0; a[1] = 2; a[2] = 4; a[3] = 6; a[4] = 8;
```

For a multi-dimensional array, the array initializer must have as many levels of nesting as there are dimensions in the array. The outermost nesting level corresponds to the leftmost dimension and the innermost nesting level corresponds to the rightmost dimension. The length of each dimension of the array is determined by the number of elements at the corresponding nesting level in the array initializer. For each nested array initializer, the number of elements must be the same as the other array initializers at the same level. The example:

```
int[,] b = {{0, 1}, {2, 3}, {4, 5}, {6, 7}, {8, 9}};
```

creates a two-dimensional array with a length of five for the leftmost dimension and a length of two for the rightmost dimension:

```
int[,] b = new int[5, 2];
```

and then initializes the array instance with the following values:

```
b[0, 0] = 0; b[0, 1] = 1;
b[1, 0] = 2; b[1, 1] = 3;
b[2, 0] = 4; b[2, 1] = 5;
b[3, 0] = 6; b[3, 1] = 7;
b[4, 0] = 8; b[4, 1] = 9;
```

If a dimension other than the rightmost is given with length zero, the subsequent dimensions are assumed to also have length zero. The example:

```
int[,] c = {};
```

creates a two-dimensional array with a length of zero for both the leftmost and the rightmost dimension:

```
int[,] c = new int[0, 0];
```

When an array creation expression includes both explicit dimension lengths and an array initializer, the lengths must be constant expressions and the number of elements at each nesting level must match the corresponding dimension length. Here are some examples:

```
int i = 3;
int[] x = new int[3] {0, 1, 2};        // OK
int[] y = new int[i] {0, 1, 2};        // Error, i not a constant
int[] z = new int[3] {0, 1, 2, 3};     // Error, length/initializer mismatch
```

Here, the initializer for `y` results in a compile-time error because the dimension length expression is not a constant, and the initializer for `z` results in a compile-time error because the length and the number of elements in the initializer do not agree.

Interfaces

18.12.2021 • 29 minutes to read

An interface defines a contract. A class or struct that implements an interface must adhere to its contract. An interface may inherit from multiple base interfaces, and a class or struct may implement multiple interfaces.

Interfaces can contain methods, properties, events, and indexers. The interface itself does not provide implementations for the members that it defines. The interface merely specifies the members that must be supplied by classes or structs that implement the interface.

Interface declarations

An *interface_declaration* is a *type_declaration* (Type declarations) that declares a new interface type.

```
interface_declaration
    : attributes? interface_modifier* 'partial'? 'interface'
      identifier variant_type_parameter_list? interface_base?
      type_parameter_constraints_clause* interface_body ';'?
    ;
```

An *interface_declaration* consists of an optional set of *attributes* (Attributes), followed by an optional set of *interface_modifier*s (Interface modifiers), followed by an optional `partial` modifier, followed by the keyword `interface` and an *identifier* that names the interface, followed by an optional *variant_type_parameter_list* specification (Variant type parameter lists), followed by an optional *interface_base* specification (Base interfaces), followed by an optional *type_parameter_constraints_clause*s specification (Type parameter constraints), followed by an *interface_body* (Interface body), optionally followed by a semicolon.

Interface modifiers

An *interface_declaration* may optionally include a sequence of interface modifiers:

```
interface_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | interface_modifier_unsafe
    ;
```

It is a compile-time error for the same modifier to appear multiple times in an interface declaration.

The `new` modifier is only permitted on interfaces defined within a class. It specifies that the interface hides an inherited member by the same name, as described in The new modifier.

The `public`, `protected`, `internal`, and `private` modifiers control the accessibility of the interface. Depending on the context in which the interface declaration occurs, only some of these modifiers may be permitted (Declared accessibility).

Partial modifier

The `partial` modifier indicates that this *interface_declaration* is a partial type declaration. Multiple partial interface declarations with the same name within an enclosing namespace or type declaration combine to form one interface declaration, following the rules specified in Partial types.

Variant type parameter lists

Variant type parameter lists can only occur on interface and delegate types. The difference from ordinary *type_parameter_list*s is the optional *variance_annotation* on each type parameter.

```
variant_type_parameter_list
    : '<' variant_type_parameters '>'
    ;

variant_type_parameters
    : attributes? variance_annotation? type_parameter
    | variant_type_parameters ',' attributes? variance_annotation? type_parameter
    ;

variance_annotation
    : 'in'
    | 'out'
    ;
```

If the variance annotation is `out`, the type parameter is said to be ***covariant***. If the variance annotation is `in`, the type parameter is said to be ***contravariant***. If there is no variance annotation, the type parameter is said to be *invariant*.

In the example

```
interface C<out X, in Y, Z>
{
  X M(Y y);
  Z P { get; set; }
}
```

`X` is covariant, `Y` is contravariant and `Z` is invariant.

Variance safety

The occurrence of variance annotations in the type parameter list of a type restricts the places where types can occur within the type declaration.

A type `T` is ***output-unsafe*** if one of the following holds:

- `T` is a contravariant type parameter
- `T` is an array type with an output-unsafe element type
- `T` is an interface or delegate type `S<A1,...,Ak>` constructed from a generic type `S<X1,...,Xk>` where for at least one `Ai` one of the following holds:
  - `Xi` is covariant or invariant and `Ai` is output-unsafe.
  - `Xi` is contravariant or invariant and `Ai` is input-safe.

A type `T` is ***input-unsafe*** if one of the following holds:

- `T` is a covariant type parameter
- `T` is an array type with an input-unsafe element type
- `T` is an interface or delegate type `S<A1,...,Ak>` constructed from a generic type `S<X1,...,Xk>` where for at least one `Ai` one of the following holds:
  - `Xi` is covariant or invariant and `Ai` is input-unsafe.
  - `Xi` is contravariant or invariant and `Ai` is output-unsafe.

Intuitively, an output-unsafe type is prohibited in an output position, and an input-unsafe type is prohibited in an input position.

A type is ***output-safe*** if it is not output-unsafe, and ***input-safe*** if it is not input-unsafe.

Variance conversion

The purpose of variance annotations is to provide for more lenient (but still type safe) conversions to interface and delegate types. To this end the definitions of implicit (Implicit conversions) and explicit conversions (Explicit conversions) make use of the notion of variance-convertibility, which is defined as follows:

A type `T<A1,...,An>` is variance-convertible to a type `T<B1,...,Bn>` if `T` is either an interface or a delegate type declared with the variant type parameters `T<X1,...,Xn>`, and for each variant type parameter `Xi` one of the following holds:

- `Xi` is covariant and an implicit reference or identity conversion exists from `Ai` to `Bi`
- `Xi` is contravariant and an implicit reference or identity conversion exists from `Bi` to `Ai`
- `Xi` is invariant and an identity conversion exists from `Ai` to `Bi`

Base interfaces

An interface can inherit from zero or more interface types, which are called the ***explicit base interfaces*** of the interface. When an interface has one or more explicit base interfaces, then in the declaration of that interface, the interface identifier is followed by a colon and a comma separated list of base interface types.

```
interface_base
    : ':' interface_type_list
    ;
```

For a constructed interface type, the explicit base interfaces are formed by taking the explicit base interface declarations on the generic type declaration, and substituting, for each *type_parameter* in the base interface declaration, the corresponding *type_argument* of the constructed type.

The explicit base interfaces of an interface must be at least as accessible as the interface itself (Accessibility constraints). For example, it is a compile-time error to specify a `private` or `internal` interface in the *interface_base* of a `public` interface.

It is a compile-time error for an interface to directly or indirectly inherit from itself.

The ***base interfaces*** of an interface are the explicit base interfaces and their base interfaces. In other words, the set of base interfaces is the complete transitive closure of the explicit base interfaces, their explicit base interfaces, and so on. An interface inherits all members of its base interfaces. In the example

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

interface IListBox: IControl
{
    void SetItems(string[] items);
}

interface IComboBox: ITextBox, IListBox {}
```

the base interfaces of `IComboBox` are `IControl`, `ITextBox`, and `IListBox`.

In other words, the `IComboBox` interface above inherits members `SetText` and `SetItems` as well as `Paint`.

Every base interface of an interface must be output-safe (Variance safety). A class or struct that implements an interface also implicitly implements all of the interface's base interfaces.

Interface body

The *interface_body* of an interface defines the members of the interface.

```
interface_body
    : '{' interface_member_declaration* '}'
    ;
```

Interface members

The members of an interface are the members inherited from the base interfaces and the members declared by the interface itself.

```
interface_member_declaration
    : interface_method_declaration
    | interface_property_declaration
    | interface_event_declaration
    | interface_indexer_declaration
    ;
```

An interface declaration may declare zero or more members. The members of an interface must be methods, properties, events, or indexers. An interface cannot contain constants, fields, operators, instance constructors, destructors, or types, nor can an interface contain static members of any kind.

All interface members implicitly have public access. It is a compile-time error for interface member declarations to include any modifiers. In particular, interfaces members cannot be declared with the modifiers `abstract`, `public`, `protected`, `internal`, `private`, `virtual`, `override`, or `static`.

The example

```
public delegate void StringListEvent(IStringList sender);

public interface IStringList
{
    void Add(string s);
    int Count { get; }
    event StringListEvent Changed;
    string this[int index] { get; set; }
}
```

declares an interface that contains one each of the possible kinds of members: A method, a property, an event, and an indexer.

An *interface_declaration* creates a new declaration space (Declarations), and the *interface_member_declaration*s immediately contained by the *interface_declaration* introduce new members into this declaration space. The following rules apply to *interface_member_declaration*s:

- The name of a method must differ from the names of all properties and events declared in the same interface. In addition, the signature (Signatures and overloading) of a method must differ from the signatures of all other methods declared in the same interface, and two methods declared in the same interface may not have signatures that differ solely by `ref` and `out`.
- The name of a property or event must differ from the names of all other members declared in the same interface.
- The signature of an indexer must differ from the signatures of all other indexers declared in the same

interface.

The inherited members of an interface are specifically not part of the declaration space of the interface. Thus, an interface is allowed to declare a member with the same name or signature as an inherited member. When this occurs, the derived interface member is said to hide the base interface member. Hiding an inherited member is not considered an error, but it does cause the compiler to issue a warning. To suppress the warning, the declaration of the derived interface member must include a `new` modifier to indicate that the derived member is intended to hide the base member. This topic is discussed further in Hiding through inheritance.

If a `new` modifier is included in a declaration that doesn't hide an inherited member, a warning is issued to that effect. This warning is suppressed by removing the `new` modifier.

Note that the members in class `object` are not, strictly speaking, members of any interface (Interface members). However, the members in class `object` are available via member lookup in any interface type (Member lookup).

Interface methods

Interface methods are declared using *interface_method_declaration*s:

```
interface_method_declaration
    : attributes? 'new'? return_type identifier type_parameter_list
      '(' formal_parameter_list? ')' type_parameter_constraints_clause* ';'
    ;
```

The *attributes*, *return_type*, *identifier*, and *formal_parameter_list* of an interface method declaration have the same meaning as those of a method declaration in a class (Methods). An interface method declaration is not permitted to specify a method body, and the declaration therefore always ends with a semicolon.

Each formal parameter type of an interface method must be input-safe (Variance safety), and the return type must be either `void` or output-safe. Furthermore, each class type constraint, interface type constraint and type parameter constraint on any type parameter of the method must be input-safe.

These rules ensure that any covariant or contravariant usage of the interface remains type-safe. For example,

```
interface I<out T> { void M<U>() where U : T; }
```

is illegal because the usage of `T` as a type parameter constraint on `U` is not input-safe.

Were this restriction not in place it would be possible to violate type safety in the following manner:

```
class B {}
class D : B{}
class E : B {}
class C : I<D> { public void M<U>() {...} }
...
I<B> b = new C();
b.M<E>();
```

This is actually a call to `C.M<E>`. But that call requires that `E` derive from `D`, so type safety would be violated here.

Interface properties

Interface properties are declared using *interface_property_declaration*s:

```
interface_property_declaration
    : attributes? 'new'? type identifier '{' interface_accessors '}'
    ;

interface_accessors
    : attributes? 'get' ';'
    | attributes? 'set' ';'
    | attributes? 'get' ';' attributes? 'set' ';'
    | attributes? 'set' ';' attributes? 'get' ';'
    ;
```

The *attributes*, *type*, and *identifier* of an interface property declaration have the same meaning as those of a property declaration in a class (Properties).

The accessors of an interface property declaration correspond to the accessors of a class property declaration (Accessors), except that the accessor body must always be a semicolon. Thus, the accessors simply indicate whether the property is read-write, read-only, or write-only.

The type of an interface property must be output-safe if there is a get accessor, and must be input-safe if there is a set accessor.

Interface events

Interface events are declared using *interface_event_declaration*s:

```
interface_event_declaration
    : attributes? 'new'? 'event' type identifier ';'
    ;
```

The *attributes*, *type*, and *identifier* of an interface event declaration have the same meaning as those of an event declaration in a class (Events).

The type of an interface event must be input-safe.

Interface indexers

Interface indexers are declared using *interface_indexer_declaration*s:

```
interface_indexer_declaration
    : attributes? 'new'? type 'this' '[' formal_parameter_list ']' '{' interface_accessors '}'
    ;
```

The *attributes*, *type*, and *formal_parameter_list* of an interface indexer declaration have the same meaning as those of an indexer declaration in a class (Indexers).

The accessors of an interface indexer declaration correspond to the accessors of a class indexer declaration (Indexers), except that the accessor body must always be a semicolon. Thus, the accessors simply indicate whether the indexer is read-write, read-only, or write-only.

All the formal parameter types of an interface indexer must be input-safe . In addition, any `out` or `ref` formal parameter types must also be output-safe. Note that even `out` parameters are required to be input-safe, due to a limitation of the underlying execution platform.

The type of an interface indexer must be output-safe if there is a get accessor, and must be input-safe if there is a set accessor.

Interface member access

Interface members are accessed through member access (Member access) and indexer access (Indexer access) expressions of the form `I.M` and `I[A]`, where `I` is an interface type, `M` is a method, property, or event of that

interface type, and `A` is an indexer argument list.

For interfaces that are strictly single-inheritance (each interface in the inheritance chain has exactly zero or one direct base interface), the effects of the member lookup (Member lookup), method invocation (Method invocations), and indexer access (Indexer access) rules are exactly the same as for classes and structs: More derived members hide less derived members with the same name or signature. However, for multiple-inheritance interfaces, ambiguities can occur when two or more unrelated base interfaces declare members with the same name or signature. This section shows several examples of such situations. In all cases, explicit casts can be used to resolve the ambiguities.

In the example

```
interface IList
{
    int Count { get; set; }
}

interface ICounter
{
    void Count(int i);
}

interface IListCounter: IList, ICounter {}

class C
{
    void Test(IListCounter x) {
        x.Count(1);                 // Error
        x.Count = 1;                // Error
        ((IList)x).Count = 1;       // Ok, invokes IList.Count.set
        ((ICounter)x).Count(1);     // Ok, invokes ICounter.Count
    }
}
```

the first two statements cause compile-time errors because the member lookup (Member lookup) of `Count` in `IListCounter` is ambiguous. As illustrated by the example, the ambiguity is resolved by casting `x` to the appropriate base interface type. Such casts have no run-time costs—they merely consist of viewing the instance as a less derived type at compile-time.

In the example

```
interface IInteger
{
    void Add(int i);
}

interface IDouble
{
    void Add(double d);
}

interface INumber: IInteger, IDouble {}

class C
{
    void Test(INumber n) {
        n.Add(1);               // Invokes IInteger.Add
        n.Add(1.0);             // Only IDouble.Add is applicable
        ((IInteger)n).Add(1);   // Only IInteger.Add is a candidate
        ((IDouble)n).Add(1);    // Only IDouble.Add is a candidate
    }
}
```

the invocation `n.Add(1)` selects `IInteger.Add` by applying the overload resolution rules of Overload resolution. Similarly the invocation `n.Add(1.0)` selects `IDouble.Add`. When explicit casts are inserted, there is only one candidate method, and thus no ambiguity.

In the example

```
interface IBase
{
    void F(int i);
}

interface ILeft: IBase
{
    new void F(int i);
}

interface IRight: IBase
{
    void G();
}

interface IDerived: ILeft, IRight {}

class A
{
    void Test(IDerived d) {
        d.F(1);                 // Invokes ILeft.F
        ((IBase)d).F(1);        // Invokes IBase.F
        ((ILeft)d).F(1);        // Invokes ILeft.F
        ((IRight)d).F(1);       // Invokes IBase.F
    }
}
```

the `IBase.F` member is hidden by the `ILeft.F` member. The invocation `d.F(1)` thus selects `ILeft.F`, even though `IBase.F` appears to not be hidden in the access path that leads through `IRight`.

The intuitive rule for hiding in multiple-inheritance interfaces is simply this: If a member is hidden in any access path, it is hidden in all access paths. Because the access path from `IDerived` to `ILeft` to `IBase` hides `IBase.F`, the member is also hidden in the access path from `IDerived` to `IRight` to `IBase`.

Fully qualified interface member names

An interface member is sometimes referred to by its ***fully qualified name***. The fully qualified name of an interface member consists of the name of the interface in which the member is declared, followed by a dot, followed by the name of the member. The fully qualified name of a member references the interface in which the member is declared. For example, given the declarations

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}
```

the fully qualified name of `Paint` is `IControl.Paint` and the fully qualified name of `SetText` is `ITextBox.SetText`.

In the example above, it is not possible to refer to `Paint` as `ITextBox.Paint`.

When an interface is part of a namespace, the fully qualified name of an interface member includes the namespace name. For example

```
namespace System
{
    public interface ICloneable
    {
        object Clone();
    }
}
```

Here, the fully qualified name of the `Clone` method is `System.ICloneable.Clone`.

Interface implementations

Interfaces may be implemented by classes and structs. To indicate that a class or struct directly implements an interface, the interface identifier is included in the base class list of the class or struct. For example:

```
interface ICloneable
{
    object Clone();
}

interface IComparable
{
    int CompareTo(object other);
}

class ListEntry: ICloneable, IComparable
{
    public object Clone() {...}
    public int CompareTo(object other) {...}
}
```

A class or struct that directly implements an interface also directly implements all of the interface's base interfaces implicitly. This is true even if the class or struct doesn't explicitly list all base interfaces in the base class list. For example:

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

class TextBox: ITextBox
{
    public void Paint() {...}
    public void SetText(string text) {...}
}
```

Here, class `TextBox` implements both `IControl` and `ITextBox`.

When a class `C` directly implements an interface, all classes derived from C also implement the interface implicitly. The base interfaces specified in a class declaration can be constructed interface types (Constructed types). A base interface cannot be a type parameter on its own, though it can involve the type parameters that are in scope. The following code illustrates how a class can implement and extend constructed types:

```
class C<U,V> {}

interface I1<V> {}

class D: C<string,int>, I1<string> {}

class E<T>: C<int,T>, I1<T> {}
```

The base interfaces of a generic class declaration must satisfy the uniqueness rule described in Uniqueness of implemented interfaces.

Explicit interface member implementations

For purposes of implementing interfaces, a class or struct may declare ***explicit interface member implementations***. An explicit interface member implementation is a method, property, event, or indexer declaration that references a fully qualified interface member name. For example

```
interface IList<T>
{
    T[] GetElements();
}

interface IDictionary<K,V>
{
    V this[K key];
    void Add(K key, V value);
}

class List<T>: IList<T>, IDictionary<int,T>
{
    T[] IList<T>.GetElements() {...}
    T IDictionary<int,T>.this[int index] {...}
    void IDictionary<int,T>.Add(int index, T value) {...}
}
```

Here `IDictionary<int,T>.this` and `IDictionary<int,T>.Add` are explicit interface member implementations.

In some cases, the name of an interface member may not be appropriate for the implementing class, in which case the interface member may be implemented using explicit interface member implementation. A class implementing a file abstraction, for example, would likely implement a `Close` member function that has the effect of releasing the file resource, and implement the `Dispose` method of the `IDisposable` interface using explicit interface member implementation:

```
interface IDisposable
{
    void Dispose();
}

class MyFile: IDisposable
{
    void IDisposable.Dispose() {
        Close();
    }

    public void Close() {
        // Do what's necessary to close the file
        System.GC.SuppressFinalize(this);
    }
}
```

It is not possible to access an explicit interface member implementation through its fully qualified name in a method invocation, property access, or indexer access. An explicit interface member implementation can only be accessed through an interface instance, and is in that case referenced simply by its member name.

It is a compile-time error for an explicit interface member implementation to include access modifiers, and it is a compile-time error to include the modifiers `abstract`, `virtual`, `override`, or `static`.

Explicit interface member implementations have different accessibility characteristics than other members. Because explicit interface member implementations are never accessible through their fully qualified name in a method invocation or a property access, they are in a sense private. However, since they can be accessed through an interface instance, they are in a sense also public.

Explicit interface member implementations serve two primary purposes:

- Because explicit interface member implementations are not accessible through class or struct instances, they allow interface implementations to be excluded from the public interface of a class or struct. This is particularly useful when a class or struct implements an internal interface that is of no interest to a consumer of that class or struct.
- Explicit interface member implementations allow disambiguation of interface members with the same signature. Without explicit interface member implementations it would be impossible for a class or struct to have different implementations of interface members with the same signature and return type, as would it be impossible for a class or struct to have any implementation at all of interface members with the same signature but with different return types.

For an explicit interface member implementation to be valid, the class or struct must name an interface in its base class list that contains a member whose fully qualified name, type, and parameter types exactly match those of the explicit interface member implementation. Thus, in the following class

```
class Shape: ICloneable
{
    object ICloneable.Clone() {...}
    int IComparable.CompareTo(object other) {...}    // invalid
}
```

the declaration of `IComparable.CompareTo` results in a compile-time error because `IComparable` is not listed in the base class list of `Shape` and is not a base interface of `ICloneable`. Likewise, in the declarations

```
class Shape: ICloneable
{
    object ICloneable.Clone() {...}
}

class Ellipse: Shape
{
    object ICloneable.Clone() {...}    // invalid
}
```

the declaration of `ICloneable.Clone` in `Ellipse` results in a compile-time error because `ICloneable` is not explicitly listed in the base class list of `Ellipse`.

The fully qualified name of an interface member must reference the interface in which the member was declared. Thus, in the declarations

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

class TextBox: ITextBox
{
    void IControl.Paint() {...}
    void ITextBox.SetText(string text) {...}
}
```

the explicit interface member implementation of `Paint` must be written as `IControl.Paint`.

Uniqueness of implemented interfaces

The interfaces implemented by a generic type declaration must remain unique for all possible constructed types. Without this rule, it would be impossible to determine the correct method to call for certain constructed types. For example, suppose a generic class declaration were permitted to be written as follows:

```
interface I<T>
{
    void F();
}

class X<U,V>: I<U>, I<V>                    // Error: I<U> and I<V> conflict
{
    void I<U>.F() {...}
    void I<V>.F() {...}
}
```

Were this permitted, it would be impossible to determine which code to execute in the following case:

```
I<int> x = new X<int,int>();
x.F();
```

To determine if the interface list of a generic type declaration is valid, the following steps are performed:

- Let `L` be the list of interfaces directly specified in a generic class, struct, or interface declaration `C`.
- Add to `L` any base interfaces of the interfaces already in `L`.
- Remove any duplicates from `L`.
- If any possible constructed type created from `C` would, after type arguments are substituted into `L`, cause two interfaces in `L` to be identical, then the declaration of `C` is invalid. Constraint declarations are not considered when determining all possible constructed types.

In the class declaration `X` above, the interface list `L` consists of `I<U>` and `I<V>`. The declaration is invalid because any constructed type with `U` and `V` being the same type would cause these two interfaces to be identical types.

It is possible for interfaces specified at different inheritance levels to unify:

```
interface I<T>
{
    void F();
}

class Base<U>: I<U>
{
    void I<U>.F() {...}
}

class Derived<U,V>: Base<U>, I<V>    // Ok
{
    void I<V>.F() {...}
}
```

This code is valid even though `Derived<U,V>` implements both `I<U>` and `I<V>`. The code

```
I<int> x = new Derived<int,int>();
x.F();
```

invokes the method in `Derived`, since `Derived<int,int>` effectively re-implements `I<int>` (Interface re-implementation).

Implementation of generic methods

When a generic method implicitly implements an interface method, the constraints given for each method type parameter must be equivalent in both declarations (after any interface type parameters are replaced with the appropriate type arguments), where method type parameters are identified by ordinal positions, left to right.

When a generic method explicitly implements an interface method, however, no constraints are allowed on the implementing method. Instead, the constraints are inherited from the interface method

```
interface I<A,B,C>
{
    void F<T>(T t) where T: A;
    void G<T>(T t) where T: B;
    void H<T>(T t) where T: C;
}

class C: I<object,C,string>
{
    public void F<T>(T t) {...}                    // Ok
    public void G<T>(T t) where T: C {...}         // Ok
    public void H<T>(T t) where T: string {...}    // Error
}
```

The method `C.F<T>` implicitly implements `I<object,C,string>.F<T>`. In this case, `C.F<T>` is not required (nor permitted) to specify the constraint `T:object` since `object` is an implicit constraint on all type parameters. The method `C.G<T>` implicitly implements `I<object,C,string>.G<T>` because the constraints match those in the interface, after the interface type parameters are replaced with the corresponding type arguments. The constraint for method `C.H<T>` is an error because sealed types (`string` in this case) cannot be used as constraints. Omitting the constraint would also be an error since constraints of implicit interface method implementations are required to match. Thus, it is impossible to implicitly implement `I<object,C,string>.H<T>`. This interface method can only be implemented using an explicit interface member implementation:

```
class C: I<object,C,string>
{
    ...

    public void H<U>(U u) where U: class {...}

    void I<object,C,string>.H<T>(T t) {
        string s = t;    // Ok
        H<T>(t);
    }
}
```

In this example, the explicit interface member implementation invokes a public method having strictly weaker constraints. Note that the assignment from `t` to `s` is valid since `T` inherits a constraint of `T:string`, even though this constraint is not expressible in source code.

Interface mapping

A class or struct must provide implementations of all members of the interfaces that are listed in the base class list of the class or struct. The process of locating implementations of interface members in an implementing class or struct is known as *interface mapping*.

Interface mapping for a class or struct `C` locates an implementation for each member of each interface specified in the base class list of `C`. The implementation of a particular interface member `I.M`, where `I` is the interface in which the member `M` is declared, is determined by examining each class or struct `S`, starting with `C` and repeating for each successive base class of `C`, until a match is located:

- If `S` contains a declaration of an explicit interface member implementation that matches `I` and `M`, then this member is the implementation of `I.M`.
- Otherwise, if `S` contains a declaration of a non-static public member that matches `M`, then this member is the implementation of `I.M`. If more than one member matches, it is unspecified which member is the implementation of `I.M`. This situation can only occur if `S` is a constructed type where the two members as declared in the generic type have different signatures, but the type arguments make their signatures identical.

A compile-time error occurs if implementations cannot be located for all members of all interfaces specified in the base class list of `C`. Note that the members of an interface include those members that are inherited from base interfaces.

For purposes of interface mapping, a class member `A` matches an interface member `B` when:

- `A` and `B` are methods, and the name, type, and formal parameter lists of `A` and `B` are identical.
- `A` and `B` are properties, the name and type of `A` and `B` are identical, and `A` has the same accessors as `B` (`A` is permitted to have additional accessors if it is not an explicit interface member implementation).
- `A` and `B` are events, and the name and type of `A` and `B` are identical.
- `A` and `B` are indexers, the type and formal parameter lists of `A` and `B` are identical, and `A` has the same accessors as `B` (`A` is permitted to have additional accessors if it is not an explicit interface member implementation).

Notable implications of the interface mapping algorithm are:

- Explicit interface member implementations take precedence over other members in the same class or struct when determining the class or struct member that implements an interface member.
- Neither non-public nor static members participate in interface mapping.

In the example

```
interface ICloneable
{
    object Clone();
}

class C: ICloneable
{
    object ICloneable.Clone() {...}
    public object Clone() {...}
}
```

the `ICloneable.Clone` member of `C` becomes the implementation of `Clone` in `ICloneable` because explicit interface member implementations take precedence over other members.

If a class or struct implements two or more interfaces containing a member with the same name, type, and parameter types, it is possible to map each of those interface members onto a single class or struct member. For example

```
interface IControl
{
    void Paint();
}

interface IForm
{
    void Paint();
}

class Page: IControl, IForm
{
    public void Paint() {...}
}
```

Here, the `Paint` methods of both `IControl` and `IForm` are mapped onto the `Paint` method in `Page`. It is of course also possible to have separate explicit interface member implementations for the two methods.

If a class or struct implements an interface that contains hidden members, then some members must necessarily be implemented through explicit interface member implementations. For example

```
interface IBase
{
    int P { get; }
}

interface IDerived: IBase
{
    new int P();
}
```

An implementation of this interface would require at least one explicit interface member implementation, and would take one of the following forms

```
class C: IDerived
{
    int IBase.P { get {...} }
    int IDerived.P() {...}
}

class C: IDerived
{
    public int P { get {...} }
    int IDerived.P() {...}
}

class C: IDerived
{
    int IBase.P { get {...} }
    public int P() {...}
}
```

When a class implements multiple interfaces that have the same base interface, there can be only one implementation of the base interface. In the example

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

interface IListBox: IControl
{
    void SetItems(string[] items);
}

class ComboBox: IControl, ITextBox, IListBox
{
    void IControl.Paint() {...}
    void ITextBox.SetText(string text) {...}
    void IListBox.SetItems(string[] items) {...}
}
```

it is not possible to have separate implementations for the `IControl` named in the base class list, the `IControl` inherited by `ITextBox`, and the `IControl` inherited by `IListBox`. Indeed, there is no notion of a separate

identity for these interfaces. Rather, the implementations of `ITextBox` and `IListBox` share the same implementation of `IControl`, and `ComboBox` is simply considered to implement three interfaces, `IControl`, `ITextBox`, and `IListBox`.

The members of a base class participate in interface mapping. In the example

```
interface Interface1
{
    void F();
}

class Class1
{
    public void F() {}
    public void G() {}
}

class Class2: Class1, Interface1
{
    new public void G() {}
}
```

the method `F` in `Class1` is used in `Class2`'s implementation of `Interface1`.

Interface implementation inheritance

A class inherits all interface implementations provided by its base classes.

Without explicitly ***re-implementing*** an interface, a derived class cannot in any way alter the interface mappings it inherits from its base classes. For example, in the declarations

```
interface IControl
{
    void Paint();
}

class Control: IControl
{
    public void Paint() {...}
}

class TextBox: Control
{
    new public void Paint() {...}
}
```

the `Paint` method in `TextBox` hides the `Paint` method in `Control`, but it does not alter the mapping of `Control.Paint` onto `IControl.Paint`, and calls to `Paint` through class instances and interface instances will have the following effects

```
Control c = new Control();
TextBox t = new TextBox();
IControl ic = c;
IControl it = t;
c.Paint();            // invokes Control.Paint();
t.Paint();            // invokes TextBox.Paint();
ic.Paint();           // invokes Control.Paint();
it.Paint();           // invokes Control.Paint();
```

However, when an interface method is mapped onto a virtual method in a class, it is possible for derived classes to override the virtual method and alter the implementation of the interface. For example, rewriting the

declarations above to

```
interface IControl
{
    void Paint();
}

class Control: IControl
{
    public virtual void Paint() {...}
}

class TextBox: Control
{
    public override void Paint() {...}
}
```

the following effects will now be observed

```
Control c = new Control();
TextBox t = new TextBox();
IControl ic = c;
IControl it = t;
c.Paint();            // invokes Control.Paint();
t.Paint();            // invokes TextBox.Paint();
ic.Paint();           // invokes Control.Paint();
it.Paint();           // invokes TextBox.Paint();
```

Since explicit interface member implementations cannot be declared virtual, it is not possible to override an explicit interface member implementation. However, it is perfectly valid for an explicit interface member implementation to call another method, and that other method can be declared virtual to allow derived classes to override it. For example

```
interface IControl
{
    void Paint();
}

class Control: IControl
{
    void IControl.Paint() { PaintControl(); }
    protected virtual void PaintControl() {...}
}

class TextBox: Control
{
    protected override void PaintControl() {...}
}
```

Here, classes derived from `Control` can specialize the implementation of `IControl.Paint` by overriding the `PaintControl` method.

Interface re-implementation

A class that inherits an interface implementation is permitted to ***re-implement*** the interface by including it in the base class list.

A re-implementation of an interface follows exactly the same interface mapping rules as an initial implementation of an interface. Thus, the inherited interface mapping has no effect whatsoever on the interface mapping established for the re-implementation of the interface. For example, in the declarations

```
interface IControl
{
    void Paint();
}

class Control: IControl
{
    void IControl.Paint() {...}
}

class MyControl: Control, IControl
{
    public void Paint() {}
}
```

the fact that `Control` maps `IControl.Paint` onto `Control.IControl.Paint` doesn't affect the re-implementation in `MyControl`, which maps `IControl.Paint` onto `MyControl.Paint`.

Inherited public member declarations and inherited explicit interface member declarations participate in the interface mapping process for re-implemented interfaces. For example

```
interface IMethods
{
    void F();
    void G();
    void H();
    void I();
}

class Base: IMethods
{
    void IMethods.F() {}
    void IMethods.G() {}
    public void H() {}
    public void I() {}
}

class Derived: Base, IMethods
{
    public void F() {}
    void IMethods.H() {}
}
```

Here, the implementation of `IMethods` in `Derived` maps the interface methods onto `Derived.F`, `Base.IMethods.G`, `Derived.IMethods.H`, and `Base.I`.

When a class implements an interface, it implicitly also implements all of that interface's base interfaces. Likewise, a re-implementation of an interface is also implicitly a re-implementation of all of the interface's base interfaces. For example

```
interface IBase
{
    void F();
}

interface IDerived: IBase
{
    void G();
}

class C: IDerived
{
    void IBase.F() {...}
    void IDerived.G() {...}
}

class D: C, IDerived
{
    public void F() {...}
    public void G() {...}
}
```

Here, the re-implementation of `IDerived` also re-implements `IBase`, mapping `IBase.F` onto `D.F`.

Abstract classes and interfaces

Like a non-abstract class, an abstract class must provide implementations of all members of the interfaces that are listed in the base class list of the class. However, an abstract class is permitted to map interface methods onto abstract methods. For example

```
interface IMethods
{
    void F();
    void G();
}

abstract class C: IMethods
{
    public abstract void F();
    public abstract void G();
}
```

Here, the implementation of `IMethods` maps `F` and `G` onto abstract methods, which must be overridden in non-abstract classes that derive from `C`.

Note that explicit interface member implementations cannot be abstract, but explicit interface member implementations are of course permitted to call abstract methods. For example

```
interface IMethods
{
    void F();
    void G();
}

abstract class C: IMethods
{
    void IMethods.F() { FF(); }
    void IMethods.G() { GG(); }
    protected abstract void FF();
    protected abstract void GG();
}
```

Here, non-abstract classes that derive from `C` would be required to override `FF` and `GG`, thus providing the actual implementation of `IMethods`.

Enums

18.12.2021 • 6 minutes to read

An ***enum type*** is a distinct value type (Value types) that declares a set of named constants.

The example

```
enum Color
{
    Red,
    Green,
    Blue
}
```

declares an enum type named `Color` with members `Red`, `Green`, and `Blue`.

Enum declarations

An enum declaration declares a new enum type. An enum declaration begins with the keyword `enum`, and defines the name, accessibility, underlying type, and members of the enum.

```
enum_declaration
    : attributes? enum_modifier* 'enum' identifier enum_base? enum_body ';'?
    ;

enum_base
    : ':' integral_type
    ;

enum_body
    : '{' enum_member_declarations? '}'
    | '{' enum_member_declarations ',' '}'
    ;
```

Each enum type has a corresponding integral type called the ***underlying type*** of the enum type. This underlying type must be able to represent all the enumerator values defined in the enumeration. An enum declaration may explicitly declare an underlying type of `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` or `ulong`. Note that `char` cannot be used as an underlying type. An enum declaration that does not explicitly declare an underlying type has an underlying type of `int`.

The example

```
enum Color: long
{
    Red,
    Green,
    Blue
}
```

declares an enum with an underlying type of `long`. A developer might choose to use an underlying type of `long`, as in the example, to enable the use of values that are in the range of `long` but not in the range of `int`, or to preserve this option for the future.

Enum modifiers

An *enum_declaration* may optionally include a sequence of enum modifiers:

```
enum_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    ;
```

It is a compile-time error for the same modifier to appear multiple times in an enum declaration.

The modifiers of an enum declaration have the same meaning as those of a class declaration (Class modifiers). Note, however, that the `abstract` and `sealed` modifiers are not permitted in an enum declaration. Enums cannot be abstract and do not permit derivation.

Enum members

The body of an enum type declaration defines zero or more enum members, which are the named constants of the enum type. No two enum members can have the same name.

```
enum_member_declarations
    : enum_member_declaration (',' enum_member_declaration)*
    ;

enum_member_declaration
    : attributes? identifier ('=' constant_expression)?
    ;
```

Each enum member has an associated constant value. The type of this value is the underlying type for the containing enum. The constant value for each enum member must be in the range of the underlying type for the enum. The example

```
enum Color: uint
{
    Red = -1,
    Green = -2,
    Blue = -3
}
```

results in a compile-time error because the constant values `-1`, `-2`, and `-3` are not in the range of the underlying integral type `uint`.

Multiple enum members may share the same associated value. The example

```
enum Color
{
    Red,
    Green,
    Blue,

    Max = Blue
}
```

shows an enum in which two enum members -- `Blue` and `Max` -- have the same associated value.

The associated value of an enum member is assigned either implicitly or explicitly. If the declaration of the enum member has a *constant_expression* initializer, the value of that constant expression, implicitly converted to the underlying type of the enum, is the associated value of the enum member. If the declaration of the enum member has no initializer, its associated value is set implicitly, as follows:

- If the enum member is the first enum member declared in the enum type, its associated value is zero.
- Otherwise, the associated value of the enum member is obtained by increasing the associated value of the textually preceding enum member by one. This increased value must be within the range of values that can be represented by the underlying type, otherwise a compile-time error occurs.

The example

```
using System;

enum Color
{
    Red,
    Green = 10,
    Blue
}

class Test
{
    static void Main() {
        Console.WriteLine(StringFromColor(Color.Red));
        Console.WriteLine(StringFromColor(Color.Green));
        Console.WriteLine(StringFromColor(Color.Blue));
    }

    static string StringFromColor(Color c) {
        switch (c) {
            case Color.Red:
                return String.Format("Red = {0}", (int) c);

            case Color.Green:
                return String.Format("Green = {0}", (int) c);

            case Color.Blue:
                return String.Format("Blue = {0}", (int) c);

            default:
                return "Invalid color";
        }
    }
}
```

prints out the enum member names and their associated values. The output is:

```
Red = 0
Green = 10
Blue = 11
```

for the following reasons:

- the enum member `Red` is automatically assigned the value zero (since it has no initializer and is the first enum member);
- the enum member `Green` is explicitly given the value `10`;
- and the enum member `Blue` is automatically assigned the value one greater than the member that textually precedes it.

The associated value of an enum member may not, directly or indirectly, use the value of its own associated enum member. Other than this circularity restriction, enum member initializers may freely refer to other enum member initializers, regardless of their textual position. Within an enum member initializer, values of other enum members are always treated as having the type of their underlying type, so that casts are not necessary when referring to other enum members.

The example

```
enum Circular
{
    A = B,
    B
}
```

results in a compile-time error because the declarations of `A` and `B` are circular. `A` depends on `B` explicitly, and `B` depends on `A` implicitly.

Enum members are named and scoped in a manner exactly analogous to fields within classes. The scope of an enum member is the body of its containing enum type. Within that scope, enum members can be referred to by their simple name. From all other code, the name of an enum member must be qualified with the name of its enum type. Enum members do not have any declared accessibility -- an enum member is accessible if its containing enum type is accessible.

The System.Enum type

The type `System.Enum` is the abstract base class of all enum types (this is distinct and different from the underlying type of the enum type), and the members inherited from `System.Enum` are available in any enum type. A boxing conversion (Boxing conversions) exists from any enum type to `System.Enum`, and an unboxing conversion (Unboxing conversions) exists from `System.Enum` to any enum type.

Note that `System.Enum` is not itself an *enum_type*. Rather, it is a *class_type* from which all *enum_type*s are derived. The type `System.Enum` inherits from the type `System.ValueType` (The System.ValueType type), which, in turn, inherits from type `object`. At run-time, a value of type `System.Enum` can be `null` or a reference to a boxed value of any enum type.

Enum values and operations

Each enum type defines a distinct type; an explicit enumeration conversion (Explicit enumeration conversions) is required to convert between an enum type and an integral type, or between two enum types. The set of values that an enum type can take on is not limited by its enum members. In particular, any value of the underlying type of an enum can be cast to the enum type, and is a distinct valid value of that enum type.

Enum members have the type of their containing enum type (except within other enum member initializers: see Enum members). The value of an enum member declared in enum type `E` with associated value `v` is `(E)v`.

The following operators can be used on values of enum types: `==`, `!=`, `<`, `>`, `<=`, `>=` (Enumeration comparison operators), binary `+` (Addition operator), binary `-` (Subtraction operator), `^`, `&`, `|` (Enumeration logical operators), `~` (Bitwise complement operator), `++` and `--` (Postfix increment and decrement operators and Prefix increment and decrement operators).

Every enum type automatically derives from the class `System.Enum` (which, in turn, derives from `System.ValueType` and `object`). Thus, inherited methods and properties of this class can be used on values of an enum type.

Delegates

18.12.2021 • 9 minutes to read

Delegates enable scenarios that other languages—such as C++, Pascal, and Modula -- have addressed with function pointers. Unlike C++ function pointers, however, delegates are fully object oriented, and unlike C++ pointers to member functions, delegates encapsulate both an object instance and a method.

A delegate declaration defines a class that is derived from the class `System.Delegate`. A delegate instance encapsulates an invocation list, which is a list of one or more methods, each of which is referred to as a callable entity. For instance methods, a callable entity consists of an instance and a method on that instance. For static methods, a callable entity consists of just a method. Invoking a delegate instance with an appropriate set of arguments causes each of the delegate's callable entities to be invoked with the given set of arguments.

An interesting and useful property of a delegate instance is that it does not know or care about the classes of the methods it encapsulates; all that matters is that those methods be compatible (Delegate declarations) with the delegate's type. This makes delegates perfectly suited for "anonymous" invocation.

Delegate declarations

A *delegate_declaration* is a *type_declaration* (Type declarations) that declares a new delegate type.

```
delegate_declaration
    : attributes? delegate_modifier* 'delegate' return_type
      identifier variant_type_parameter_list?
      '(' formal_parameter_list? ')' type_parameter_constraints_clause* ';'
    ;

delegate_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | delegate_modifier_unsafe
    ;
```

It is a compile-time error for the same modifier to appear multiple times in a delegate declaration.

The `new` modifier is only permitted on delegates declared within another type, in which case it specifies that such a delegate hides an inherited member by the same name, as described in The new modifier.

The `public`, `protected`, `internal`, and `private` modifiers control the accessibility of the delegate type. Depending on the context in which the delegate declaration occurs, some of these modifiers may not be permitted (Declared accessibility).

The delegate's type name is *identifier*.

The optional *formal_parameter_list* specifies the parameters of the delegate, and *return_type* indicates the return type of the delegate.

The optional *variant_type_parameter_list* (Variant type parameter lists) specifies the type parameters to the delegate itself.

The return type of a delegate type must be either `void`, or output-safe (Variance safety).

All the formal parameter types of a delegate type must be input-safe. Additionally, any `out` or `ref` parameter types must also be output-safe. Note that even `out` parameters are required to be input-safe, due to a limitation of the underlying execution platform.

Delegate types in C# are name equivalent, not structurally equivalent. Specifically, two different delegate types that have the same parameter lists and return type are considered different delegate types. However, instances of two distinct but structurally equivalent delegate types may compare as equal (Delegate equality operators).

For example:

```
delegate int D1(int i, double d);

class A
{
    public static int M1(int a, double b) {...}
}

class B
{
    delegate int D2(int c, double d);
    public static int M1(int f, double g) {...}
    public static void M2(int k, double l) {...}
    public static int M3(int g) {...}
    public static void M4(int g) {...}
}
```

The methods `A.M1` and `B.M1` are compatible with both the delegate types `D1` and `D2`, since they have the same return type and parameter list; however, these delegate types are two different types, so they are not interchangeable. The methods `B.M2`, `B.M3`, and `B.M4` are incompatible with the delegate types `D1` and `D2`, since they have different return types or parameter lists.

Like other generic type declarations, type arguments must be given to create a constructed delegate type. The parameter types and return type of a constructed delegate type are created by substituting, for each type parameter in the delegate declaration, the corresponding type argument of the constructed delegate type. The resulting return type and parameter types are used in determining what methods are compatible with a constructed delegate type. For example:

```
delegate bool Predicate<T>(T value);

class X
{
    static bool F(int i) {...}
    static bool G(string s) {...}
}
```

The method `X.F` is compatible with the delegate type `Predicate<int>` and the method `X.G` is compatible with the delegate type `Predicate<string>`.

The only way to declare a delegate type is via a *delegate_declaration*. A delegate type is a class type that is derived from `System.Delegate`. Delegate types are implicitly `sealed`, so it is not permissible to derive any type from a delegate type. It is also not permissible to derive a non-delegate class type from `System.Delegate`. Note that `System.Delegate` is not itself a delegate type; it is a class type from which all delegate types are derived.

C# provides special syntax for delegate instantiation and invocation. Except for instantiation, any operation that can be applied to a class or class instance can also be applied to a delegate class or instance, respectively. In particular, it is possible to access members of the `System.Delegate` type via the usual member access syntax.

The set of methods encapsulated by a delegate instance is called an invocation list. When a delegate instance is

created (Delegate compatibility) from a single method, it encapsulates that method, and its invocation list contains only one entry. However, when two non-null delegate instances are combined, their invocation lists are concatenated -- in the order left operand then right operand -- to form a new invocation list, which contains two or more entries.

Delegates are combined using the binary `+` (Addition operator) and `+=` operators (Compound assignment). A delegate can be removed from a combination of delegates, using the binary `-` (Subtraction operator) and `-=` operators (Compound assignment). Delegates can be compared for equality (Delegate equality operators).

The following example shows the instantiation of a number of delegates, and their corresponding invocation lists:

```
delegate void D(int x);

class C
{
    public static void M1(int i) {...}
    public static void M2(int i) {...}

}

class Test
{
    static void Main() {
        D cd1 = new D(C.M1);      // M1
        D cd2 = new D(C.M2);      // M2
        D cd3 = cd1 + cd2;        // M1 + M2
        D cd4 = cd3 + cd1;        // M1 + M2 + M1
        D cd5 = cd4 + cd3;        // M1 + M2 + M1 + M1 + M2
    }

}
```

When `cd1` and `cd2` are instantiated, they each encapsulate one method. When `cd3` is instantiated, it has an invocation list of two methods, `M1` and `M2`, in that order. `cd4`'s invocation list contains `M1`, `M2`, and `M1`, in that order. Finally, `cd5`'s invocation list contains `M1`, `M2`, `M1`, `M1`, and `M2`, in that order. For more examples of combining (as well as removing) delegates, see Delegate invocation.

Delegate compatibility

A method or delegate `M` is ***compatible*** with a delegate type `D` if all of the following are true:

- `D` and `M` have the same number of parameters, and each parameter in `D` has the same `ref` or `out` modifiers as the corresponding parameter in `M`.
- For each value parameter (a parameter with no `ref` or `out` modifier), an identity conversion (Identity conversion) or implicit reference conversion (Implicit reference conversions) exists from the parameter type in `D` to the corresponding parameter type in `M`.
- For each `ref` or `out` parameter, the parameter type in `D` is the same as the parameter type in `M`.
- An identity or implicit reference conversion exists from the return type of `M` to the return type of `D`.

Delegate instantiation

An instance of a delegate is created by a *delegate_creation_expression* (Delegate creation expressions) or a conversion to a delegate type. The newly created delegate instance then refers to either:

- The static method referenced in the *delegate_creation_expression*, or
- The target object (which cannot be `null`) and instance method referenced in the

delegate_creation_expression, or

- Another delegate.

For example:

```
delegate void D(int x);

class C
{
    public static void M1(int i) {...}
    public void M2(int i) {...}
}

class Test
{
    static void Main() {
        D cd1 = new D(C.M1);        // static method
        C t = new C();
        D cd2 = new D(t.M2);        // instance method
        D cd3 = new D(cd2);        // another delegate
    }
}
```

Once instantiated, delegate instances always refer to the same target object and method. Remember, when two delegates are combined, or one is removed from another, a new delegate results with its own invocation list; the invocation lists of the delegates combined or removed remain unchanged.

Delegate invocation

C# provides special syntax for invoking a delegate. When a non-null delegate instance whose invocation list contains one entry is invoked, it invokes the one method with the same arguments it was given, and returns the same value as the referred to method. (See Delegate invocations for detailed information on delegate invocation.) If an exception occurs during the invocation of such a delegate, and that exception is not caught within the method that was invoked, the search for an exception catch clause continues in the method that called the delegate, as if that method had directly called the method to which that delegate referred.

Invocation of a delegate instance whose invocation list contains multiple entries proceeds by invoking each of the methods in the invocation list, synchronously, in order. Each method so called is passed the same set of arguments as was given to the delegate instance. If such a delegate invocation includes reference parameters (Reference parameters), each method invocation will occur with a reference to the same variable; changes to that variable by one method in the invocation list will be visible to methods further down the invocation list. If the delegate invocation includes output parameters or a return value, their final value will come from the invocation of the last delegate in the list.

If an exception occurs during processing of the invocation of such a delegate, and that exception is not caught within the method that was invoked, the search for an exception catch clause continues in the method that called the delegate, and any methods further down the invocation list are not invoked.

Attempting to invoke a delegate instance whose value is null results in an exception of type `System.NullReferenceException`.

The following example shows how to instantiate, combine, remove, and invoke delegates:

```
using System;

delegate void D(int x);

class C
{
    public static void M1(int i) {
        Console.WriteLine("C.M1: " + i);
    }

    public static void M2(int i) {
        Console.WriteLine("C.M2: " + i);
    }

    public void M3(int i) {
        Console.WriteLine("C.M3: " + i);
    }
}

class Test
{
    static void Main() {
        D cd1 = new D(C.M1);
        cd1(-1);                // call M1

        D cd2 = new D(C.M2);
        cd2(-2);                // call M2

        D cd3 = cd1 + cd2;
        cd3(10);                // call M1 then M2

        cd3 += cd1;
        cd3(20);                // call M1, M2, then M1

        C c = new C();
        D cd4 = new D(c.M3);
        cd3 += cd4;
        cd3(30);                // call M1, M2, M1, then M3

        cd3 -= cd1;             // remove last M1
        cd3(40);                // call M1, M2, then M3

        cd3 -= cd4;
        cd3(50);                // call M1 then M2

        cd3 -= cd2;
        cd3(60);                // call M1

        cd3 -= cd2;             // impossible removal is benign
        cd3(60);                // call M1

        cd3 -= cd1;             // invocation list is empty so cd3 is null

        cd3(70);                // System.NullReferenceException thrown

        cd3 -= cd1;             // impossible removal is benign
    }
}
```

As shown in the statement `cd3 += cd1;`, a delegate can be present in an invocation list multiple times. In this case, it is simply invoked once per occurrence. In an invocation list such as this, when that delegate is removed, the last occurrence in the invocation list is the one actually removed.

Immediately prior to the execution of the final statement, `cd3 -= cd1;`, the delegate `cd3` refers to an empty invocation list. Attempting to remove a delegate from an empty list (or to remove a non-existent delegate from

a non-empty list) is not an error.

The output produced is:

```
C.M1: -1
C.M2: -2
C.M1: 10
C.M2: 10
C.M1: 20
C.M2: 20
C.M1: 20
C.M1: 30
C.M2: 30
C.M1: 30
C.M3: 30
C.M1: 40
C.M2: 40
C.M3: 40
C.M1: 50
C.M2: 50
C.M1: 60
C.M1: 60
```

Exceptions

18.12.2021 • 4 minutes to read

Exceptions in C# provide a structured, uniform, and type-safe way of handling both system level and application level error conditions. The exception mechanism in C# is quite similar to that of C++, with a few important differences:

- In C#, all exceptions must be represented by an instance of a class type derived from `System.Exception`. In C++, any value of any type can be used to represent an exception.
- In C#, a finally block (The try statement) can be used to write termination code that executes in both normal execution and exceptional conditions. Such code is difficult to write in C++ without duplicating code.
- In C#, system-level exceptions such as overflow, divide-by-zero, and null dereferences have well defined exception classes and are on a par with application-level error conditions.

Causes of exceptions

Exception can be thrown in two different ways.

- A `throw` statement (The throw statement) throws an exception immediately and unconditionally. Control never reaches the statement immediately following the `throw`.
- Certain exceptional conditions that arise during the processing of C# statements and expression cause an exception in certain circumstances when the operation cannot be completed normally. For example, an integer division operation (Division operator) throws a `System.DivideByZeroException` if the denominator is zero. See Common Exception Classes for a list of the various exceptions that can occur in this way.

The System.Exception class

The `System.Exception` class is the base type of all exceptions. This class has a few notable properties that all exceptions share:

- `Message` is a read-only property of type `string` that contains a human-readable description of the reason for the exception.
- `InnerException` is a read-only property of type `Exception`. If its value is non-null, it refers to the exception that caused the current exception—that is, the current exception was raised in a catch block handling the `InnerException`. Otherwise, its value is null, indicating that this exception was not caused by another exception. The number of exception objects chained together in this manner can be arbitrary.

The value of these properties can be specified in calls to the instance constructor for `System.Exception`.

How exceptions are handled

Exceptions are handled by a `try` statement (The try statement).

When an exception occurs, the system searches for the nearest `catch` clause that can handle the exception, as determined by the run-time type of the exception. First, the current method is searched for a lexically enclosing `try` statement, and the associated catch clauses of the try statement are considered in order. If that fails, the method that called the current method is searched for a lexically enclosing `try` statement that encloses the point of the call to the current method. This search continues until a `catch` clause is found that can handle the current exception, by naming an exception class that is of the same class, or a base class, of the run-time type of the exception being thrown. A `catch` clause that doesn't name an exception class can handle any exception.

Once a matching catch clause is found, the system prepares to transfer control to the first statement of the catch clause. Before execution of the catch clause begins, the system first executes, in order, any `finally` clauses that were associated with try statements more nested that than the one that caught the exception.

If no matching catch clause is found, one of two things occurs:

- If the search for a matching catch clause reaches a static constructor (Static constructors) or static field initializer, then a `System.TypeInitializationException` is thrown at the point that triggered the invocation of the static constructor. The inner exception of the `System.TypeInitializationException` contains the exception that was originally thrown.
- If the search for matching catch clauses reaches the code that initially started the thread, then execution of the thread is terminated. The impact of such termination is implementation-defined.

Exceptions that occur during destructor execution are worth special mention. If an exception occurs during destructor execution, and that exception is not caught, then the execution of that destructor is terminated and the destructor of the base class (if any) is called. If there is no base class (as in the case of the `object` type) or if there is no base class destructor, then the exception is discarded.

Common Exception Classes

The following exceptions are thrown by certain C# operations.

EXCEPTION TYPE	DESCRIPTION
`System.ArithmeticException`	A base class for exceptions that occur during arithmetic operations, such as `System.DivideByZeroException` and `System.OverflowException`.
`System.ArrayTypeMismatchException`	Thrown when a store into an array fails because the actual type of the stored element is incompatible with the actual type of the array.
`System.DivideByZeroException`	Thrown when an attempt to divide an integral value by zero occurs.
`System.IndexOutOfRangeException`	Thrown when an attempt to index an array via an index that is less than zero or outside the bounds of the array.
`System.InvalidCastException`	Thrown when an explicit conversion from a base type or interface to a derived type fails at run time.
`System.NullReferenceException`	Thrown when a `null` reference is used in a way that causes the referenced object to be required.
`System.OutOfMemoryException`	Thrown when an attempt to allocate memory (via `new`) fails.
`System.OverflowException`	Thrown when an arithmetic operation in a `checked` context overflows.
`System.StackOverflowException`	Thrown when the execution stack is exhausted by having too many pending method calls; typically indicative of very deep or unbounded recursion.

EXCEPTION TYPE	DESCRIPTION
`System.TypeInitializationException`	Thrown when a static constructor throws an exception, and no `catch` clauses exists to catch it.

Attributes

18.12.2021 • 25 minutes to read

Much of the C# language enables the programmer to specify declarative information about the entities defined in the program. For example, the accessibility of a method in a class is specified by decorating it with the *method_modifier*s `public`, `protected`, `internal`, and `private`.

C# enables programmers to invent new kinds of declarative information, called ***attributes***. Programmers can then attach attributes to various program entities, and retrieve attribute information in a run-time environment. For instance, a framework might define a `HelpAttribute` attribute that can be placed on certain program elements (such as classes and methods) to provide a mapping from those program elements to their documentation.

Attributes are defined through the declaration of attribute classes (Attribute classes), which may have positional and named parameters (Positional and named parameters). Attributes are attached to entities in a C# program using attribute specifications (Attribute specification), and can be retrieved at run-time as attribute instances (Attribute instances).

Attribute classes

A class that derives from the abstract class `System.Attribute`, whether directly or indirectly, is an ***attribute class***. The declaration of an attribute class defines a new kind of ***attribute*** that can be placed on a declaration. By convention, attribute classes are named with a suffix of `Attribute`. Uses of an attribute may either include or omit this suffix.

Attribute usage

The attribute `AttributeUsage` (The AttributeUsage attribute) is used to describe how an attribute class can be used.

`AttributeUsage` has a positional parameter (Positional and named parameters) that enables an attribute class to specify the kinds of declarations on which it can be used. The example

```
using System;

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
public class SimpleAttribute: Attribute
{
    ...
}
```

defines an attribute class named `SimpleAttribute` that can be placed on *class_declaration*s and *interface_declaration*s only. The example

```
[Simple] class Class1 {...}

[Simple] interface Interface1 {...}
```

shows several uses of the `Simple` attribute. Although this attribute is defined with the name `SimpleAttribute`, when this attribute is used, the `Attribute` suffix may be omitted, resulting in the short name `Simple`. Thus, the example above is semantically equivalent to the following:

```
[SimpleAttribute] class Class1 {...}

[SimpleAttribute] interface Interface1 {...}
```

`AttributeUsage` has a named parameter (Positional and named parameters) called `AllowMultiple`, which indicates whether the attribute can be specified more than once for a given entity. If `AllowMultiple` for an attribute class is true, then that attribute class is a ***multi-use attribute class***, and can be specified more than once on an entity. If `AllowMultiple` for an attribute class is false or it is unspecified, then that attribute class is a ***single-use attribute class***, and can be specified at most once on an entity.

The example

```
using System;

[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
public class AuthorAttribute: Attribute
{
    private string name;

    public AuthorAttribute(string name) {
        this.name = name;
    }

    public string Name {
        get { return name; }
    }
}
```

defines a multi-use attribute class named `AuthorAttribute`. The example

```
[Author("Brian Kernighan"), Author("Dennis Ritchie")]
class Class1
{
    ...
}
```

shows a class declaration with two uses of the `Author` attribute.

`AttributeUsage` has another named parameter called `Inherited`, which indicates whether the attribute, when specified on a base class, is also inherited by classes that derive from that base class. If `Inherited` for an attribute class is true, then that attribute is inherited. If `Inherited` for an attribute class is false then that attribute is not inherited. If it is unspecified, its default value is true.

An attribute class `X` not having an `AttributeUsage` attribute attached to it, as in

```
using System;

class X: Attribute {...}
```

is equivalent to the following:

```
using System;

[AttributeUsage(
    AttributeTargets.All,
    AllowMultiple = false,
    Inherited = true)
]
class X: Attribute {...}
```

Positional and named parameters

Attribute classes can have ***positional parameters*** and ***named parameters***. Each public instance constructor for an attribute class defines a valid sequence of positional parameters for that attribute class. Each non-static public read-write field and property for an attribute class defines a named parameter for the attribute class.

The example

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class HelpAttribute: Attribute
{
    public HelpAttribute(string url) {        // Positional parameter
        ...
    }

    public string Topic {                     // Named parameter
        get {...}
        set {...}
    }

    public string Url {
        get {...}
    }
}
```

defines an attribute class named `HelpAttribute` that has one positional parameter, `url`, and one named parameter, `Topic`. Although it is non-static and public, the property `Url` does not define a named parameter, since it is not read-write.

This attribute class might be used as follows:

```
[Help("http://www.mycompany.com/.../Class1.htm")]
class Class1
{
    ...
}

[Help("http://www.mycompany.com/.../Misc.htm", Topic = "Class2")]
class Class2
{
    ...
}
```

Attribute parameter types

The types of positional and named parameters for an attribute class are limited to the ***attribute parameter types***, which are:

- One of the following types: `bool`, `byte`, `char`, `double`, `float`, `int`, `long`, `sbyte`, `short`, `string`, `uint`, `ulong`, `ushort`.

- The type `object`.
- The type `System.Type`.
- An enum type, provided it has public accessibility and the types in which it is nested (if any) also have public accessibility (Attribute specification).
- Single-dimensional arrays of the above types.
- A constructor argument or public field which does not have one of these types, cannot be used as a positional or named parameter in an attribute specification.

Attribute specification

Attribute specification is the application of a previously defined attribute to a declaration. An attribute is a piece of additional declarative information that is specified for a declaration. Attributes can be specified at global scope (to specify attributes on the containing assembly or module) and for *type_declaration*s (Type declarations), *class_member_declaration*s (Type parameter constraints), *interface_member_declaration*s (Interface members), *struct_member_declaration*s (Struct members), *enum_member_declaration*s (Enum members), *accessor_declarations* (Accessors), *event_accessor_declarations* (Field-like events), and *formal_parameter_list*s (Method parameters).

Attributes are specified in ***attribute sections***. An attribute section consists of a pair of square brackets, which surround a comma-separated list of one or more attributes. The order in which attributes are specified in such a list, and the order in which sections attached to the same program entity are arranged, is not significant. For instance, the attribute specifications `[A][B]`, `[B][A]`, `[A,B]`, and `[B,A]` are equivalent.

```
global_attributes
    : global_attribute_section+
    ;

global_attribute_section
    : '[' global_attribute_target_specifier attribute_list ']'
    | '[' global_attribute_target_specifier attribute_list ',' ']'
    ;

global_attribute_target_specifier
    : global_attribute_target ':'
    ;

global_attribute_target
    : 'assembly'
    | 'module'
    ;

attributes
    : attribute_section+
    ;

attribute_section
    : '[' attribute_target_specifier? attribute_list ']'
    | '[' attribute_target_specifier? attribute_list ',' ']'
    ;

attribute_target_specifier
    : attribute_target ':'
    ;

attribute_target
    : 'field'
    | 'event'
    | 'method'
    | 'param'
    | 'property'
    | 'return'
```

```
    | 'type'
    ;

attribute_list
    : attribute (',' attribute)*
    ;

attribute
    : attribute_name attribute_arguments?
    ;

attribute_name
    : type_name
    ;

attribute_arguments
    : '(' positional_argument_list? ')'
    | '(' positional_argument_list ',' named_argument_list ')'
    | '(' named_argument_list ')'
    ;

positional_argument_list
    : positional_argument (',' positional_argument)*
    ;

positional_argument
    : attribute_argument_expression
    ;

named_argument_list
    : named_argument (','  named_argument)*
    ;

named_argument
    : identifier '=' attribute_argument_expression
    ;

attribute_argument_expression
    : expression
    ;
```

An attribute consists of an *attribute_name* and an optional list of positional and named arguments. The positional arguments (if any) precede the named arguments. A positional argument consists of an *attribute_argument_expression*; a named argument consists of a name, followed by an equal sign, followed by an *attribute_argument_expression*, which, together, are constrained by the same rules as simple assignment. The order of named arguments is not significant.

The *attribute_name* identifies an attribute class. If the form of *attribute_name* is *type_name* then this name must refer to an attribute class. Otherwise, a compile-time error occurs. The example

```
class Class1 {}

[Class1] class Class2 {}    // Error
```

results in a compile-time error because it attempts to use `Class1` as an attribute class when `Class1` is not an attribute class.

Certain contexts permit the specification of an attribute on more than one target. A program can explicitly specify the target by including an *attribute_target_specifier*. When an attribute is placed at the global level, a *global_attribute_target_specifier* is required. In all other locations, a reasonable default is applied, but an *attribute_target_specifier* can be used to affirm or override the default in certain ambiguous cases (or to just affirm the default in non-ambiguous cases). Thus, typically, *attribute_target_specifier*s can be omitted except at

the global level. The potentially ambiguous contexts are resolved as follows:

- An attribute specified at global scope can apply either to the target assembly or the target module. No default exists for this context, so an *attribute_target_specifier* is always required in this context. The presence of the `assembly` *attribute_target_specifier* indicates that the attribute applies to the target assembly; the presence of the `module` *attribute_target_specifier* indicates that the attribute applies to the target module.
- An attribute specified on a delegate declaration can apply either to the delegate being declared or to its return value. In the absence of an *attribute_target_specifier*, the attribute applies to the delegate. The presence of the `type` *attribute_target_specifier* indicates that the attribute applies to the delegate; the presence of the `return` *attribute_target_specifier* indicates that the attribute applies to the return value.
- An attribute specified on a method declaration can apply either to the method being declared or to its return value. In the absence of an *attribute_target_specifier*, the attribute applies to the method. The presence of the `method` *attribute_target_specifier* indicates that the attribute applies to the method; the presence of the `return` *attribute_target_specifier* indicates that the attribute applies to the return value.
- An attribute specified on an operator declaration can apply either to the operator being declared or to its return value. In the absence of an *attribute_target_specifier*, the attribute applies to the operator. The presence of the `method` *attribute_target_specifier* indicates that the attribute applies to the operator; the presence of the `return` *attribute_target_specifier* indicates that the attribute applies to the return value.
- An attribute specified on an event declaration that omits event accessors can apply to the event being declared, to the associated field (if the event is not abstract), or to the associated add and remove methods. In the absence of an *attribute_target_specifier*, the attribute applies to the event. The presence of the `event` *attribute_target_specifier* indicates that the attribute applies to the event; the presence of the `field` *attribute_target_specifier* indicates that the attribute applies to the field; and the presence of the `method` *attribute_target_specifier* indicates that the attribute applies to the methods.
- An attribute specified on a get accessor declaration for a property or indexer declaration can apply either to the associated method or to its return value. In the absence of an *attribute_target_specifier*, the attribute applies to the method. The presence of the `method` *attribute_target_specifier* indicates that the attribute applies to the method; the presence of the `return` *attribute_target_specifier* indicates that the attribute applies to the return value.
- An attribute specified on a set accessor for a property or indexer declaration can apply either to the associated method or to its lone implicit parameter. In the absence of an *attribute_target_specifier*, the attribute applies to the method. The presence of the `method` *attribute_target_specifier* indicates that the attribute applies to the method; the presence of the `param` *attribute_target_specifier* indicates that the attribute applies to the parameter; the presence of the `return` *attribute_target_specifier* indicates that the attribute applies to the return value.
- An attribute specified on an add or remove accessor declaration for an event declaration can apply either to the associated method or to its lone parameter. In the absence of an *attribute_target_specifier*, the attribute applies to the method. The presence of the `method` *attribute_target_specifier* indicates that the attribute applies to the method; the presence of the `param` *attribute_target_specifier* indicates that the attribute applies to the parameter; the presence of the `return` *attribute_target_specifier* indicates that the attribute applies to the return value.

In other contexts, inclusion of an *attribute_target_specifier* is permitted but unnecessary. For instance, a class declaration may either include or omit the specifier `type`:

```
[type: Author("Brian Kernighan")]
class Class1 {}

[Author("Dennis Ritchie")]
class Class2 {}
```

It is an error to specify an invalid *attribute_target_specifier*. For instance, the specifier `param` cannot be used on

a class declaration:

```
[param: Author("Brian Kernighan")]        // Error
class Class1 {}
```

By convention, attribute classes are named with a suffix of `Attribute`. An *attribute_name* of the form *type_name* may either include or omit this suffix. If an attribute class is found both with and without this suffix, an ambiguity is present, and a compile-time error results. If the *attribute_name* is spelled such that its right-most *identifier* is a verbatim identifier (Identifiers), then only an attribute without a suffix is matched, thus enabling such an ambiguity to be resolved. The example

```
using System;

[AttributeUsage(AttributeTargets.All)]
public class X: Attribute
{}

[AttributeUsage(AttributeTargets.All)]
public class XAttribute: Attribute
{}

[X]                    // Error: ambiguity
class Class1 {}

[XAttribute]           // Refers to XAttribute
class Class2 {}

[@X]                   // Refers to X
class Class3 {}

[@XAttribute]          // Refers to XAttribute
class Class4 {}
```

shows two attribute classes named `X` and `XAttribute`. The attribute `[X]` is ambiguous, since it could refer to either `X` or `XAttribute`. Using a verbatim identifier allows the exact intent to be specified in such rare cases. The attribute `[XAttribute]` is not ambiguous (although it would be if there was an attribute class named `XAttributeAttribute`!). If the declaration for class `X` is removed, then both attributes refer to the attribute class named `XAttribute`, as follows:

```
using System;

[AttributeUsage(AttributeTargets.All)]
public class XAttribute: Attribute
{}

[X]                    // Refers to XAttribute
class Class1 {}

[XAttribute]           // Refers to XAttribute
class Class2 {}

[@X]                   // Error: no attribute named "X"
class Class3 {}
```

It is a compile-time error to use a single-use attribute class more than once on the same entity. The example

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class HelpStringAttribute: Attribute
{
    string value;

    public HelpStringAttribute(string value) {
        this.value = value;
    }

    public string Value {
        get {...}
    }
}

[HelpString("Description of Class1")]
[HelpString("Another description of Class1")]
public class Class1 {}
```

results in a compile-time error because it attempts to use `HelpString`, which is a single-use attribute class, more than once on the declaration of `Class1`.

An expression `E` is an *attribute_argument_expression* if all of the following statements are true:

- The type of `E` is an attribute parameter type (Attribute parameter types).
- At compile-time, the value of `E` can be resolved to one of the following:
  - A constant value.
  - A `System.Type` object.
  - A one-dimensional array of *attribute_argument_expression*s.

For example:

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class TestAttribute: Attribute
{
    public int P1 {
        get {...}
        set {...}
    }

    public Type P2 {
        get {...}
        set {...}
    }

    public object P3 {
        get {...}
        set {...}
    }
}

[Test(P1 = 1234, P3 = new int[] {1, 3, 5}, P2 = typeof(float))]
class MyClass {}
```

A *typeof_expression* (The typeof operator) used as an attribute argument expression can reference a non-generic type, a closed constructed type, or an unbound generic type, but it cannot reference an open type. This is to ensure that the expression can be resolved at compile-time.

```
class A: Attribute
{
    public A(Type t) {...}
}

class G<T>
{
    [A(typeof(T))] T t;                  // Error, open type in attribute
}

class X
{
    [A(typeof(List<int>))] int x;        // Ok, closed constructed type
    [A(typeof(List<>))] int y;           // Ok, unbound generic type
}
```

Attribute instances

An ***attribute instance*** is an instance that represents an attribute at run-time. An attribute is defined with an attribute class, positional arguments, and named arguments. An attribute instance is an instance of the attribute class that is initialized with the positional and named arguments.

Retrieval of an attribute instance involves both compile-time and run-time processing, as described in the following sections.

Compilation of an attribute

The compilation of an *attribute* with attribute class `T`, *positional_argument_list* `P` and *named_argument_list* `N`, consists of the following steps:

- Follow the compile-time processing steps for compiling an *object_creation_expression* of the form `new T(P)`. These steps either result in a compile-time error, or determine an instance constructor `C` on `T` that can be invoked at run-time.
- If `C` does not have public accessibility, then a compile-time error occurs.
- For each *named_argument* `Arg` in `N`:
  - Let `Name` be the *identifier* of the *named_argument* `Arg`.
  - `Name` must identify a non-static read-write public field or property on `T`. If `T` has no such field or property, then a compile-time error occurs.
- Keep the following information for run-time instantiation of the attribute: the attribute class `T`, the instance constructor `C` on `T`, the *positional_argument_list* `P` and the *named_argument_list* `N`.

Run-time retrieval of an attribute instance

Compilation of an *attribute* yields an attribute class `T`, an instance constructor `C` on `T`, a *positional_argument_list* `P`, and a *named_argument_list* `N`. Given this information, an attribute instance can be retrieved at run-time using the following steps:

- Follow the run-time processing steps for executing an *object_creation_expression* of the form `new T(P)`, using the instance constructor `C` as determined at compile-time. These steps either result in an exception, or produce an instance `O` of `T`.
- For each *named_argument* `Arg` in `N`, in order:
  - Let `Name` be the *identifier* of the *named_argument* `Arg`. If `Name` does not identify a non-static public read-write field or property on `O`, then an exception is thrown.
  - Let `Value` be the result of evaluating the *attribute_argument_expression* of `Arg`.
  - If `Name` identifies a field on `O`, then set this field to `Value`.
  - Otherwise, `Name` identifies a property on `O`. Set this property to `Value`.

- The result is `O`, an instance of the attribute class `T` that has been initialized with the *positional_argument_list* `P` and the *named_argument_list* `N`.

Reserved attributes

A small number of attributes affect the language in some way. These attributes include:

- `System.AttributeUsageAttribute` (The AttributeUsage attribute), which is used to describe the ways in which an attribute class can be used.
- `System.Diagnostics.ConditionalAttribute` (The Conditional attribute), which is used to define conditional methods.
- `System.ObsoleteAttribute` (The Obsolete attribute), which is used to mark a member as obsolete.
- `System.Runtime.CompilerServices.CallerLineNumberAttribute`, `System.Runtime.CompilerServices.CallerFilePathAttribute` and `System.Runtime.CompilerServices.CallerMemberNameAttribute` (Caller info attributes), which are used to supply information about the calling context to optional parameters.

The AttributeUsage attribute

The attribute `AttributeUsage` is used to describe the manner in which the attribute class can be used.

A class that is decorated with the `AttributeUsage` attribute must derive from `System.Attribute`, either directly or indirectly. Otherwise, a compile-time error occurs.

```
namespace System
{
    [AttributeUsage(AttributeTargets.Class)]
    public class AttributeUsageAttribute: Attribute
    {
        public AttributeUsageAttribute(AttributeTargets validOn) {...}
        public virtual bool AllowMultiple { get {...} set {...} }
        public virtual bool Inherited { get {...} set {...} }
        public virtual AttributeTargets ValidOn { get {...} }
    }

    public enum AttributeTargets
    {
        Assembly     = 0x0001,
        Module       = 0x0002,
        Class        = 0x0004,
        Struct       = 0x0008,
        Enum         = 0x0010,
        Constructor  = 0x0020,
        Method       = 0x0040,
        Property     = 0x0080,
        Field        = 0x0100,
        Event        = 0x0200,
        Interface    = 0x0400,
        Parameter    = 0x0800,
        Delegate     = 0x1000,
        ReturnValue  = 0x2000,

        All = Assembly | Module | Class | Struct | Enum | Constructor |
            Method | Property | Field | Event | Interface | Parameter |
            Delegate | ReturnValue
    }
}
```

The Conditional attribute

The attribute `Conditional` enables the definition of *conditional methods* and *conditional attribute classes*.

```
namespace System.Diagnostics
{
    [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class, AllowMultiple = true)]
    public class ConditionalAttribute: Attribute
    {
        public ConditionalAttribute(string conditionString) {...}
        public string ConditionString { get {...} }
    }
}
```

Conditional methods

A method decorated with the `Conditional` attribute is a conditional method. The `Conditional` attribute indicates a condition by testing a conditional compilation symbol. Calls to a conditional method are either included or omitted depending on whether this symbol is defined at the point of the call. If the symbol is defined, the call is included; otherwise, the call (including evaluation of the receiver and parameters of the call) is omitted.

A conditional method is subject to the following restrictions:

- The conditional method must be a method in a *class_declaration* or *struct_declaration*. A compile-time error occurs if the `Conditional` attribute is specified on a method in an interface declaration.
- The conditional method must have a return type of `void`.
- The conditional method must not be marked with the `override` modifier. A conditional method may be marked with the `virtual` modifier, however. Overrides of such a method are implicitly conditional, and must not be explicitly marked with a `Conditional` attribute.
- The conditional method must not be an implementation of an interface method. Otherwise, a compile-time error occurs.

In addition, a compile-time error occurs if a conditional method is used in a *delegate_creation_expression*. The example

```
#define DEBUG

using System;
using System.Diagnostics;

class Class1
{
    [Conditional("DEBUG")]
    public static void M() {
        Console.WriteLine("Executed Class1.M");
    }
}

class Class2
{
    public static void Test() {
        Class1.M();
    }
}
```

declares `Class1.M` as a conditional method. `Class2`'s `Test` method calls this method. Since the conditional compilation symbol `DEBUG` is defined, if `Class2.Test` is called, it will call `M`. If the symbol `DEBUG` had not been defined, then `Class2.Test` would not call `Class1.M`.

It is important to note that the inclusion or exclusion of a call to a conditional method is controlled by the conditional compilation symbols at the point of the call. In the example

File `class1.cs`:

```
using System.Diagnostics;

class Class1
{
    [Conditional("DEBUG")]
    public static void F() {
        Console.WriteLine("Executed Class1.F");
    }
}
```

File `class2.cs`:

```
#define DEBUG

class Class2
{
    public static void G() {
        Class1.F();               // F is called
    }
}
```

File `class3.cs`:

```
#undef DEBUG

class Class3
{
    public static void H() {
        Class1.F();               // F is not called
    }
}
```

the classes `Class2` and `Class3` each contain calls to the conditional method `Class1.F`, which is conditional based on whether or not `DEBUG` is defined. Since this symbol is defined in the context of `Class2` but not `Class3`, the call to `F` in `Class2` is included, while the call to `F` in `Class3` is omitted.

The use of conditional methods in an inheritance chain can be confusing. Calls made to a conditional method through `base`, of the form `base.M`, are subject to the normal conditional method call rules. In the example

File `class1.cs`:

```
using System;
using System.Diagnostics;

class Class1
{
    [Conditional("DEBUG")]
    public virtual void M() {
        Console.WriteLine("Class1.M executed");
    }
}
```

File `class2.cs`:

```
using System;

class Class2: Class1
{
    public override void M() {
        Console.WriteLine("Class2.M executed");
        base.M();                       // base.M is not called!
    }
}
```

File `class3.cs`:

```
#define DEBUG

using System;

class Class3
{
    public static void Test() {
        Class2 c = new Class2();
        c.M();                          // M is called
    }
}
```

`Class2` includes a call to the `M` defined in its base class. This call is omitted because the base method is conditional based on the presence of the symbol `DEBUG`, which is undefined. Thus, the method writes to the console "`Class2.M executed`" only. Judicious use of *pp_declaration*s can eliminate such problems.

Conditional attribute classes

An attribute class (Attribute classes) decorated with one or more `Conditional` attributes is a ***conditional attribute class***. A conditional attribute class is thus associated with the conditional compilation symbols declared in its `Conditional` attributes. This example:

```
using System;
using System.Diagnostics;
[Conditional("ALPHA")]
[Conditional("BETA")]
public class TestAttribute : Attribute {}
```

declares `TestAttribute` as a conditional attribute class associated with the conditional compilations symbols `ALPHA` and `BETA`.

Attribute specifications (Attribute specification) of a conditional attribute are included if one or more of its associated conditional compilation symbols is defined at the point of specification, otherwise the attribute specification is omitted.

It is important to note that the inclusion or exclusion of an attribute specification of a conditional attribute class is controlled by the conditional compilation symbols at the point of the specification. In the example

File `test.cs`:

```
using System;
using System.Diagnostics;

[Conditional("DEBUG")]

public class TestAttribute : Attribute {}
```

File `class1.cs`:

```
#define DEBUG

[Test]               // TestAttribute is specified

class Class1 {}
```

File `class2.cs`:

```
#undef DEBUG

[Test]                // TestAttribute is not specified

class Class2 {}
```

the classes `Class1` and `Class2` are each decorated with attribute `Test`, which is conditional based on whether or not `DEBUG` is defined. Since this symbol is defined in the context of `Class1` but not `Class2`, the specification of the `Test` attribute on `Class1` is included, while the specification of the `Test` attribute on `Class2` is omitted.

The Obsolete attribute

The attribute `Obsolete` is used to mark types and members of types that should no longer be used.

```
namespace System
{
    [AttributeUsage(
        AttributeTargets.Class |
        AttributeTargets.Struct |
        AttributeTargets.Enum |
        AttributeTargets.Interface |
        AttributeTargets.Delegate |
        AttributeTargets.Method |
        AttributeTargets.Constructor |
        AttributeTargets.Property |
        AttributeTargets.Field |
        AttributeTargets.Event,
        Inherited = false)
    ]
    public class ObsoleteAttribute: Attribute
    {
        public ObsoleteAttribute() {...}
        public ObsoleteAttribute(string message) {...}
        public ObsoleteAttribute(string message, bool error) {...}
        public string Message { get {...} }
        public bool IsError { get {...} }
    }
}
```

If a program uses a type or member that is decorated with the `Obsolete` attribute, the compiler issues a warning or an error. Specifically, the compiler issues a warning if no error parameter is provided, or if the error parameter is provided and has the value `false`. The compiler issues an error if the error parameter is specified and has the value `true`.

In the example

```
[Obsolete("This class is obsolete; use class B instead")]
class A
{
    public void F() {}
}

class B
{
    public void F() {}
}

class Test
{
    static void Main() {
        A a = new A();         // Warning
        a.F();
    }
}
```

the class `A` is decorated with the `Obsolete` attribute. Each use of `A` in `Main` results in a warning that includes the specified message, "This class is obsolete; use class B instead."

Caller info attributes

For purposes such as logging and reporting, it is sometimes useful for a function member to obtain certain compile-time information about the calling code. The caller info attributes provide a way to pass such information transparently.

When an optional parameter is annotated with one of the caller info attributes, omitting the corresponding argument in a call does not necessarily cause the default parameter value to be substituted. Instead, if the specified information about the calling context is available, that information will be passed as the argument value.

For example:

```
using System.Runtime.CompilerServices

...

public void Log(
    [CallerLineNumber] int line = -1,
    [CallerFilePath]   string path = null,
    [CallerMemberName] string name = null
)
{
    Console.WriteLine((line < 0) ? "No line" : "Line "+ line);
    Console.WriteLine((path == null) ? "No file path" : path);
    Console.WriteLine((name == null) ? "No member name" : name);
}
```

A call to `Log()` with no arguments would print the line number and file path of the call, as well as the name of the member within which the call occurred.

Caller info attributes can occur on optional parameters anywhere, including in delegate declarations. However, the specific caller info attributes have restrictions on the types of the parameters they can attribute, so that there will always be an implicit conversion from a substituted value to the parameter type.

It is an error to have the same caller info attribute on a parameter of both the defining and implementing part of a partial method declaration. Only caller info attributes in the defining part are applied, whereas caller info attributes occurring only in the implementing part are ignored.

Caller information does not affect overload resolution. As the attributed optional parameters are still omitted from the source code of the caller, overload resolution ignores those parameters in the same way it ignores other omitted optional parameters (Overload resolution).

Caller information is only substituted when a function is explicitly invoked in source code. Implicit invocations such as implicit parent constructor calls do not have a source location and will not substitute caller information. Also, calls that are dynamically bound will not substitute caller information. When a caller info attributed parameter is omitted in such cases, the specified default value of the parameter is used instead.

One exception is query-expressions. These are considered syntactic expansions, and if the calls they expand to omit optional parameters with caller info attributes, caller information will be substituted. The location used is the location of the query clause which the call was generated from.

If more than one caller info attribute is specified on a given parameter, they are preferred in the following order: `CallerLineNumber`, `CallerFilePath`, `CallerMemberName`.

The CallerLineNumber attribute

The `System.Runtime.CompilerServices.CallerLineNumberAttribute` is allowed on optional parameters when there is a standard implicit conversion (Standard implicit conversions) from the constant value `int.MaxValue` to the parameter's type. This ensures that any non-negative line number up to that value can be passed without error.

If a function invocation from a location in source code omits an optional parameter with the `CallerLineNumberAttribute`, then a numeric literal representing that location's line number is used as an argument to the invocation instead of the default parameter value.

If the invocation spans multiple lines, the line chosen is implementation-dependent.

Note that the line number may be affected by `#line` directives (Line directives).

The CallerFilePath attribute

The `System.Runtime.CompilerServices.CallerFilePathAttribute` is allowed on optional parameters when there is a standard implicit conversion (Standard implicit conversions) from `string` to the parameter's type.

If a function invocation from a location in source code omits an optional parameter with the `CallerFilePathAttribute`, then a string literal representing that location's file path is used as an argument to the invocation instead of the default parameter value.

The format of the file path is implementation-dependent.

Note that the file path may be affected by `#line` directives (Line directives).

The CallerMemberName attribute

The `System.Runtime.CompilerServices.CallerMemberNameAttribute` is allowed on optional parameters when there is a standard implicit conversion (Standard implicit conversions) from `string` to the parameter's type.

If a function invocation from a location within the body of a function member or within an attribute applied to the function member itself or its return type, parameters or type parameters in source code omits an optional parameter with the `CallerMemberNameAttribute`, then a string literal representing the name of that member is used as an argument to the invocation instead of the default parameter value.

For invocations that occur within generic methods, only the method name itself is used, without the type parameter list.

For invocations that occur within explicit interface member implementations, only the method name itself is used, without the preceding interface qualification.

For invocations that occur within property or event accessors, the member name used is that of the property or event itself.

For invocations that occur within indexer accessors, the member name used is that supplied by an `IndexerNameAttribute` (The IndexerName attribute) on the indexer member, if present, or the default name `Item` otherwise.

For invocations that occur within declarations of instance constructors, static constructors, destructors and operators the member name used is implementation-dependent.

Attributes for Interoperation

Note: This section is applicable only to the Microsoft .NET implementation of C#.

Interoperation with COM and Win32 components

The .NET run-time provides a large number of attributes that enable C# programs to interoperate with components written using COM and Win32 DLLs. For example, the `DllImport` attribute can be used on a `static extern` method to indicate that the implementation of the method is to be found in a Win32 DLL. These attributes are found in the `System.Runtime.InteropServices` namespace, and detailed documentation for these attributes is found in the .NET runtime documentation.

Interoperation with other .NET languages

The IndexerName attribute

Indexers are implemented in .NET using indexed properties, and have a name in the .NET metadata. If no `IndexerName` attribute is present for an indexer, then the name `Item` is used by default. The `IndexerName` attribute enables a developer to override this default and specify a different name.

```
namespace System.Runtime.CompilerServices.CSharp
{
    [AttributeUsage(AttributeTargets.Property)]
    public class IndexerNameAttribute: Attribute
    {
        public IndexerNameAttribute(string indexerName) {...}
        public string Value { get {...} }
    }
}
```

Unsafe code

18.12.2021 • 39 minutes to read

The core C# language, as defined in the preceding chapters, differs notably from C and C++ in its omission of pointers as a data type. Instead, C# provides references and the ability to create objects that are managed by a garbage collector. This design, coupled with other features, makes C# a much safer language than C or C++. In the core C# language it is simply not possible to have an uninitialized variable, a "dangling" pointer, or an expression that indexes an array beyond its bounds. Whole categories of bugs that routinely plague C and C++ programs are thus eliminated.

While practically every pointer type construct in C or C++ has a reference type counterpart in C#, nonetheless, there are situations where access to pointer types becomes a necessity. For example, interfacing with the underlying operating system, accessing a memory-mapped device, or implementing a time-critical algorithm may not be possible or practical without access to pointers. To address this need, C# provides the ability to write ***unsafe code***.

In unsafe code it is possible to declare and operate on pointers, to perform conversions between pointers and integral types, to take the address of variables, and so forth. In a sense, writing unsafe code is much like writing C code within a C# program.

Unsafe code is in fact a "safe" feature from the perspective of both developers and users. Unsafe code must be clearly marked with the modifier `unsafe`, so developers can't possibly use unsafe features accidentally, and the execution engine works to ensure that unsafe code cannot be executed in an untrusted environment.

Unsafe contexts

The unsafe features of C# are available only in unsafe contexts. An unsafe context is introduced by including an `unsafe` modifier in the declaration of a type or member, or by employing an *unsafe_statement*.

- A declaration of a class, struct, interface, or delegate may include an `unsafe` modifier, in which case the entire textual extent of that type declaration (including the body of the class, struct, or interface) is considered an unsafe context.
- A declaration of a field, method, property, event, indexer, operator, instance constructor, destructor, or static constructor may include an `unsafe` modifier, in which case the entire textual extent of that member declaration is considered an unsafe context.
- An *unsafe_statement* enables the use of an unsafe context within a *block*. The entire textual extent of the associated *block* is considered an unsafe context.

The associated grammar productions are shown below.

```
class_modifier_unsafe
    : 'unsafe'
    ;

struct_modifier_unsafe
    : 'unsafe'
    ;

interface_modifier_unsafe
    : 'unsafe'
    ;

delegate_modifier_unsafe
    : 'unsafe'
    ;

field_modifier_unsafe
    : 'unsafe'
    ;

method_modifier_unsafe
    : 'unsafe'
    ;

property_modifier_unsafe
    : 'unsafe'
    ;

event_modifier_unsafe
    : 'unsafe'
    ;

indexer_modifier_unsafe
    : 'unsafe'
    ;

operator_modifier_unsafe
    : 'unsafe'
    ;

constructor_modifier_unsafe
    : 'unsafe'
    ;

destructor_declaration_unsafe
    : attributes? 'extern'? 'unsafe'? '~' identifier '(' ')' destructor_body
    | attributes? 'unsafe'? 'extern'? '~' identifier '(' ')' destructor_body
    ;

static_constructor_modifiers_unsafe
    : 'extern'? 'unsafe'? 'static'
    | 'unsafe'? 'extern'? 'static'
    | 'extern'? 'static' 'unsafe'?
    | 'unsafe'? 'static' 'extern'?
    | 'static' 'extern'? 'unsafe'?
    | 'static' 'unsafe'? 'extern'?
    ;

embedded_statement_unsafe
    : unsafe_statement
    | fixed_statement
    ;

unsafe_statement
    : 'unsafe' block
    ;
```

In the example

```
public unsafe struct Node
{
    public int Value;
    public Node* Left;
    public Node* Right;
}
```

the `unsafe` modifier specified in the struct declaration causes the entire textual extent of the struct declaration to become an unsafe context. Thus, it is possible to declare the `Left` and `Right` fields to be of a pointer type. The example above could also be written

```
public struct Node
{
    public int Value;
    public unsafe Node* Left;
    public unsafe Node* Right;
}
```

Here, the `unsafe` modifiers in the field declarations cause those declarations to be considered unsafe contexts.

Other than establishing an unsafe context, thus permitting the use of pointer types, the `unsafe` modifier has no effect on a type or a member. In the example

```
public class A
{
    public unsafe virtual void F() {
        char* p;
        ...
    }
}

public class B: A
{
    public override void F() {
        base.F();
        ...
    }
}
```

the `unsafe` modifier on the `F` method in `A` simply causes the textual extent of `F` to become an unsafe context in which the unsafe features of the language can be used. In the override of `F` in `B`, there is no need to re-specify the `unsafe` modifier -- unless, of course, the `F` method in `B` itself needs access to unsafe features.

The situation is slightly different when a pointer type is part of the method's signature

```
public unsafe class A
{
    public virtual void F(char* p) {...}
}

public class B: A
{
    public unsafe override void F(char* p) {...}
}
```

Here, because `F`'s signature includes a pointer type, it can only be written in an unsafe context. However, the unsafe context can be introduced by either making the entire class unsafe, as is the case in `A`, or by including an

`unsafe` modifier in the method declaration, as is the case in `B`.

Pointer types

In an unsafe context, a *type* (Types) may be a *pointer_type* as well as a *value_type* or a *reference_type*. However, a *pointer_type* may also be used in a `typeof` expression (Anonymous object creation expressions) outside of an unsafe context as such usage is not unsafe.

```
type_unsafe
    : pointer_type
    ;
```

A *pointer_type* is written as an *unmanaged_type* or the keyword `void`, followed by a `*` token:

```
pointer_type
    : unmanaged_type '*'
    | 'void' '*'
    ;

unmanaged_type
    : type
    ;
```

The type specified before the `*` in a pointer type is called the ***referent type*** of the pointer type. It represents the type of the variable to which a value of the pointer type points.

Unlike references (values of reference types), pointers are not tracked by the garbage collector -- the garbage collector has no knowledge of pointers and the data to which they point. For this reason a pointer is not permitted to point to a reference or to a struct that contains references, and the referent type of a pointer must be an *unmanaged_type*.

An *unmanaged_type* is any type that isn't a *reference_type* or constructed type, and doesn't contain *reference_type* or constructed type fields at any level of nesting. In other words, an *unmanaged_type* is one of the following:

- `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, or `bool`.
- Any *enum_type*.
- Any *pointer_type*.
- Any user-defined *struct_type* that is not a constructed type and contains fields of *unmanaged_type*s only.

The intuitive rule for mixing of pointers and references is that referents of references (objects) are permitted to contain pointers, but referents of pointers are not permitted to contain references.

Some examples of pointer types are given in the table below:

EXAMPLE	DESCRIPTION
`byte*`	Pointer to `byte`
`char*`	Pointer to `char`
`int**`	Pointer to pointer to `int`
`int*[]`	Single-dimensional array of pointers to `int`

EXAMPLE	DESCRIPTION
`void*`	Pointer to unknown type

For a given implementation, all pointer types must have the same size and representation.

Unlike C and C++, when multiple pointers are declared in the same declaration, in C# the `*` is written along with the underlying type only, not as a prefix punctuator on each pointer name. For example

```
int* pi, pj;    // NOT as int *pi, *pj;
```

The value of a pointer having type `T*` represents the address of a variable of type `T`. The pointer indirection operator `*` (Pointer indirection) may be used to access this variable. For example, given a variable `P` of type `int*`, the expression `*P` denotes the `int` variable found at the address contained in `P`.

Like an object reference, a pointer may be `null`. Applying the indirection operator to a `null` pointer results in implementation-defined behavior. A pointer with value `null` is represented by all-bits-zero.

The `void*` type represents a pointer to an unknown type. Because the referent type is unknown, the indirection operator cannot be applied to a pointer of type `void*`, nor can any arithmetic be performed on such a pointer. However, a pointer of type `void*` can be cast to any other pointer type (and vice versa).

Pointer types are a separate category of types. Unlike reference types and value types, pointer types do not inherit from `object` and no conversions exist between pointer types and `object`. In particular, boxing and unboxing (Boxing and unboxing) are not supported for pointers. However, conversions are permitted between different pointer types and between pointer types and the integral types. This is described in Pointer conversions.

A *pointer_type* cannot be used as a type argument (Constructed types), and type inference (Type inference) fails on generic method calls that would have inferred a type argument to be a pointer type.

A *pointer_type* may be used as the type of a volatile field (Volatile fields).

Although pointers can be passed as `ref` or `out` parameters, doing so can cause undefined behavior, since the pointer may well be set to point to a local variable which no longer exists when the called method returns, or the fixed object to which it used to point, is no longer fixed. For example:

```
using System;

class Test
{
    static int value = 20;

    unsafe static void F(out int* pi1, ref int* pi2) {
        int i = 10;
        pi1 = &i;

        fixed (int* pj = &value) {
            // ...
            pi2 = pj;
        }
    }

    static void Main() {
        int i = 10;
        unsafe {
            int* px1;
            int* px2 = &i;

            F(out px1, ref px2);

            Console.WriteLine("*px1 = {0}, *px2 = {1}",
                *px1, *px2);    // undefined behavior
        }
    }
}
```

A method can return a value of some type, and that type can be a pointer. For example, when given a pointer to a contiguous sequence of `int`s, that sequence's element count, and some other `int` value, the following method returns the address of that value in that sequence, if a match occurs; otherwise it returns `null`:

```
unsafe static int* Find(int* pi, int size, int value) {
    for (int i = 0; i < size; ++i) {
        if (*pi == value)
            return pi;
        ++pi;
    }
    return null;
}
```

In an unsafe context, several constructs are available for operating on pointers:

- The `*` operator may be used to perform pointer indirection (Pointer indirection).
- The `->` operator may be used to access a member of a struct through a pointer (Pointer member access).
- The `[]` operator may be used to index a pointer (Pointer element access).
- The `&` operator may be used to obtain the address of a variable (The address-of operator).
- The `++` and `--` operators may be used to increment and decrement pointers (Pointer increment and decrement).
- The `+` and `-` operators may be used to perform pointer arithmetic (Pointer arithmetic).
- The `==`, `!=`, `<`, `>`, `<=`, and `>=` operators may be used to compare pointers (Pointer comparison).
- The `stackalloc` operator may be used to allocate memory from the call stack (Fixed size buffers).
- The `fixed` statement may be used to temporarily fix a variable so its address can be obtained (The fixed statement).

Fixed and moveable variables

The address-of operator (The address-of operator) and the `fixed` statement (The fixed statement) divide variables into two categories: ***Fixed variables*** and ***moveable variables***.

Fixed variables reside in storage locations that are unaffected by operation of the garbage collector. (Examples of fixed variables include local variables, value parameters, and variables created by dereferencing pointers.) On the other hand, moveable variables reside in storage locations that are subject to relocation or disposal by the garbage collector. (Examples of moveable variables include fields in objects and elements of arrays.)

The `&` operator (The address-of operator) permits the address of a fixed variable to be obtained without restrictions. However, because a moveable variable is subject to relocation or disposal by the garbage collector, the address of a moveable variable can only be obtained using a `fixed` statement (The fixed statement), and that address remains valid only for the duration of that `fixed` statement.

In precise terms, a fixed variable is one of the following:

- A variable resulting from a *simple_name* (Simple names) that refers to a local variable or a value parameter, unless the variable is captured by an anonymous function.
- A variable resulting from a *member_access* (Member access) of the form `V.I`, where `V` is a fixed variable of a *struct_type*.
- A variable resulting from a *pointer_indirection_expression* (Pointer indirection) of the form `*P`, a *pointer_member_access* (Pointer member access) of the form `P->I`, or a *pointer_element_access* (Pointer element access) of the form `P[E]`.

All other variables are classified as moveable variables.

Note that a static field is classified as a moveable variable. Also note that a `ref` or `out` parameter is classified as a moveable variable, even if the argument given for the parameter is a fixed variable. Finally, note that a variable produced by dereferencing a pointer is always classified as a fixed variable.

Pointer conversions

In an unsafe context, the set of available implicit conversions (Implicit conversions) is extended to include the following implicit pointer conversions:

- From any *pointer_type* to the type `void*`.
- From the `null` literal to any *pointer_type*.

Additionally, in an unsafe context, the set of available explicit conversions (Explicit conversions) is extended to include the following explicit pointer conversions:

- From any *pointer_type* to any other *pointer_type*.
- From `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, or `ulong` to any *pointer_type*.
- From any *pointer_type* to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, or `ulong`.

Finally, in an unsafe context, the set of standard implicit conversions (Standard implicit conversions) includes the following pointer conversion:

- From any *pointer_type* to the type `void*`.

Conversions between two pointer types never change the actual pointer value. In other words, a conversion from one pointer type to another has no effect on the underlying address given by the pointer.

When one pointer type is converted to another, if the resulting pointer is not correctly aligned for the pointed-to type, the behavior is undefined if the result is dereferenced. In general, the concept "correctly aligned" is transitive: if a pointer to type `A` is correctly aligned for a pointer to type `B`, which, in turn, is correctly aligned for a pointer to type `C`, then a pointer to type `A` is correctly aligned for a pointer to type `C`.

Consider the following case in which a variable having one type is accessed via a pointer to a different type:

```
char c = 'A';
char* pc = &c;
void* pv = pc;
int* pi = (int*)pv;
int i = *pi;         // undefined
*pi = 123456;        // undefined
```

When a pointer type is converted to a pointer to byte, the result points to the lowest addressed byte of the variable. Successive increments of the result, up to the size of the variable, yield pointers to the remaining bytes of that variable. For example, the following method displays each of the eight bytes in a double as a hexadecimal value:

```
using System;

class Test
{
    unsafe static void Main() {
      double d = 123.456e23;
        unsafe {
           byte* pb = (byte*)&d;
            for (int i = 0; i < sizeof(double); ++i)
               Console.Write("{0:X2} ", *pb++);
            Console.WriteLine();
        }
    }
}
```

Of course, the output produced depends on endianness.

Mappings between pointers and integers are implementation-defined. However, on 32* and 64-bit CPU architectures with a linear address space, conversions of pointers to or from integral types typically behave exactly like conversions of `uint` or `ulong` values, respectively, to or from those integral types.

Pointer arrays

In an unsafe context, arrays of pointers can be constructed. Only some of the conversions that apply to other array types are allowed on pointer arrays:

- The implicit reference conversion (Implicit reference conversions) from any *array_type* to `System.Array` and the interfaces it implements also applies to pointer arrays. However, any attempt to access the array elements through `System.Array` or the interfaces it implements will result in an exception at run-time, as pointer types are not convertible to `object`.
- The implicit and explicit reference conversions (Implicit reference conversions, Explicit reference conversions) from a single-dimensional array type `S[]` to `System.Collections.Generic.IList<T>` and its generic base interfaces never apply to pointer arrays, since pointer types cannot be used as type arguments, and there are no conversions from pointer types to non-pointer types.
- The explicit reference conversion (Explicit reference conversions) from `System.Array` and the interfaces it implements to any *array_type* applies to pointer arrays.
- The explicit reference conversions (Explicit reference conversions) from `System.Collections.Generic.IList<S>` and its base interfaces to a single-dimensional array type `T[]` never applies to pointer arrays, since pointer types cannot be used as type arguments, and there are no conversions from pointer types to non-pointer types.

These restrictions mean that the expansion for the `foreach` statement over arrays described in The foreach statement cannot be applied to pointer arrays. Instead, a foreach statement of the form

```
foreach (V v in x) embedded_statement
```

where the type of `x` is an array type of the form `T[,,...,]`, `N` is the number of dimensions minus 1 and `T` or `V` is a pointer type, is expanded using nested for-loops as follows:

```
{
    T[,,...,] a = x;
    for (int i0 = a.GetLowerBound(0); i0 <= a.GetUpperBound(0); i0++)
    for (int i1 = a.GetLowerBound(1); i1 <= a.GetUpperBound(1); i1++)
    ...
    for (int iN = a.GetLowerBound(N); iN <= a.GetUpperBound(N); iN++) {
        V v = (V)a.GetValue(i0,i1,...,iN);
        embedded_statement
    }
}
```

The variables `a`, `i0`, `i1`, ..., `iN` are not visible to or accessible to `x` or the *embedded_statement* or any other source code of the program. The variable `v` is read-only in the embedded statement. If there is not an explicit conversion (Pointer conversions) from `T` (the element type) to `V`, an error is produced and no further steps are taken. If `x` has the value `null`, a `System.NullReferenceException` is thrown at run-time.

Pointers in expressions

In an unsafe context, an expression may yield a result of a pointer type, but outside an unsafe context it is a compile-time error for an expression to be of a pointer type. In precise terms, outside an unsafe context a compile-time error occurs if any *simple_name* (Simple names), *member_access* (Member access), *invocation_expression* (Invocation expressions), or *element_access* (Element access) is of a pointer type.

In an unsafe context, the *primary_no_array_creation_expression* (Primary expressions) and *unary_expression* (Unary operators) productions permit the following additional constructs:

```
primary_no_array_creation_expression_unsafe
    : pointer_member_access
    | pointer_element_access
    | sizeof_expression
    ;

unary_expression_unsafe
    : pointer_indirection_expression
    | addressof_expression
    ;
```

These constructs are described in the following sections. The precedence and associativity of the unsafe operators is implied by the grammar.

Pointer indirection

A *pointer_indirection_expression* consists of an asterisk (`*`) followed by a *unary_expression*.

```
pointer_indirection_expression
    : '*' unary_expression
    ;
```

The unary `*` operator denotes pointer indirection and is used to obtain the variable to which a pointer points. The result of evaluating `*P`, where `P` is an expression of a pointer type `T*`, is a variable of type `T`. It is a compile-time error to apply the unary `*` operator to an expression of type `void*` or to an expression that isn't

of a pointer type.

The effect of applying the unary `*` operator to a `null` pointer is implementation-defined. In particular, there is no guarantee that this operation throws a `System.NullReferenceException`.

If an invalid value has been assigned to the pointer, the behavior of the unary `*` operator is undefined. Among the invalid values for dereferencing a pointer by the unary `*` operator are an address inappropriately aligned for the type pointed to (see example in Pointer conversions), and the address of a variable after the end of its lifetime.

For purposes of definite assignment analysis, a variable produced by evaluating an expression of the form `*P` is considered initially assigned (Initially assigned variables).

Pointer member access

A *pointer_member_access* consists of a *primary_expression*, followed by a "`->`" token, followed by an *identifier* and an optional *type_argument_list*.

```
pointer_member_access
    : primary_expression '->' identifier
    ;
```

In a pointer member access of the form `P->I`, `P` must be an expression of a pointer type other than `void*`, and `I` must denote an accessible member of the type to which `P` points.

A pointer member access of the form `P->I` is evaluated exactly as `(*P).I`. For a description of the pointer indirection operator (`*`), see Pointer indirection. For a description of the member access operator (`.`), see Member access.

In the example

```
using System;

struct Point
{
    public int x;
    public int y;

    public override string ToString() {
        return "(" + x + "," + y + ")";
    }
}

class Test
{
    static void Main() {
        Point point;
        unsafe {
            Point* p = &point;
            p->x = 10;
            p->y = 20;
            Console.WriteLine(p->ToString());
        }
    }
}
```

the `->` operator is used to access fields and invoke a method of a struct through a pointer. Because the operation `P->I` is precisely equivalent to `(*P).I`, the `Main` method could equally well have been written:

```
class Test
{
    static void Main() {
        Point point;
        unsafe {
            Point* p = &point;
            (*p).x = 10;
            (*p).y = 20;
            Console.WriteLine((*p).ToString());
        }
    }
}
```

Pointer element access

A *pointer_element_access* consists of a *primary_no_array_creation_expression* followed by an expression enclosed in "`[`" and "`]`".

```
pointer_element_access
    : primary_no_array_creation_expression '[' expression ']'
    ;
```

In a pointer element access of the form `P[E]`, `P` must be an expression of a pointer type other than `void*`, and `E` must be an expression that can be implicitly converted to `int`, `uint`, `long`, or `ulong`.

A pointer element access of the form `P[E]` is evaluated exactly as `*(P + E)`. For a description of the pointer indirection operator (`*`), see Pointer indirection. For a description of the pointer addition operator (`+`), see Pointer arithmetic.

In the example

```
class Test
{
    static void Main() {
        unsafe {
            char* p = stackalloc char[256];
            for (int i = 0; i < 256; i++) p[i] = (char)i;
        }
    }
}
```

a pointer element access is used to initialize the character buffer in a `for` loop. Because the operation `P[E]` is precisely equivalent to `*(P + E)`, the example could equally well have been written:

```
class Test
{
    static void Main() {
        unsafe {
            char* p = stackalloc char[256];
            for (int i = 0; i < 256; i++) *(p + i) = (char)i;
        }
    }
}
```

The pointer element access operator does not check for out-of-bounds errors and the behavior when accessing an out-of-bounds element is undefined. This is the same as C and C++.

The address-of operator

An *addressof_expression* consists of an ampersand (`&`) followed by a *unary_expression*.

```
addressof_expression
    : '&' unary_expression
    ;
```

Given an expression `E` which is of a type `T` and is classified as a fixed variable (Fixed and moveable variables), the construct `&E` computes the address of the variable given by `E`. The type of the result is `T*` and is classified as a value. A compile-time error occurs if `E` is not classified as a variable, if `E` is classified as a read-only local variable, or if `E` denotes a moveable variable. In the last case, a fixed statement (The fixed statement) can be used to temporarily "fix" the variable before obtaining its address. As stated in Member access, outside an instance constructor or static constructor for a struct or class that defines a `readonly` field, that field is considered a value, not a variable. As such, its address cannot be taken. Similarly, the address of a constant cannot be taken.

The `&` operator does not require its argument to be definitely assigned, but following an `&` operation, the variable to which the operator is applied is considered definitely assigned in the execution path in which the operation occurs. It is the responsibility of the programmer to ensure that correct initialization of the variable actually does take place in this situation.

In the example

```
using System;

class Test
{
    static void Main() {
        int i;
        unsafe {
            int* p = &i;
            *p = 123;
        }
        Console.WriteLine(i);
    }
}
```

`i` is considered definitely assigned following the `&i` operation used to initialize `p`. The assignment to `*p` in effect initializes `i`, but the inclusion of this initialization is the responsibility of the programmer, and no compile-time error would occur if the assignment was removed.

The rules of definite assignment for the `&` operator exist such that redundant initialization of local variables can be avoided. For example, many external APIs take a pointer to a structure which is filled in by the API. Calls to such APIs typically pass the address of a local struct variable, and without the rule, redundant initialization of the struct variable would be required.

Pointer increment and decrement

In an unsafe context, the `++` and `--` operators (Postfix increment and decrement operators and Prefix increment and decrement operators) can be applied to pointer variables of all types except `void*`. Thus, for every pointer type `T*`, the following operators are implicitly defined:

```
T* operator ++(T* x);
T* operator --(T* x);
```

The operators produce the same results as `x + 1` and `x - 1`, respectively (Pointer arithmetic). In other words, for a pointer variable of type `T*`, the `++` operator adds `sizeof(T)` to the address contained in the variable, and

the `--` operator subtracts `sizeof(T)` from the address contained in the variable.

If a pointer increment or decrement operation overflows the domain of the pointer type, the result is implementation-defined, but no exceptions are produced.

Pointer arithmetic

In an unsafe context, the `+` and `-` operators (Addition operator and Subtraction operator) can be applied to values of all pointer types except `void*`. Thus, for every pointer type `T*`, the following operators are implicitly defined:

```
T* operator +(T* x, int y);
T* operator +(T* x, uint y);
T* operator +(T* x, long y);
T* operator +(T* x, ulong y);

T* operator +(int x, T* y);
T* operator +(uint x, T* y);
T* operator +(long x, T* y);
T* operator +(ulong x, T* y);

T* operator -(T* x, int y);
T* operator -(T* x, uint y);
T* operator -(T* x, long y);
T* operator -(T* x, ulong y);

long operator -(T* x, T* y);
```

Given an expression `P` of a pointer type `T*` and an expression `N` of type `int`, `uint`, `long`, or `ulong`, the expressions `P + N` and `N + P` compute the pointer value of type `T*` that results from adding `N * sizeof(T)` to the address given by `P`. Likewise, the expression `P - N` computes the pointer value of type `T*` that results from subtracting `N * sizeof(T)` from the address given by `P`.

Given two expressions, `P` and `Q`, of a pointer type `T*`, the expression `P - Q` computes the difference between the addresses given by `P` and `Q` and then divides that difference by `sizeof(T)`. The type of the result is always `long`. In effect, `P - Q` is computed as `((long)(P) - (long)(Q)) / sizeof(T)`.

For example:

```
using System;

class Test
{
    static void Main() {
        unsafe {
            int* values = stackalloc int[20];
            int* p = &values[1];
            int* q = &values[15];
            Console.WriteLine("p - q = {0}", p - q);
            Console.WriteLine("q - p = {0}", q - p);
        }
    }
}
```

which produces the output:

```
p - q = -14
q - p = 14
```

If a pointer arithmetic operation overflows the domain of the pointer type, the result is truncated in an

implementation-defined fashion, but no exceptions are produced.

Pointer comparison

In an unsafe context, the `==`, `!=`, `<`, `>`, `<=`, and `=>` operators (Relational and type-testing operators) can be applied to values of all pointer types. The pointer comparison operators are:

```
bool operator ==(void* x, void* y);
bool operator !=(void* x, void* y);
bool operator <(void* x, void* y);
bool operator >(void* x, void* y);
bool operator <=(void* x, void* y);
bool operator >=(void* x, void* y);
```

Because an implicit conversion exists from any pointer type to the `void*` type, operands of any pointer type can be compared using these operators. The comparison operators compare the addresses given by the two operands as if they were unsigned integers.

The sizeof operator

The `sizeof` operator returns the number of bytes occupied by a variable of a given type. The type specified as an operand to `sizeof` must be an *unmanaged_type* (Pointer types).

```
sizeof_expression
    : 'sizeof' '(' unmanaged_type ')'
    ;
```

The result of the `sizeof` operator is a value of type `int`. For certain predefined types, the `sizeof` operator yields a constant value as shown in the table below.

EXPRESSION	RESULT
`sizeof(sbyte)`	`1`
`sizeof(byte)`	`1`
`sizeof(short)`	`2`
`sizeof(ushort)`	`2`
`sizeof(int)`	`4`
`sizeof(uint)`	`4`
`sizeof(long)`	`8`
`sizeof(ulong)`	`8`
`sizeof(char)`	`2`
`sizeof(float)`	`4`
`sizeof(double)`	`8`

EXPRESSION	RESULT
`sizeof(bool)`	`1`

For all other types, the result of the `sizeof` operator is implementation-defined and is classified as a value, not a constant.

The order in which members are packed into a struct is unspecified.

For alignment purposes, there may be unnamed padding at the beginning of a struct, within a struct, and at the end of the struct. The contents of the bits used as padding are indeterminate.

When applied to an operand that has struct type, the result is the total number of bytes in a variable of that type, including any padding.

The fixed statement

In an unsafe context, the *embedded_statement* (Statements) production permits an additional construct, the `fixed` statement, which is used to "fix" a moveable variable such that its address remains constant for the duration of the statement.

```
fixed_statement
    : 'fixed' '(' pointer_type fixed_pointer_declarators ')' embedded_statement
    ;

fixed_pointer_declarators
    : fixed_pointer_declarator (','  fixed_pointer_declarator)*
    ;

fixed_pointer_declarator
    : identifier '=' fixed_pointer_initializer
    ;

fixed_pointer_initializer
    : '&' variable_reference
    | expression
    ;
```

Each *fixed_pointer_declarator* declares a local variable of the given *pointer_type* and initializes that local variable with the address computed by the corresponding *fixed_pointer_initializer*. A local variable declared in a `fixed` statement is accessible in any *fixed_pointer_initializer*s occurring to the right of that variable's declaration, and in the *embedded_statement* of the `fixed` statement. A local variable declared by a `fixed` statement is considered read-only. A compile-time error occurs if the embedded statement attempts to modify this local variable (via assignment or the `++` and `--` operators) or pass it as a `ref` or `out` parameter.

A *fixed_pointer_initializer* can be one of the following:

- The token "`&`" followed by a *variable_reference* (Precise rules for determining definite assignment) to a moveable variable (Fixed and moveable variables) of an unmanaged type `T`, provided the type `T*` is implicitly convertible to the pointer type given in the `fixed` statement. In this case, the initializer computes the address of the given variable, and the variable is guaranteed to remain at a fixed address for the duration of the `fixed` statement.
- An expression of an *array_type* with elements of an unmanaged type `T`, provided the type `T*` is implicitly convertible to the pointer type given in the `fixed` statement. In this case, the initializer computes the address of the first element in the array, and the entire array is guaranteed to remain at a fixed address for the duration of the `fixed` statement. If the array expression is null or if the array has zero elements, the initializer computes an address equal to zero.

- An expression of type `string`, provided the type `char*` is implicitly convertible to the pointer type given in the `fixed` statement. In this case, the initializer computes the address of the first character in the string, and the entire string is guaranteed to remain at a fixed address for the duration of the `fixed` statement. The behavior of the `fixed` statement is implementation-defined if the string expression is null.
- A *simple_name* or *member_access* that references a fixed size buffer member of a moveable variable, provided the type of the fixed size buffer member is implicitly convertible to the pointer type given in the `fixed` statement. In this case, the initializer computes a pointer to the first element of the fixed size buffer (Fixed size buffers in expressions), and the fixed size buffer is guaranteed to remain at a fixed address for the duration of the `fixed` statement.

For each address computed by a *fixed_pointer_initializer* the `fixed` statement ensures that the variable referenced by the address is not subject to relocation or disposal by the garbage collector for the duration of the `fixed` statement. For example, if the address computed by a *fixed_pointer_initializer* references a field of an object or an element of an array instance, the `fixed` statement guarantees that the containing object instance is not relocated or disposed of during the lifetime of the statement.

It is the programmer's responsibility to ensure that pointers created by `fixed` statements do not survive beyond execution of those statements. For example, when pointers created by `fixed` statements are passed to external APIs, it is the programmer's responsibility to ensure that the APIs retain no memory of these pointers.

Fixed objects may cause fragmentation of the heap (because they can't be moved). For that reason, objects should be fixed only when absolutely necessary and then only for the shortest amount of time possible.

The example

```
class Test
{
    static int x;
    int y;

    unsafe static void F(int* p) {
        *p = 1;
    }

    static void Main() {
        Test t = new Test();
        int[] a = new int[10];
        unsafe {
            fixed (int* p = &x) F(p);
            fixed (int* p = &t.y) F(p);
            fixed (int* p = &a[0]) F(p);
            fixed (int* p = a) F(p);
        }
    }
}
```

demonstrates several uses of the `fixed` statement. The first statement fixes and obtains the address of a static field, the second statement fixes and obtains the address of an instance field, and the third statement fixes and obtains the address of an array element. In each case it would have been an error to use the regular `&` operator since the variables are all classified as moveable variables.

The fourth `fixed` statement in the example above produces a similar result to the third.

This example of the `fixed` statement uses `string`:

```
class Test
{
    static string name = "xx";

    unsafe static void F(char* p) {
        for (int i = 0; p[i] != '\0'; ++i)
            Console.WriteLine(p[i]);
    }

    static void Main() {
        unsafe {
            fixed (char* p = name) F(p);
            fixed (char* p = "xx") F(p);
        }
    }
}
```

In an unsafe context array elements of single-dimensional arrays are stored in increasing index order, starting with index `0` and ending with index `Length - 1`. For multi-dimensional arrays, array elements are stored such that the indices of the rightmost dimension are increased first, then the next left dimension, and so on to the left. Within a `fixed` statement that obtains a pointer `p` to an array instance `a`, the pointer values ranging from `p` to `p + a.Length - 1` represent addresses of the elements in the array. Likewise, the variables ranging from `p[0]` to `p[a.Length - 1]` represent the actual array elements. Given the way in which arrays are stored, we can treat an array of any dimension as though it were linear.

For example:

```
using System;

class Test
{
    static void Main() {
        int[,,] a = new int[2,3,4];
        unsafe {
            fixed (int* p = a) {
                for (int i = 0; i < a.Length; ++i)    // treat as linear
                    p[i] = i;
            }
        }

        for (int i = 0; i < 2; ++i)
            for (int j = 0; j < 3; ++j) {
                for (int k = 0; k < 4; ++k)
                    Console.Write("[{0},{1},{2}] = {3,2} ", i, j, k, a[i,j,k]);
                Console.WriteLine();
            }
    }
}
```

which produces the output:

```
[0,0,0] =  0 [0,0,1] =  1 [0,0,2] =  2 [0,0,3] =  3
[0,1,0] =  4 [0,1,1] =  5 [0,1,2] =  6 [0,1,3] =  7
[0,2,0] =  8 [0,2,1] =  9 [0,2,2] = 10 [0,2,3] = 11
[1,0,0] = 12 [1,0,1] = 13 [1,0,2] = 14 [1,0,3] = 15
[1,1,0] = 16 [1,1,1] = 17 [1,1,2] = 18 [1,1,3] = 19
[1,2,0] = 20 [1,2,1] = 21 [1,2,2] = 22 [1,2,3] = 23
```

In the example

```
class Test
{
    unsafe static void Fill(int* p, int count, int value) {
        for (; count != 0; count--) *p++ = value;
    }

    static void Main() {
        int[] a = new int[100];
        unsafe {
            fixed (int* p = a) Fill(p, 100, -1);
        }
    }
}
```

a `fixed` statement is used to fix an array so its address can be passed to a method that takes a pointer.

In the example:

```
unsafe struct Font
{
    public int size;
    public fixed char name[32];
}

class Test
{
    unsafe static void PutString(string s, char* buffer, int bufSize) {
        int len = s.Length;
        if (len > bufSize) len = bufSize;
        for (int i = 0; i < len; i++) buffer[i] = s[i];
        for (int i = len; i < bufSize; i++) buffer[i] = (char)0;
    }

    Font f;

    unsafe static void Main()
    {
        Test test = new Test();
        test.f.size = 10;
        fixed (char* p = test.f.name) {
            PutString("Times New Roman", p, 32);
        }
    }
}
```

a fixed statement is used to fix a fixed size buffer of a struct so its address can be used as a pointer.

A `char*` value produced by fixing a string instance always points to a null-terminated string. Within a fixed statement that obtains a pointer `p` to a string instance `s`, the pointer values ranging from `p` to `p + s.Length - 1` represent addresses of the characters in the string, and the pointer value `p + s.Length` always points to a null character (the character with value `'\0'`).

Modifying objects of managed type through fixed pointers can results in undefined behavior. For example, because strings are immutable, it is the programmer's responsibility to ensure that the characters referenced by a pointer to a fixed string are not modified.

The automatic null-termination of strings is particularly convenient when calling external APIs that expect "C-style" strings. Note, however, that a string instance is permitted to contain null characters. If such null characters are present, the string will appear truncated when treated as a null-terminated `char*`.

Fixed size buffers

Fixed size buffers are used to declare "C style" in-line arrays as members of structs, and are primarily useful for interfacing with unmanaged APIs.

Fixed size buffer declarations

A ***fixed size buffer*** is a member that represents storage for a fixed length buffer of variables of a given type. A fixed size buffer declaration introduces one or more fixed size buffers of a given element type. Fixed size buffers are only permitted in struct declarations and can only occur in unsafe contexts (Unsafe contexts).

```
struct_member_declaration_unsafe
    : fixed_size_buffer_declaration
    ;

fixed_size_buffer_declaration
    : attributes? fixed_size_buffer_modifier* 'fixed' buffer_element_type fixed_size_buffer_declarator+ ';'
    ;

fixed_size_buffer_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'unsafe'
    ;

buffer_element_type
    : type
    ;

fixed_size_buffer_declarator
    : identifier '[' constant_expression ']'
    ;
```

A fixed size buffer declaration may include a set of attributes (Attributes), a `new` modifier (Modifiers), a valid combination of the four access modifiers (Type parameters and constraints) and an `unsafe` modifier (Unsafe contexts). The attributes and modifiers apply to all of the members declared by the fixed size buffer declaration. It is an error for the same modifier to appear multiple times in a fixed size buffer declaration.

A fixed size buffer declaration is not permitted to include the `static` modifier.

The buffer element type of a fixed size buffer declaration specifies the element type of the buffer(s) introduced by the declaration. The buffer element type must be one of the predefined types `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, or `bool`.

The buffer element type is followed by a list of fixed size buffer declarators, each of which introduces a new member. A fixed size buffer declarator consists of an identifier that names the member, followed by a constant expression enclosed in `[` and `]` tokens. The constant expression denotes the number of elements in the member introduced by that fixed size buffer declarator. The type of the constant expression must be implicitly convertible to type `int`, and the value must be a non-zero positive integer.

The elements of a fixed size buffer are guaranteed to be laid out sequentially in memory.

A fixed size buffer declaration that declares multiple fixed size buffers is equivalent to multiple declarations of a single fixed size buffer declaration with the same attributes, and element types. For example

```
unsafe struct A
{
   public fixed int x[5], y[10], z[100];
}
```

is equivalent to

```
unsafe struct A
{
   public fixed int x[5];
   public fixed int y[10];
   public fixed int z[100];
}
```

Fixed size buffers in expressions

Member lookup (Operators) of a fixed size buffer member proceeds exactly like member lookup of a field.

A fixed size buffer can be referenced in an expression using a *simple_name* (Type inference) or a *member_access* (Compile-time checking of dynamic overload resolution).

When a fixed size buffer member is referenced as a simple name, the effect is the same as a member access of the form `this.I`, where `I` is the fixed size buffer member.

In a member access of the form `E.I`, if `E` is of a struct type and a member lookup of `I` in that struct type identifies a fixed size member, then `E.I` is evaluated an classified as follows:

- If the expression `E.I` does not occur in an unsafe context, a compile-time error occurs.
- If `E` is classified as a value, a compile-time error occurs.
- Otherwise, if `E` is a moveable variable (Fixed and moveable variables) and the expression `E.I` is not a *fixed_pointer_initializer* (The fixed statement), a compile-time error occurs.
- Otherwise, `E` references a fixed variable and the result of the expression is a pointer to the first element of the fixed size buffer member `I` in `E`. The result is of type `S*`, where `S` is the element type of `I`, and is classified as a value.

The subsequent elements of the fixed size buffer can be accessed using pointer operations from the first element. Unlike access to arrays, access to the elements of a fixed size buffer is an unsafe operation and is not range checked.

The following example declares and uses a struct with a fixed size buffer member.

```
unsafe struct Font
{
    public int size;
    public fixed char name[32];
}

class Test
{
    unsafe static void PutString(string s, char* buffer, int bufSize) {
        int len = s.Length;
        if (len > bufSize) len = bufSize;
        for (int i = 0; i < len; i++) buffer[i] = s[i];
        for (int i = len; i < bufSize; i++) buffer[i] = (char)0;
    }

    unsafe static void Main()
    {
        Font f;
        f.size = 10;
        PutString("Times New Roman", f.name, 32);
    }
}
```

Definite assignment checking

Fixed size buffers are not subject to definite assignment checking (Definite assignment), and fixed size buffer members are ignored for purposes of definite assignment checking of struct type variables.

When the outermost containing struct variable of a fixed size buffer member is a static variable, an instance variable of a class instance, or an array element, the elements of the fixed size buffer are automatically initialized to their default values (Default values). In all other cases, the initial content of a fixed size buffer is undefined.

Stack allocation

In an unsafe context, a local variable declaration (Local variable declarations) may include a stack allocation initializer which allocates memory from the call stack.

```
local_variable_initializer_unsafe
    : stackalloc_initializer
    ;

stackalloc_initializer
    : 'stackalloc' unmanaged_type '[' expression ']'
    ;
```

The *unmanaged_type* indicates the type of the items that will be stored in the newly allocated location, and the *expression* indicates the number of these items. Taken together, these specify the required allocation size. Since the size of a stack allocation cannot be negative, it is a compile-time error to specify the number of items as a *constant_expression* that evaluates to a negative value.

A stack allocation initializer of the form `stackalloc T[E]` requires `T` to be an unmanaged type (Pointer types) and `E` to be an expression of type `int`. The construct allocates `E * sizeof(T)` bytes from the call stack and returns a pointer, of type `T*`, to the newly allocated block. If `E` is a negative value, then the behavior is undefined. If `E` is zero, then no allocation is made, and the pointer returned is implementation-defined. If there is not enough memory available to allocate a block of the given size, a `System.StackOverflowException` is thrown.

The content of the newly allocated memory is undefined.

Stack allocation initializers are not permitted in `catch` or `finally` blocks (The try statement).

There is no way to explicitly free memory allocated using `stackalloc`. All stack allocated memory blocks created during the execution of a function member are automatically discarded when that function member returns. This corresponds to the `alloca` function, an extension commonly found in C and C++ implementations.

In the example

```
using System;

class Test
{
    static string IntToString(int value) {
        int n = value >= 0? value: -value;
        unsafe {
            char* buffer = stackalloc char[16];
            char* p = buffer + 16;
            do {
                *--p = (char)(n % 10 + '0');
                n /= 10;
            } while (n != 0);
            if (value < 0) *--p = '-';
            return new string(p, 0, (int)(buffer + 16 - p));
        }
    }

    static void Main() {
        Console.WriteLine(IntToString(12345));
        Console.WriteLine(IntToString(-999));
    }
}
```

a `stackalloc` initializer is used in the `IntToString` method to allocate a buffer of 16 characters on the stack. The buffer is automatically discarded when the method returns.

Dynamic memory allocation

Except for the `stackalloc` operator, C# provides no predefined constructs for managing non-garbage collected memory. Such services are typically provided by supporting class libraries or imported directly from the underlying operating system. For example, the `Memory` class below illustrates how the heap functions of an underlying operating system might be accessed from C#:

```
using System;
using System.Runtime.InteropServices;

public static unsafe class Memory
{
    // Handle for the process heap. This handle is used in all calls to the
    // HeapXXX APIs in the methods below.
    private static readonly IntPtr s_heap = GetProcessHeap();

    // Allocates a memory block of the given size. The allocated memory is
    // automatically initialized to zero.
    public static void* Alloc(int size)
    {
        void* result = HeapAlloc(s_heap, HEAP_ZERO_MEMORY, (UIntPtr)size);
        if (result == null) throw new OutOfMemoryException();
        return result;
    }

    // Copies count bytes from src to dst. The source and destination
    // blocks are permitted to overlap.
    public static void Copy(void* src, void* dst, int count)
    {
        byte* ps = (byte*)src;
        byte* pd = (byte*)dst;
        if (ps > pd)
        {
            for (; count != 0; count--) *pd++ = *ps++;
        }
        else if (ps < pd)
        {
```

```
        {
            for (ps += count, pd += count; count != 0; count--) *--pd = *--ps;
        }
    }

    // Frees a memory block.
    public static void Free(void* block)
    {
        if (!HeapFree(s_heap, 0, block)) throw new InvalidOperationException();
    }

    // Re-allocates a memory block. If the reallocation request is for a
    // larger size, the additional region of memory is automatically
    // initialized to zero.
    public static void* ReAlloc(void* block, int size)
    {
        void* result = HeapReAlloc(s_heap, HEAP_ZERO_MEMORY, block, (UIntPtr)size);
        if (result == null) throw new OutOfMemoryException();
        return result;
    }

    // Returns the size of a memory block.
    public static int SizeOf(void* block)
    {
        int result = (int)HeapSize(s_heap, 0, block);
        if (result == -1) throw new InvalidOperationException();
        return result;
    }

    // Heap API flags
    private const int HEAP_ZERO_MEMORY = 0x00000008;

    // Heap API functions
    [DllImport("kernel32")]
    private static extern IntPtr GetProcessHeap();

    [DllImport("kernel32")]
    private static extern void* HeapAlloc(IntPtr hHeap, int flags, UIntPtr size);

    [DllImport("kernel32")]
    private static extern bool HeapFree(IntPtr hHeap, int flags, void* block);

    [DllImport("kernel32")]
    private static extern void* HeapReAlloc(IntPtr hHeap, int flags, void* block, UIntPtr size);

    [DllImport("kernel32")]
    private static extern UIntPtr HeapSize(IntPtr hHeap, int flags, void* block);
}
```

An example that uses the `Memory` class is given below:

```
class Test
{
    static unsafe void Main()
    {
        byte* buffer = null;
        try
        {
            const int Size = 256;
            buffer = (byte*)Memory.Alloc(Size);
            for (int i = 0; i < Size; i++) buffer[i] = (byte)i;
            byte[] array = new byte[Size];
            fixed (byte* p = array) Memory.Copy(buffer, p, Size);
            for (int i = 0; i < Size; i++) Console.WriteLine(array[i]);
        }
        finally
        {
            if (buffer != null) Memory.Free(buffer);
        }
    }
}
```

The example allocates 256 bytes of memory through `Memory.Alloc` and initializes the memory block with values increasing from 0 to 255. It then allocates a 256 element byte array and uses `Memory.Copy` to copy the contents of the memory block into the byte array. Finally, the memory block is freed using `Memory.Free` and the contents of the byte array are output on the console.

Documentation comments

18.12.2021 • 22 minutes to read

C# provides a mechanism for programmers to document their code using a special comment syntax that contains XML text. In source code files, comments having a certain form can be used to direct a tool to produce XML from those comments and the source code elements, which they precede. Comments using such syntax are called ***documentation comments***. They must immediately precede a user-defined type (such as a class, delegate, or interface) or a member (such as a field, event, property, or method). The XML generation tool is called the ***documentation generator***. (This generator could be, but need not be, the C# compiler itself.) The output produced by the documentation generator is called the ***documentation file***. A documentation file is used as input to a ***documentation viewer***; a tool intended to produce some sort of visual display of type information and its associated documentation.

This specification suggests a set of tags to be used in documentation comments, but use of these tags is not required, and other tags may be used if desired, as long the rules of well-formed XML are followed.

Introduction

Comments having a special form can be used to direct a tool to produce XML from those comments and the source code elements, which they precede. Such comments are single-line comments that start with three slashes (`///`), or delimited comments that start with a slash and two stars (`/**`). They must immediately precede a user-defined type (such as a class, delegate, or interface) or a member (such as a field, event, property, or method) that they annotate. Attribute sections (Attribute specification) are considered part of declarations, so documentation comments must precede attributes applied to a type or member.

Syntax:

```
single_line_doc_comment
    : '///' input_character*
    ;

delimited_doc_comment
    : '/**' delimited_comment_section* asterisk+ '/'
    ;
```

In a *single_line_doc_comment*, if there is a *whitespace* character following the `///` characters on each of the *single_line_doc_comment*s adjacent to the current *single_line_doc_comment*, then that *whitespace* character is not included in the XML output.

In a delimited-doc-comment, if the first non-whitespace character on the second line is an asterisk and the same pattern of optional whitespace characters and an asterisk character is repeated at the beginning of each of the line within the delimited-doc-comment, then the characters of the repeated pattern are not included in the XML output. The pattern may include whitespace characters after, as well as before, the asterisk character.

Example:

```
/// <summary>Class <c>Point</c> models a point in a two-dimensional
/// plane.</summary>
///
public class Point
{
    /// <summary>method <c>draw</c> renders the point.</summary>
    void draw() {...}
}
```

The text within documentation comments must be well formed according to the rules of XML (https://www.w3.org/TR/REC-xml). If the XML is ill formed, a warning is generated and the documentation file will contain a comment saying that an error was encountered.

Although developers are free to create their own set of tags, a recommended set is defined in Recommended tags. Some of the recommended tags have special meanings:

- The `<param>` tag is used to describe parameters. If such a tag is used, the documentation generator must verify that the specified parameter exists and that all parameters are described in documentation comments. If such verification fails, the documentation generator issues a warning.
- The `cref` attribute can be attached to any tag to provide a reference to a code element. The documentation generator must verify that this code element exists. If the verification fails, the documentation generator issues a warning. When looking for a name described in a `cref` attribute, the documentation generator must respect namespace visibility according to `using` statements appearing within the source code. For code elements that are generic, the normal generic syntax (that is, "`List<T>`") cannot be used because it produces invalid XML. Braces can be used instead of brackets (that is, "`List{T}`"), or the XML escape syntax can be used (that is, "`List<T>`").
- The `<summary>` tag is intended to be used by a documentation viewer to display additional information about a type or member.
- The `<include>` tag includes information from an external XML file.

Note carefully that the documentation file does not provide full information about the type and members (for example, it does not contain any type information). To get such information about a type or member, the documentation file must be used in conjunction with reflection on the actual type or member.

Recommended tags

The documentation generator must accept and process any tag that is valid according to the rules of XML. The following tags provide commonly used functionality in user documentation. (Of course, other tags are possible.)

TAG	SECTION	PURPOSE
`<c>`	`<c>`	Set text in a code-like font
`<code>`	`<code>`	Set one or more lines of source code or program output
`<example>`	`<example>`	Indicate an example
`<exception>`	`<exception>`	Identifies the exceptions a method can throw
`<include>`	`<include>`	Includes XML from an external file

TAG	SECTION	PURPOSE
`<list>`	`<list>`	Create a list or table
`<para>`	`<para>`	Permit structure to be added to text
`<param>`	`<param>`	Describe a parameter for a method or constructor
`<paramref>`	`<paramref>`	Identify that a word is a parameter name
`<permission>`	`<permission>`	Document the security accessibility of a member
`<remarks>`	`<remarks>`	Describe additional information about a type
`<returns>`	`<returns>`	Describe the return value of a method
`<see>`	`<see>`	Specify a link
`<seealso>`	`<seealso>`	Generate a See Also entry
`<summary>`	`<summary>`	Describe a type or a member of a type
`<value>`	`<value>`	Describe a property
`<typeparam>`		Describe a generic type parameter
`<typeparamref>`		Identify that a word is a type parameter name

`<c>`

This tag provides a mechanism to indicate that a fragment of text within a description should be set in a special font such as that used for a block of code. For lines of actual code, use `<code>` (`<code>`).

Syntax:

```
<c>text</c>
```

Example:

```
/// <summary>Class <c>Point</c> models a point in a two-dimensional
/// plane.</summary>

public class Point
{
    // ...
}
```

`<code>`

This tag is used to set one or more lines of source code or program output in some special font. For small code

fragments in narrative, use `<c>` (`<c>`).

Syntax:

```
<code>source code or program output</code>
```

Example:

```
/// <summary>This method changes the point's location by
///    the given x- and y-offsets.
/// <example>For example:
/// <code>
///    Point p = new Point(3,5);
///    p.Translate(-1,3);
/// </code>
/// results in <c>p</c>'s having the value (2,8).
/// </example>
/// </summary>

public void Translate(int xor, int yor) {
    X += xor;
    Y += yor;
}
```

`<example>`

This tag allows example code within a comment, to specify how a method or other library member may be used. Ordinarily, this would also involve use of the tag `<code>` (`<code>`) as well.

Syntax:

```
<example>description</example>
```

Example:

See `<code>` (`<code>`) for an example.

`<exception>`

This tag provides a way to document the exceptions a method can throw.

Syntax:

```
<exception cref="member">description</exception>
```

where

- `member` is the name of a member. The documentation generator checks that the given member exists and translates `member` to the canonical element name in the documentation file.
- `description` is a description of the circumstances in which the exception is thrown.

Example:

```
public class DataBaseOperations
{
    /// <exception cref="MasterFileFormatCorruptException"></exception>
    /// <exception cref="MasterFileLockedOpenException"></exception>
    public static void ReadRecord(int flag) {
        if (flag == 1)
            throw new MasterFileFormatCorruptException();
        else if (flag == 2)
            throw new MasterFileLockedOpenException();
        // ...
    }
}
```

`<include>`

This tag allows including information from an XML document that is external to the source code file. The external file must be a well-formed XML document, and an XPath expression is applied to that document to specify what XML from that document to include. The `<include>` tag is then replaced with the selected XML from the external document.

Syntax:

```
<include file="filename" path="xpath" />
```

where

- `filename` is the file name of an external XML file. The file name is interpreted relative to the file that contains the include tag.
- `xpath` is an XPath expression that selects some of the XML in the external XML file.

Example:

If the source code contained a declaration like:

```
/// <include file="docs.xml" path='extradoc/class[@name="IntList"]/*' />
public class IntList { ... }
```

and the external file "docs.xml" had the following contents:

```
<?xml version="1.0"?>
<extradoc>
  <class name="IntList">
     <summary>
        Contains a list of integers.
     </summary>
  </class>
  <class name="StringList">
     <summary>
        Contains a list of integers.
     </summary>
  </class>
</extradoc>
```

then the same documentation is output as if the source code contained:

```
/// <summary>
///    Contains a list of integers.
/// </summary>
public class IntList { ... }
```

`<list>`

This tag is used to create a list or table of items. It may contain a `<listheader>` block to define the heading row of either a table or definition list. (When defining a table, only an entry for `term` in the heading need be supplied.)

Each item in the list is specified with an `<item>` block. When creating a definition list, both `term` and `description` must be specified. However, for a table, bulleted list, or numbered list, only `description` need be specified.

Syntax:

```
<list type="bullet" | "number" | "table">
   <listheader>
      <term>term</term>
      <description>*description*</description>
   </listheader>
   <item>
      <term>term</term>
      <description>*description*</description>
   </item>
    ...
   <item>
      <term>term</term>
      <description>description</description>
   </item>
</list>
```

where

- `term` is the term to define, whose definition is in `description`.
- `description` is either an item in a bullet or numbered list, or the definition of a `term`.

Example:

```
public class MyClass
{
    /// <summary>Here is an example of a bulleted list:
    /// <list type="bullet">
    /// <item>
    /// <description>Item 1.</description>
    /// </item>
    /// <item>
    /// <description>Item 2.</description>
    /// </item>
    /// </list>
    /// </summary>
    public static void Main () {
        // ...
    }
}
```

`<para>`

This tag is for use inside other tags, such as `<summary>` (`<remarks>`) or `<returns>` (`<returns>`), and permits

structure to be added to text.

Syntax:

```
<para>content</para>
```

where `content` is the text of the paragraph.

Example:

```
/// <summary>This is the entry point of the Point class testing program.
/// <para>This program tests each method and operator, and
/// is intended to be run after any non-trivial maintenance has
/// been performed on the Point class.</para></summary>
public static void Main() {
    // ...
}
```

`<param>`

This tag is used to describe a parameter for a method, constructor, or indexer.

Syntax:

```
<param name="name">description</param>
```

where

- `name` is the name of the parameter.
- `description` is a description of the parameter.

Example:

```
/// <summary>This method changes the point's location to
///    the given coordinates.</summary>
/// <param name="xor">the new x-coordinate.</param>
/// <param name="yor">the new y-coordinate.</param>
public void Move(int xor, int yor) {
    X = xor;
    Y = yor;
}
```

`<paramref>`

This tag is used to indicate that a word is a parameter. The documentation file can be processed to format this parameter in some distinct way.

Syntax:

```
<paramref name="name"/>
```

where `name` is the name of the parameter.

Example:

```
/// <summary>This constructor initializes the new Point to
///    (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
/// <param name="xor">the new Point's x-coordinate.</param>
/// <param name="yor">the new Point's y-coordinate.</param>

public Point(int xor, int yor) {
    X = xor;
    Y = yor;
}
```

`<permission>`

This tag allows the security accessibility of a member to be documented.

Syntax:

```
<permission cref="member">description</permission>
```

where

- `member` is the name of a member. The documentation generator checks that the given code element exists and translates *member* to the canonical element name in the documentation file.
- `description` is a description of the access to the member.

Example:

```
/// <permission cref="System.Security.PermissionSet">Everyone can
/// access this method.</permission>

public static void Test() {
    // ...
}
```

`<remarks>`

This tag is used to specify extra information about a type. (Use `<summary>` (`<summary>`) to describe the type itself and the members of a type.)

Syntax:

```
<remarks>description</remarks>
```

where `description` is the text of the remark.

Example:

```
/// <summary>Class <c>Point</c> models a point in a
/// two-dimensional plane.</summary>
/// <remarks>Uses polar coordinates</remarks>
public class Point
{
    // ...
}
```

`<returns>`

This tag is used to describe the return value of a method.

Syntax:

```
<returns>description</returns>
```

where `description` is a description of the return value.

Example:

```
/// <summary>Report a point's location as a string.</summary>
/// <returns>A string representing a point's location, in the form (x,y),
///    without any leading, trailing, or embedded whitespace.</returns>
public override string ToString() {
    return "(" + X + "," + Y + ")";
}
```

`<see>`

This tag allows a link to be specified within text. Use `<seealso>` (`<seealso>`) to indicate text that is to appear in a See Also section.

Syntax:

```
<see cref="member"/>
```

where `member` is the name of a member. The documentation generator checks that the given code element exists and changes *member* to the element name in the generated documentation file.

Example:

```
/// <summary>This method changes the point's location to
///    the given coordinates.</summary>
/// <see cref="Translate"/>
public void Move(int xor, int yor) {
    X = xor;
    Y = yor;
}

/// <summary>This method changes the point's location by
///    the given x- and y-offsets.
/// </summary>
/// <see cref="Move"/>
public void Translate(int xor, int yor) {
    X += xor;
    Y += yor;
}
```

`<seealso>`

This tag allows an entry to be generated for the See Also section. Use `<see>` (`<see>`) to specify a link from within text.

Syntax:

```
<seealso cref="member"/>
```

where `member` is the name of a member. The documentation generator checks that the given code element exists and changes *member* to the element name in the generated documentation file.

Example:

```
/// <summary>This method determines whether two Points have the same
///    location.</summary>
/// <seealso cref="operator=="/>
/// <seealso cref="operator!="/>
public override bool Equals(object o) {
    // ...
}
```

`<summary>`

This tag can be used to describe a type or a member of a type. Use `<remarks>` (`<remarks>`) to describe the type itself.

Syntax:

```
<summary>description</summary>
```

where `description` is a summary of the type or member.

Example:

```
/// <summary>This constructor initializes the new Point to (0,0).</summary>
public Point() : this(0,0) {
}
```

`<value>`

This tag allows a property to be described.

Syntax:

```
<value>property description</value>
```

where `property description` is a description for the property.

Example:

```
/// <value>Property <c>X</c> represents the point's x-coordinate.</value>
public int X
{
    get { return x; }
    set { x = value; }
}
```

`<typeparam>`

This tag is used to describe a generic type parameter for a class, struct, interface, delegate, or method.

Syntax:

```
<typeparam name="name">description</typeparam>
```

where `name` is the name of the type parameter, and `description` is its description.

Example:

```
/// <summary>A generic list class.</summary>
/// <typeparam name="T">The type stored by the list.</typeparam>
public class MyList<T> {
    ...
}
```

`<typeparamref>`

This tag is used to indicate that a word is a type parameter. The documentation file can be processed to format this type parameter in some distinct way.

Syntax:

```
<typeparamref name="name"/>
```

where `name` is the name of the type parameter.

Example:

```
/// <summary>This method fetches data and returns a list of <typeparamref name="T"/>.</summary>
/// <param name="query">query to execute</param>
public List<T> FetchData<T>(string query) {
    ...
}
```

Processing the documentation file

The documentation generator generates an ID string for each element in the source code that is tagged with a documentation comment. This ID string uniquely identifies a source element. A documentation viewer can use an ID string to identify the corresponding metadata/reflection item to which the documentation applies.

The documentation file is not a hierarchical representation of the source code; rather, it is a flat list with a generated ID string for each element.

ID string format

The documentation generator observes the following rules when it generates the ID strings:

- No white space is placed in the string.
- The first part of the string identifies the kind of member being documented, via a single character followed by a colon. The following kinds of members are defined:

CHARACTER	DESCRIPTION
E	Event
F	Field
M	Method (including constructors, destructors, and operators)
N	Namespace
P	Property (including indexers)

CHARACTER	DESCRIPTION
T	Type (such as class, delegate, enum, interface, and struct)
!	Error string; the rest of the string provides information about the error. For example, the documentation generator generates error information for links that cannot be resolved.

- The second part of the string is the fully qualified name of the element, starting at the root of the namespace. The name of the element, its enclosing type(s), and namespace are separated by periods. If the name of the item itself has periods, they are replaced by `#(U+0023)` characters. (It is assumed that no element has this character in its name.)
- For methods and properties with arguments, the argument list follows, enclosed in parentheses. For those without arguments, the parentheses are omitted. The arguments are separated by commas. The encoding of each argument is the same as a CLI signature, as follows:
  - Arguments are represented by their documentation name, which is based on their fully qualified name, modified as follows:
    - Arguments that represent generic types have an appended `` ` `` (backtick) character followed by the number of type parameters
    - Arguments having the `out` or `ref` modifier have an `@` following their type name. Arguments passed by value or via `params` have no special notation.
    - Arguments that are arrays are represented as `[lowerbound:size, ... , lowerbound:size]` where the number of commas is the rank less one, and the lower bounds and size of each dimension, if known, are represented in decimal. If a lower bound or size is not specified, it is omitted. If the lower bound and size for a particular dimension are omitted, the `:` is omitted as well. Jagged arrays are represented by one `[]` per level.
    - Arguments that have pointer types other than void are represented using a `*` following the type name. A void pointer is represented using a type name of `System.Void`.
    - Arguments that refer to generic type parameters defined on types are encoded using the `` ` `` (backtick) character followed by the zero-based index of the type parameter.
    - Arguments that use generic type parameters defined in methods use a double-backtick `` `` `` instead of the `` ` `` used for types.
    - Arguments that refer to constructed generic types are encoded using the generic type, followed by `{`, followed by a comma-separated list of type arguments, followed by `}`.

ID string examples

The following examples each show a fragment of C# code, along with the ID string produced from each source element capable of having a documentation comment:

- Types are represented using their fully qualified name, augmented with generic information:

```
enum Color { Red, Blue, Green }

namespace Acme
{
    interface IProcess {...}

    struct ValueType {...}

    class Widget: IProcess
    {
        public class NestedClass {...}
        public interface IMenuItem {...}
        public delegate void Del(int i);
        public enum Direction { North, South, East, West }
    }

    class MyList<T>
    {
        class Helper<U,V> {...}
    }
}

"T:Color"
"T:Acme.IProcess"
"T:Acme.ValueType"
"T:Acme.Widget"
"T:Acme.Widget.NestedClass"
"T:Acme.Widget.IMenuItem"
"T:Acme.Widget.Del"
"T:Acme.Widget.Direction"
"T:Acme.MyList`1"
"T:Acme.MyList`1.Helper`2"
```

- Fields are represented by their fully qualified name:

```
namespace Acme
{
    struct ValueType
    {
        private int total;
    }

    class Widget: IProcess
    {
        public class NestedClass
        {
            private int value;
        }

        private string message;
        private static Color defaultColor;
        private const double PI = 3.14159;
        protected readonly double monthlyAverage;
        private long[] array1;
        private Widget[,] array2;
        private unsafe int *pCount;
        private unsafe float **ppValues;
    }
}

"F:Acme.ValueType.total"
"F:Acme.Widget.NestedClass.value"
"F:Acme.Widget.message"
"F:Acme.Widget.defaultColor"
"F:Acme.Widget.PI"
"F:Acme.Widget.monthlyAverage"
"F:Acme.Widget.array1"
"F:Acme.Widget.array2"
"F:Acme.Widget.pCount"
"F:Acme.Widget.ppValues"
```

- Constructors.

```
namespace Acme
{
    class Widget: IProcess
    {
        static Widget() {...}
        public Widget() {...}
        public Widget(string s) {...}
    }
}

"M:Acme.Widget.#cctor"
"M:Acme.Widget.#ctor"
"M:Acme.Widget.#ctor(System.String)"
```

- Destructors.

```
namespace Acme
{
    class Widget: IProcess
    {
        ~Widget() {...}
    }
}

"M:Acme.Widget.Finalize"
```

- Methods.

```
namespace Acme
{
    struct ValueType
    {
        public void M(int i) {...}
    }

    class Widget: IProcess
    {
        public class NestedClass
        {
            public void M(int i) {...}
        }

        public static void M0() {...}
        public void M1(char c, out float f, ref ValueType v) {...}
        public void M2(short[] x1, int[,] x2, long[][] x3) {...}
        public void M3(long[][] x3, Widget[][,,] x4) {...}
        public unsafe void M4(char *pc, Color **pf) {...}
        public unsafe void M5(void *pv, double *[][,] pd) {...}
        public void M6(int i, params object[] args) {...}
    }

    class MyList<T>
    {
        public void Test(T t) { }
    }

    class UseList
    {
        public void Process(MyList<int> list) { }
        public MyList<T> GetValues<T>(T inputValue) { return null; }
    }
}

"M:Acme.ValueType.M(System.Int32)"
"M:Acme.Widget.NestedClass.M(System.Int32)"
"M:Acme.Widget.M0"
"M:Acme.Widget.M1(System.Char,System.Single@,Acme.ValueType@)"
"M:Acme.Widget.M2(System.Int16[],System.Int32[0:,0:],System.Int64[][])"
"M:Acme.Widget.M3(System.Int64[][],Acme.Widget[0:,0:,0:][])"
"M:Acme.Widget.M4(System.Char*,Color**)"
"M:Acme.Widget.M5(System.Void*,System.Double*[0:,0:][])"
"M:Acme.Widget.M6(System.Int32,System.Object[])"
"M:Acme.MyList`1.Test(`0)"
"M:Acme.UseList.Process(Acme.MyList{System.Int32})"
"M:Acme.UseList.GetValues``(``0)"
```

- Properties and indexers.

```
namespace Acme
{
    class Widget: IProcess
    {
        public int Width { get {...} set {...} }
        public int this[int i] { get {...} set {...} }
        public int this[string s, int i] { get {...} set {...} }
    }
}

"P:Acme.Widget.Width"
"P:Acme.Widget.Item(System.Int32)"
"P:Acme.Widget.Item(System.String,System.Int32)"
```

- Events.

```
namespace Acme
{
    class Widget: IProcess
    {
        public event Del AnEvent;
    }
}

"E:Acme.Widget.AnEvent"
```

- Unary operators.

```
namespace Acme
{
    class Widget: IProcess
    {
        public static Widget operator+(Widget x) {...}
    }
}

"M:Acme.Widget.op_UnaryPlus(Acme.Widget)"
```

  The complete set of unary operator function names used is as follows: `op_UnaryPlus`, `op_UnaryNegation`, `op_LogicalNot`, `op_OnesComplement`, `op_Increment`, `op_Decrement`, `op_True`, and `op_False`.

- Binary operators.

```
namespace Acme
{
    class Widget: IProcess
    {
        public static Widget operator+(Widget x1, Widget x2) {...}
    }
}

"M:Acme.Widget.op_Addition(Acme.Widget,Acme.Widget)"
```

  The complete set of binary operator function names used is as follows: `op_Addition`, `op_Subtraction`, `op_Multiply`, `op_Division`, `op_Modulus`, `op_BitwiseAnd`, `op_BitwiseOr`, `op_ExclusiveOr`, `op_LeftShift`, `op_RightShift`, `op_Equality`, `op_Inequality`, `op_LessThan`, `op_LessThanOrEqual`, `op_GreaterThan`, and `op_GreaterThanOrEqual`.

- Conversion operators have a trailing "`~`" followed by the return type.

```
namespace Acme
{
    class Widget: IProcess
    {
        public static explicit operator int(Widget x) {...}
        public static implicit operator long(Widget x) {...}
    }
}

"M:Acme.Widget.op_Explicit(Acme.Widget)~System.Int32"
"M:Acme.Widget.op_Implicit(Acme.Widget)~System.Int64"
```

An example

C# source code

The following example shows the source code of a `Point` class:

```
namespace Graphics
{

/// <summary>Class <c>Point</c> models a point in a two-dimensional plane.
/// </summary>
public class Point
{

    /// <summary>Instance variable <c>x</c> represents the point's
    ///    x-coordinate.</summary>
    private int x;

    /// <summary>Instance variable <c>y</c> represents the point's
    ///    y-coordinate.</summary>
    private int y;

    /// <value>Property <c>X</c> represents the point's x-coordinate.</value>
    public int X
    {
        get { return x; }
        set { x = value; }
    }

    /// <value>Property <c>Y</c> represents the point's y-coordinate.</value>
    public int Y
    {
        get { return y; }
        set { y = value; }
    }

    /// <summary>This constructor initializes the new Point to
    ///    (0,0).</summary>
    public Point() : this(0,0) {}

    /// <summary>This constructor initializes the new Point to
    ///    (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
    /// <param><c>xor</c> is the new Point's x-coordinate.</param>
    /// <param><c>yor</c> is the new Point's y-coordinate.</param>
    public Point(int xor, int yor) {
        X = xor;
        Y = yor;
    }

    /// <summary>This method changes the point's location to
    ///    the given coordinates.</summary>
    /// <param><c>xor</c> is the new x-coordinate.</param>
    /// <param><c>yor</c> is the new y-coordinate.</param>
```

```
    /// <see cref="Translate"/>
    public void Move(int xor, int yor) {
        X = xor;
        Y = yor;
    }

    /// <summary>This method changes the point's location by
    ///    the given x- and y-offsets.
    /// <example>For example:
    /// <code>
    ///    Point p = new Point(3,5);
    ///    p.Translate(-1,3);
    /// </code>
    /// results in <c>p</c>'s having the value (2,8).
    /// </example>
    /// </summary>
    /// <param><c>xor</c> is the relative x-offset.</param>
    /// <param><c>yor</c> is the relative y-offset.</param>
    /// <see cref="Move"/>
    public void Translate(int xor, int yor) {
        X += xor;
        Y += yor;
    }

    /// <summary>This method determines whether two Points have the same
    ///    location.</summary>
    /// <param><c>o</c> is the object to be compared to the current object.
    /// </param>
    /// <returns>True if the Points have the same location and they have
    ///    the exact same type; otherwise, false.</returns>
    /// <seealso cref="operator=="/>
    /// <seealso cref="operator!="/>
    public override bool Equals(object o) {
        if (o == null) {
            return false;
        }

        if (this == o) {
            return true;
        }

        if (GetType() == o.GetType()) {
            Point p = (Point)o;
            return (X == p.X) && (Y == p.Y);
        }
        return false;
    }

    /// <summary>Report a point's location as a string.</summary>
    /// <returns>A string representing a point's location, in the form (x,y),
    ///    without any leading, training, or embedded whitespace.</returns>
    public override string ToString() {
        return "(" + X + "," + Y + ")";
    }

    /// <summary>This operator determines whether two Points have the same
    ///    location.</summary>
    /// <param><c>p1</c> is the first Point to be compared.</param>
    /// <param><c>p2</c> is the second Point to be compared.</param>
    /// <returns>True if the Points have the same location and they have
    ///    the exact same type; otherwise, false.</returns>
    /// <seealso cref="Equals"/>
    /// <seealso cref="operator!="/>
    public static bool operator==(Point p1, Point p2) {
        if ((object)p1 == null || (object)p2 == null) {
            return false;
        }

        if (p1.GetType() == p2.GetType()) {
```

```
            return (p1.X == p2.X) && (p1.Y == p2.Y);
        }

        return false;
    }

    /// <summary>This operator determines whether two Points have the same
    ///    location.</summary>
    /// <param><c>p1</c> is the first Point to be compared.</param>
    /// <param><c>p2</c> is the second Point to be compared.</param>
    /// <returns>True if the Points do not have the same location and the
    ///    exact same type; otherwise, false.</returns>
    /// <seealso cref="Equals"/>
    /// <seealso cref="operator=="/>
    public static bool operator!=(Point p1, Point p2) {
        return !(p1 == p2);
    }

    /// <summary>This is the entry point of the Point class testing
    /// program.
    /// <para>This program tests each method and operator, and
    /// is intended to be run after any non-trivial maintenance has
    /// been performed on the Point class.</para></summary>
    public static void Main() {
        // class test code goes here
    }
}
}
```

Resulting XML

Here is the output produced by one documentation generator when given the source code for class `Point`, shown above:

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>Point</name>
    </assembly>
    <members>
        <member name="T:Graphics.Point">
            <summary>Class <c>Point</c> models a point in a two-dimensional
            plane.
            </summary>
        </member>

        <member name="F:Graphics.Point.x">
            <summary>Instance variable <c>x</c> represents the point's
            x-coordinate.</summary>
        </member>

        <member name="F:Graphics.Point.y">
            <summary>Instance variable <c>y</c> represents the point's
            y-coordinate.</summary>
        </member>

        <member name="M:Graphics.Point.#ctor">
            <summary>This constructor initializes the new Point to
        (0,0).</summary>
        </member>

        <member name="M:Graphics.Point.#ctor(System.Int32,System.Int32)">
            <summary>This constructor initializes the new Point to
            (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
            <param><c>xor</c> is the new Point's x-coordinate.</param>
            <param><c>yor</c> is the new Point's y-coordinate.</param>
        </member>
```

```
    <member name="M:Graphics.Point.Move(System.Int32,System.Int32)">
        <summary>This method changes the point's location to
        the given coordinates.</summary>
        <param><c>xor</c> is the new x-coordinate.</param>
        <param><c>yor</c> is the new y-coordinate.</param>
        <see cref="M:Graphics.Point.Translate(System.Int32,System.Int32)"/>
    </member>

    <member
        name="M:Graphics.Point.Translate(System.Int32,System.Int32)">
        <summary>This method changes the point's location by
        the given x- and y-offsets.
        <example>For example:
        <code>
        Point p = new Point(3,5);
        p.Translate(-1,3);
        </code>
        results in <c>p</c>'s having the value (2,8).
        </example>
        </summary>
        <param><c>xor</c> is the relative x-offset.</param>
        <param><c>yor</c> is the relative y-offset.</param>
        <see cref="M:Graphics.Point.Move(System.Int32,System.Int32)"/>
    </member>

    <member name="M:Graphics.Point.Equals(System.Object)">
        <summary>This method determines whether two Points have the same
        location.</summary>
        <param><c>o</c> is the object to be compared to the current
        object.
        </param>
        <returns>True if the Points have the same location and they have
        the exact same type; otherwise, false.</returns>
        <seealso
  cref="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)"/>
        <seealso
  cref="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)"/>
    </member>

    <member name="M:Graphics.Point.ToString">
        <summary>Report a point's location as a string.</summary>
        <returns>A string representing a point's location, in the form
        (x,y),
        without any leading, training, or embedded whitespace.</returns>
    </member>

    <member
   name="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)">
        <summary>This operator determines whether two Points have the
        same
        location.</summary>
        <param><c>p1</c> is the first Point to be compared.</param>
        <param><c>p2</c> is the second Point to be compared.</param>
        <returns>True if the Points have the same location and they have
        the exact same type; otherwise, false.</returns>
        <seealso cref="M:Graphics.Point.Equals(System.Object)"/>
        <seealso
 cref="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)"/>
    </member>

    <member
  name="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)">
        <summary>This operator determines whether two Points have the
        same
        location.</summary>
        <param><c>p1</c> is the first Point to be compared.</param>
        <param><c>p2</c> is the second Point to be compared.</param>
        <returns>True if the Points do not have the same location and
```

```
            the
            exact same type; otherwise, false.</returns>
            <seealso cref="M:Graphics.Point.Equals(System.Object)"/>
            <seealso
      cref="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)"/>
        </member>

        <member name="M:Graphics.Point.Main">
            <summary>This is the entry point of the Point class testing
            program.
            <para>This program tests each method and operator, and
            is intended to be run after any non-trivial maintenance has
            been performed on the Point class.</para></summary>
        </member>

        <member name="P:Graphics.Point.X">
            <value>Property <c>X</c> represents the point's
            x-coordinate.</value>
        </member>

        <member name="P:Graphics.Point.Y">
            <value>Property <c>Y</c> represents the point's
            y-coordinate.</value>
        </member>
    </members>
</doc>
```

Pattern Matching for C# 7

18.12.2021 • 15 minutes to read

Pattern matching extensions for C# enable many of the benefits of algebraic data types and pattern matching from functional languages, but in a way that smoothly integrates with the feel of the underlying language. The basic features are: record types, which are types whose semantic meaning is described by the shape of the data; and pattern matching, which is a new expression form that enables extremely concise multilevel decomposition of these data types. Elements of this approach are inspired by related features in the programming languages F# and Scala.

Is expression

The `is` operator is extended to test an expression against a *pattern*.

```
relational_expression
    : relational_expression 'is' pattern
    ;
```

This form of *relational_expression* is in addition to the existing forms in the C# specification. It is a compile-time error if the *relational_expression* to the left of the `is` token does not designate a value or does not have a type.

Every *identifier* of the pattern introduces a new local variable that is *definitely assigned* after the `is` operator is `true` (i.e. *definitely assigned when true*).

> Note: There is technically an ambiguity between *type* in an `is-expression` and *constant_pattern*, either of which might be a valid parse of a qualified identifier. We try to bind it as a type for compatibility with previous versions of the language; only if that fails do we resolve it as we do in other contexts, to the first thing found (which must be either a constant or a type). This ambiguity is only present on the right-hand-side of an `is` expression.

Patterns

Patterns are used in the `is` operator and in a *switch_statement* to express the shape of data against which incoming data is to be compared. Patterns may be recursive so that parts of the data may be matched against sub-patterns.

```
pattern
    : declaration_pattern
    | constant_pattern
    | var_pattern
    ;

declaration_pattern
    : type simple_designation
    ;

constant_pattern
    : shift_expression
    ;

var_pattern
    : 'var' simple_designation
    ;
```

Note: There is technically an ambiguity between *type* in an `is-expression` and *constant_pattern*, either of which might be a valid parse of a qualified identifier. We try to bind it as a type for compatibility with previous versions of the language; only if that fails do we resolve it as we do in other contexts, to the first thing found (which must be either a constant or a type). This ambiguity is only present on the right-hand-side of an `is` expression.

Declaration pattern

The *declaration_pattern* both tests that an expression is of a given type and casts it to that type if the test succeeds. If the *simple_designation* is an identifier, it introduces a local variable of the given type named by the given identifier. That local variable is *definitely assigned* when the result of the pattern-matching operation is true.

```
declaration_pattern
    : type simple_designation
    ;
```

The runtime semantic of this expression is that it tests the runtime type of the left-hand *relational_expression* operand against the *type* in the pattern. If it is of that runtime type (or some subtype), the result of the `is operator` is `true`. It declares a new local variable named by the *identifier* that is assigned the value of the left-hand operand when the result is `true`.

Certain combinations of static type of the left-hand-side and the given type are considered incompatible and result in compile-time error. A value of static type `E` is said to be *pattern compatible* with the type `T` if there exists an identity conversion, an implicit reference conversion, a boxing conversion, an explicit reference conversion, or an unboxing conversion from `E` to `T`. It is a compile-time error if an expression of type `E` is not pattern compatible with the type in a type pattern that it is matched with.

Note: In C# 7.1 we extend this to permit a pattern-matching operation if either the input type or the type `T` is an open type. This paragraph is replaced by the following:

Certain combinations of static type of the left-hand-side and the given type are considered incompatible and result in compile-time error. A value of static type `E` is said to be *pattern compatible* with the type `T` if there exists an identity conversion, an implicit reference conversion, a boxing conversion, an explicit reference conversion, or an unboxing conversion from `E` to `T`, **or if either `E` or `T` is an open type**. It is a compile-time error if an expression of type `E` is not pattern compatible with the type in a type pattern that it is matched with.

The declaration pattern is useful for performing run-time type tests of reference types, and replaces the idiom

```
var v = expr as Type;
if (v != null) { // code using v }
```

With the slightly more concise

```
if (expr is Type v) { // code using v }
```

It is an error if *type* is a nullable value type.

The declaration pattern can be used to test values of nullable types: a value of type `Nullable<T>` (or a boxed `T`) matches a type pattern `T2 id` if the value is non-null and the type of `T2` is `T`, or some base type or interface of `T`. For example, in the code fragment

```
int? x = 3;
if (x is int v) { // code using v }
```

The condition of the `if` statement is `true` at runtime and the variable `v` holds the value `3` of type `int` inside the block.

Constant pattern

```
constant_pattern
    : shift_expression
    ;
```

A constant pattern tests the value of an expression against a constant value. The constant may be any constant expression, such as a literal, the name of a declared `const` variable, or an enumeration constant, or a `typeof` expression.

If both *e* and *c* are of integral types, the pattern is considered matched if the result of the expression `e == c` is `true`.

Otherwise the pattern is considered matching if `object.Equals(e, c)` returns `true`. In this case it is a compile-time error if the static type of *e* is not *pattern compatible* with the type of the constant.

Var pattern

```
var_pattern
    : 'var' simple_designation
    ;
```

An expression *e* matches a *var_pattern* always. In other words, a match to a *var pattern* always succeeds. If the *simple_designation* is an identifier, then at runtime the value of *e* is bound to a newly introduced local variable. The type of the local variable is the static type of *e*.

It is an error if the name `var` binds to a type.

Switch statement

The `switch` statement is extended to select for execution the first block having an associated pattern that matches the *switch expression*.

```
switch_label
    : 'case' complex_pattern case_guard? ':'
    | 'case' constant_expression case_guard? ':'
    | 'default' ':'
    ;

case_guard
    : 'when' expression
    ;
```

The order in which patterns are matched is not defined. A compiler is permitted to match patterns out of order, and to reuse the results of already matched patterns to compute the result of matching of other patterns.

If a *case-guard* is present, its expression is of type `bool`. It is evaluated as an additional condition that must be satisfied for the case to be considered satisfied.

It is an error if a *switch_label* can have no effect at runtime because its pattern is subsumed by previous cases. [TODO: We should be more precise about the techniques the compiler is required to use to reach this judgment.]

A pattern variable declared in a *switch_label* is definitely assigned in its case block if and only if that case block contains precisely one *switch_label*.

[TODO: We should specify when a *switch block* is reachable.]

Scope of pattern variables

The scope of a variable declared in a pattern is as follows:

- If the pattern is a case label, then the scope of the variable is the *case block*.

Otherwise the variable is declared in an *is_pattern* expression, and its scope is based on the construct immediately enclosing the expression containing the *is_pattern* expression as follows:

- If the expression is in an expression-bodied lambda, its scope is the body of the lambda.
- If the expression is in an expression-bodied method or property, its scope is the body of the method or property.
- If the expression is in a `when` clause of a `catch` clause, its scope is that `catch` clause.
- If the expression is in an *iteration_statement*, its scope is just that statement.
- Otherwise if the expression is in some other statement form, its scope is the scope containing the statement.

For the purpose of determining the scope, an *embedded_statement* is considered to be in its own scope. For example, the grammar for an *if_statement* is

```
if_statement
    : 'if' '(' boolean_expression ')' embedded_statement
    | 'if' '(' boolean_expression ')' embedded_statement 'else' embedded_statement
    ;
```

So if the controlled statement of an *if_statement* declares a pattern variable, its scope is restricted to that *embedded_statement*:

```
if (x) M(y is var z);
```

In this case the scope of `z` is the embedded statement `M(y is var z);`.

Other cases are errors for other reasons (e.g. in a parameter's default value or an attribute, both of which are an error because those contexts require a constant expression).

In C# 7.3 we added the following contexts in which a pattern variable may be declared:

- If the expression is in a *constructor initializer*, its scope is the *constructor initializer* and the constructor's body.
- If the expression is in a field initializer, its scope is the *equals_value_clause* in which it appears.
- If the expression is in a query clause that is specified to be translated into the body of a lambda, its scope is just that expression.

Changes to syntactic disambiguation

There are situations involving generics where the C# grammar is ambiguous, and the language spec says how to resolve those ambiguities:

7.6.5.2 Grammar ambiguities

The productions for *simple-name* (§7.6.3) and *member-access* (§7.6.5) can give rise to ambiguities in the grammar for expressions. For example, the statement:

```
F(G<A,B>(7));
```

could be interpreted as a call to `F` with two arguments, `G < A` and `B > (7)`. Alternatively, it could be interpreted as a call to `F` with one argument, which is a call to a generic method `G` with two type arguments and one regular argument.

If a sequence of tokens can be parsed (in context) as a *simple-name* (§7.6.3), *member-access* (§7.6.5), or *pointer-member-access* (§18.5.2) ending with a *type-argument-list* (§4.4.1), the token immediately following the closing `>` token is examined. If it is one of

```
( ) ] } : ; , . ? == != | ^
```

then the *type-argument-list* is retained as part of the *simple-name*, *member-access* or *pointer-member-access* and any other possible parse of the sequence of tokens is discarded. Otherwise, the *type-argument-list* is not considered to be part of the *simple-name*, *member-access* or > *pointer-member-access*, even if there is no other possible parse of the sequence of tokens. Note that these rules are not applied when parsing a *type-argument-list* in a *namespace-or-type-name* (§3.8). The statement

```
F(G<A,B>(7));
```

will, according to this rule, be interpreted as a call to `F` with one argument, which is a call to a generic method `G` with two type arguments and one regular argument. The statements

```
F(G < A, B > 7);
F(G < A, B >> 7);
```

will each be interpreted as a call to `F` with two arguments. The statement

```
x = F < A > +y;
```

will be interpreted as a less than operator, greater than operator, and unary plus operator, as if the statement had been written `x = (F < A) > (+y)`, instead of as a *simple-name* with a *type-argument-list* followed by a binary plus operator. In the statement

```
x = y is C<T> + z;
```

the tokens `C<T>` are interpreted as a *namespace-or-type-name* with a *type-argument-list*.

There are a number of changes being introduced in C# 7 that make these disambiguation rules no longer sufficient to handle the complexity of the language.

Out variable declarations

It is now possible to declare a variable in an out argument:

```
M(out Type name);
```

However, the type may be generic:

```
M(out A<B> name);
```

Since the language grammar for the argument uses *expression*, this context is subject to the disambiguation rule. In this case the closing `>` is followed by an *identifier*, which is not one of the tokens that permits it to be treated as a *type-argument-list*. I therefore propose to **add *identifier* to the set of tokens that triggers the disambiguation to a *type-argument-list***.

Tuples and deconstruction declarations

A tuple literal runs into exactly the same issue. Consider the tuple expression

```
(A < B, C > D, E < F, G > H)
```

Under the old C# 6 rules for parsing an argument list, this would parse as a tuple with four elements, starting with `A < B` as the first. However, when this appears on the left of a deconstruction, we want the disambiguation triggered by the *identifier* token as described above:

```
(A<B,C> D, E<F,G> H) = e;
```

This is a deconstruction declaration which declares two variables, the first of which is of type `A<B,C>` and named `D`. In other words, the tuple literal contains two expressions, each of which is a declaration expression.

For simplicity of the specification and compiler, I propose that this tuple literal be parsed as a two-element tuple wherever it appears (whether or not it appears on the left-hand-side of an assignment). That would be a natural result of the disambiguation described in the previous section.

Pattern-matching

Pattern matching introduces a new context where the expression-type ambiguity arises. Previously the right-hand-side of an `is` operator was a type. Now it can be a type or expression, and if it is a type it may be followed by an identifier. This can, technically, change the meaning of existing code:

```
var x = e is T < A > B;
```

This could be parsed under C#6 rules as

```
var x = ((e is T) < A) > B;
```

but under under C#7 rules (with the disambiguation proposed above) would be parsed as

```
var x = e is T<A> B;
```

which declares a variable `B` of type `T<A>`. Fortunately, the native and Roslyn compilers have a bug whereby they give a syntax error on the C#6 code. Therefore this particular breaking change is not a concern.

Pattern-matching introduces additional tokens that should drive the ambiguity resolution toward selecting a type. The following examples of existing valid C#6 code would be broken without additional disambiguation rules:

```
var x = e is A<B> && f;           // &&
var x = e is A<B> || f;           // ||
var x = e is A<B> & f;            // &
var x = e is A<B>[];              // [
```

Proposed change to the disambiguation rule

I propose to revise the specification to change the list of disambiguating tokens from

```
( ) ] } : ; , . ? == != | ^
```

to

```
( ) ] } : ; , . ? == != | ^ && || & [
```

And, in certain contexts, we treat *identifier* as a disambiguating token. Those contexts are where the sequence of tokens being disambiguated is immediately preceded by one of the keywords `is`, `case`, or `out`, or arises while parsing the first element of a tuple literal (in which case the tokens are preceded by `(` or `:` and the identifier is followed by a `,`) or a subsequent element of a tuple literal.

Modified disambiguation rule

The revised disambiguation rule would be something like this

> If a sequence of tokens can be parsed (in context) as a *simple-name* (§7.6.3), *member-access* (§7.6.5), or *pointer-member-access* (§18.5.2) ending with a *type-argument-list* (§4.4.1), the token immediately following the closing `>` token is examined, to see if it is
>
> - One of `( ) ] } : ; , . ? == != | ^ && || & [`; or
> - One of the relational operators `< > <= >= is as`; or
> - A contextual query keyword appearing inside a query expression; or
> - In certain contexts, we treat *identifier* as a disambiguating token. Those contexts are where the sequence of tokens being disambiguated is immediately preceded by one of the keywords `is`, `case` or `out`, or arises while parsing the first element of a tuple literal (in which case the tokens are preceded by `(` or `:` and the identifier is followed by a `,`) or a subsequent element of a tuple literal.
>
> If the following token is among this list, or an identifier in such a context, then the *type-argument-list* is retained as part of the *simple-name*, *member-access* or *pointer-member-access* and any other possible parse of the sequence of tokens is discarded. Otherwise, the *type-argument-list* is not considered to be part of the *simple-name*, *member-access* or *pointer-member-access*, even if there is no other possible parse of the sequence of tokens. Note that these rules are not applied when parsing a *type-argument-list* in a

> *namespace-or-type-name* (§3.8).

Breaking changes due to this proposal

No breaking changes are known due to this proposed disambiguation rule.

Interesting examples

Here are some interesting results of these disambiguation rules:

The expression `(A < B, C > D)` is a tuple with two elements, each a comparison.

The expression `(A<B,C> D, E)` is a tuple with two elements, the first of which is a declaration expression.

The invocation `M(A < B, C > D, E)` has three arguments.

The invocation `M(out A<B,C> D, E)` has two arguments, the first of which is an `out` declaration.

The expression `e is A<B> C` uses a declaration expression.

The case label `case A<B> C:` uses a declaration expression.

Some examples of pattern matching

Is-As

We can replace the idiom

```
var v = expr as Type;
if (v != null) {
    // code using v
}
```

With the slightly more concise and direct

```
if (expr is Type v) {
    // code using v
}
```

Testing nullable

We can replace the idiom

```
Type? v = x?.y?.z;
if (v.HasValue) {
    var value = v.GetValueOrDefault();
    // code using value
}
```

With the slightly more concise and direct

```
if (x?.y?.z is Type value) {
    // code using value
}
```

Arithmetic simplification

Suppose we define a set of recursive types to represent expressions (per a separate proposal):

```
abstract class Expr;
class X() : Expr;
class Const(double Value) : Expr;
class Add(Expr Left, Expr Right) : Expr;
class Mult(Expr Left, Expr Right) : Expr;
class Neg(Expr Value) : Expr;
```

Now we can define a function to compute the (unreduced) derivative of an expression:

```
Expr Deriv(Expr e)
{
  switch (e) {
    case X(): return Const(1);
    case Const(*): return Const(0);
    case Add(var Left, var Right):
      return Add(Deriv(Left), Deriv(Right));
    case Mult(var Left, var Right):
      return Add(Mult(Deriv(Left), Right), Mult(Left, Deriv(Right)));
    case Neg(var Value):
      return Neg(Deriv(Value));
  }
}
```

An expression simplifier demonstrates positional patterns:

```
Expr Simplify(Expr e)
{
  switch (e) {
    case Mult(Const(0), *): return Const(0);
    case Mult(*, Const(0)): return Const(0);
    case Mult(Const(1), var x): return Simplify(x);
    case Mult(var x, Const(1)): return Simplify(x);
    case Mult(Const(var l), Const(var r)): return Const(l*r);
    case Add(Const(0), var x): return Simplify(x);
    case Add(var x, Const(0)): return Simplify(x);
    case Add(Const(var l), Const(var r)): return Const(l+r);
    case Neg(Const(var k)): return Const(-k);
    default: return e;
  }
}
```

Local functions

18.12.2021 • 2 minutes to read

We extend C# to support the declaration of functions in block scope. Local functions may use (capture) variables from the enclosing scope.

The compiler uses flow analysis to detect which variables a local function uses before assigning it a value. Every call of the function requires such variables to be definitely assigned. Similarly the compiler determines which variables are definitely assigned on return. Such variables are considered definitely assigned after the local function is invoked.

Local functions may be called from a lexical point before its definition. Local function declaration statements do not cause a warning when they are not reachable.

TODO: *WRITE SPEC*

Syntax grammar

This grammar is represented as a diff from the current spec grammar.

```
declaration-statement
    : local-variable-declaration ';'
    | local-constant-declaration ';'
+   | local-function-declaration
    ;

+local-function-declaration
+   : local-function-header local-function-body
+   ;

+local-function-header
+   : local-function-modifiers? return-type identifier type-parameter-list?
+       ( formal-parameter-list? ) type-parameter-constraints-clauses
+   ;

+local-function-modifiers
+   : (async | unsafe)
+   ;

+local-function-body
+   : block
+   | arrow-expression-body
+   ;
```

Local functions may use variables defined in the enclosing scope. The current implementation requires that every variable read inside a local function be definitely assigned, as if executing the local function at its point of definition. Also, the local function definition must have been "executed" at any use point.

After experimenting with that a bit (for example, it is not possible to define two mutually recursive local functions), we've since revised how we want the definite assignment to work. The revision (not yet implemented) is that all local variables read in a local function must be definitely assigned at each invocation of the local function. That's actually more subtle than it sounds, and there is a bunch of work remaining to make it work. Once it is done you'll be able to move your local functions to the end of its enclosing block.

The new definite assignment rules are incompatible with inferring the return type of a local function, so we'll likely be removing support for inferring the return type.

Unless you convert a local function to a delegate, capturing is done into frames that are value types. That means you don't get any GC pressure from using local functions with capturing.

Reachability

We add to the spec

> The body of a statement-bodied lambda expression or local function is considered reachable.

Out variable declarations

18.12.2021 • 2 minutes to read

The *out variable declaration* feature enables a variable to be declared at the location that it is being passed as an `out` argument.

```
argument_value
    : 'out' type identifier
    | ...
    ;
```

A variable declared this way is called an *out variable*. You may use the contextual keyword `var` for the variable's type. The scope will be the same as for a *pattern-variable* introduced via pattern-matching.

According to Language Specification (section 7.6.7 Element access) the argument-list of an element-access (indexing expression) does not contain ref or out arguments. However, they are permitted by the compiler for various scenarios, for example indexers declared in metadata that accept `out`.

Within the scope of a local variable introduced by an argument_value, it is a compile-time error to refer to that local variable in a textual position that precedes its declaration.

It is also an error to reference an implicitly-typed (§8.5.1) out variable in the same argument list that immediately contains its declaration.

Overload resolution is modified as follows:

We add a new conversion:

> There is a *conversion from expression* from an implicitly-typed out variable declaration to every type.

Also

> The type of an explicitly-typed out variable argument is the declared type.

and

> An implicitly-typed out variable argument has no type.

The *conversion from expression* from an implicitly-typed out variable declaration is not considered better than any other *conversion from expression*.

The type of an implicitly-typed out variable is the type of the corresponding parameter in the signature of the method selected by overload resolution.

The new syntax node `DeclarationExpressionSyntax` is added to represent the declaration in an out var argument.

Throw expression

18.12.2021 • 2 minutes to read

We extend the set of expression forms to include

```
throw_expression
    : 'throw' null_coalescing_expression
    ;

null_coalescing_expression
    : throw_expression
    ;
```

The type rules are as follows:

- A *throw_expression* has no type.
- A *throw_expression* is convertible to every type by an implicit conversion.

A *throw expression* throws the value produced by evaluating the *null_coalescing_expression*, which must denote a value of the class type `System.Exception`, of a class type that derives from `System.Exception` or of a type parameter type that has `System.Exception` (or a subclass thereof) as its effective base class. If evaluation of the expression produces `null`, a `System.NullReferenceException` is thrown instead.

The behavior at runtime of the evaluation of a *throw expression* is the same as specified for a *throw statement*.

The flow-analysis rules are as follows:

- For every variable *v*, *v* is definitely assigned before the *null_coalescing_expression* of a *throw_expression* iff it is definitely assigned before the *throw_expression*.
- For every variable *v*, *v* is definitely assigned after *throw_expression*.

A *throw expression* is permitted in only the following syntactic contexts:

- As the second or third operand of a ternary conditional operator `?:`
- As the second operand of a null coalescing operator `??`
- As the body of an expression-bodied lambda or method.

Binary literals

18.12.2021 • 2 minutes to read

There's a relatively common request to add binary literals to C# and VB. For bitmasks (e.g. flag enums) this seems genuinely useful, but it would also be great just for educational purposes.

Binary literals would look like this:

```
int nineteen = 0b10011;
```

Syntactically and semantically they are identical to hexadecimal literals, except for using `b` / `B` instead of `x` / `X`, having only digits `0` and `1` and being interpreted in base 2 instead of 16.

There's little cost to implementing these, and little conceptual overhead to users of the language.

Syntax

The grammar would be as follows:

```
integer-literal:
    : ...
    | binary-integer-literal
    ;
binary-integer-literal:
    : `0b` binary-digits integer-type-suffix-opt
    | `0B` binary-digits integer-type-suffix-opt
    ;
binary-digits:
    : binary-digit
    | binary-digits binary-digit
    ;
binary-digit:
    : `0`
    | `1`
    ;
```

Digit separators

18.12.2021 • 2 minutes to read

Being able to group digits in large numeric literals would have great readability impact and no significant downside.

Adding binary literals (#215) would increase the likelihood of numeric literals being long, so the two features enhance each other.

We would follow Java and others, and use an underscore `_` as a digit separator. It would be able to occur everywhere in a numeric literal (except as the first and last character), since different groupings may make sense in different scenarios and especially for different numeric bases:

```
int bin = 0b1001_1010_0001_0100;
int hex = 0x1b_a0_44_fe;
int dec = 33_554_432;
int weird = 1_2__3___4____5_____6______7_______8________9;
double real = 1_000.111_1e-1_000;
```

Any sequence of digits may be separated by underscores, possibly more than one underscore between two consecutive digits. They are allowed in decimals as well as exponents, but following the previous rule, they may not appear next to the decimal (`10_.0`), next to the exponent character (`1.1e_1`), or next to the type specifier (`10_f`). When used in binary and hexadecimal literals, they may not appear immediately following the `0x` or `0b` .

The syntax is straightforward, and the separators have no semantic impact - they are simply ignored.

This has broad value and is easy to implement.

Async Task Types in C#

18.12.2021 • 3 minutes to read

Extend `async` to support *task types* that match a specific pattern, in addition to the well known types `System.Threading.Tasks.Task` and `System.Threading.Tasks.Task<T>`.

Task Type

A *task type* is a `class` or `struct` with an associated *builder type* identified with `System.Runtime.CompilerServices.AsyncMethodBuilderAttribute`. The *task type* may be non-generic, for async methods that do not return a value, or generic, for methods that return a value.

To support `await`, the *task type* must have a corresponding, accessible `GetAwaiter()` method that returns an instance of an *awaiter type* (see *C# 7.7.7.1 Awaitable expressions*).

```
[AsyncMethodBuilder(typeof(MyTaskMethodBuilder<>))]
class MyTask<T>
{
    public Awaiter<T> GetAwaiter();
}

class Awaiter<T> : INotifyCompletion
{
    public bool IsCompleted { get; }
    public T GetResult();
    public void OnCompleted(Action completion);
}
```

Builder Type

The *builder type* is a `class` or `struct` that corresponds to the specific *task type*. The *builder type* can have at most 1 type parameter and must not be nested in a generic type. The *builder type* has the following `public` methods. For non-generic *builder types*, `SetResult()` has no parameters.

```
class MyTaskMethodBuilder<T>
{
    public static MyTaskMethodBuilder<T> Create();

    public void Start<TStateMachine>(ref TStateMachine stateMachine)
        where TStateMachine : IAsyncStateMachine;

    public void SetStateMachine(IAsyncStateMachine stateMachine);
    public void SetException(Exception exception);
    public void SetResult(T result);

    public void AwaitOnCompleted<TAwaiter, TStateMachine>(
        ref TAwaiter awaiter, ref TStateMachine stateMachine)
        where TAwaiter : INotifyCompletion
        where TStateMachine : IAsyncStateMachine;
    public void AwaitUnsafeOnCompleted<TAwaiter, TStateMachine>(
        ref TAwaiter awaiter, ref TStateMachine stateMachine)
        where TAwaiter : ICriticalNotifyCompletion
        where TStateMachine : IAsyncStateMachine;

    public MyTask<T> Task { get; }
}
```

Execution

The types above are used by the compiler to generate the code for the state machine of an `async` method. (The generated code is equivalent to the code generated for async methods that return `Task`, `Task<T>`, or `void`. The difference is, for those well known types, the *builder types* are also known to the compiler.)

`Builder.Create()` is invoked to create an instance of the *builder type*.

`builder.Start(ref stateMachine)` is invoked to associate the builder with compiler-generated state machine instance. The builder must call `stateMachine.MoveNext()` either in `Start()` or after `Start()` has returned to advance the state machine. After `Start()` returns, the `async` method calls `builder.Task` for the task to return from the async method.

Each call to `stateMachine.MoveNext()` will advance the state machine. If the state machine completes successfully, `builder.SetResult()` is called, with the method return value if any. If an exception is thrown in the state machine, `builder.SetException(exception)` is called.

If the state machine reaches an `await expr` expression, `expr.GetAwaiter()` is invoked. If the awaiter implements `ICriticalNotifyCompletion` and `IsCompleted` is false, the state machine invokes `builder.AwaitUnsafeOnCompleted(ref awaiter, ref stateMachine)`. `AwaitUnsafeOnCompleted()` should call `awaiter.UnsafeOnCompleted(action)` with an `Action` that calls `stateMachine.MoveNext()` when the awaiter completes. Similarly for `INotifyCompletion` and `builder.AwaitOnCompleted()`.

`SetStateMachine(IAsyncStateMachine)` is called by the compiler-generated `IAsyncStateMachine` implementation. That can be used to identify the instance of the builder associated with a state machine instance, particularly for cases where the state machine is implemented as a value type: if the builder calls `stateMachine.SetStateMachine(stateMachine)`, the `stateMachine` will call `builder.SetStateMachine(stateMachine)` on the *builder instance associated with* `stateMachine`.

Overload Resolution

Overload resolution is extended to recognize *task types* in addition to `Task` and `Task<T>`.

An `async` lambda with no return value is an exact match for an overload candidate parameter of non-generic *task type*, and an `async` lambda with return type `T` is an exact match for an overload candidate parameter of

generic *task type*.

Otherwise if an `async` lambda is not an exact match for either of two candidate parameters of *task types*, or an exact match for both, and there is an implicit conversion from one candidate type to the other, the from candidate wins. Otherwise recursively evaluate the types `A` and `B` within `Task1<A>` and `Task2<B>` for better match.

Otherwise if an `async` lambda is not an exact match for either of two candidate parameters of *task types*, but one candidate is a more specialized type than the other, the more specialized candidate wins.

Async Main

18.12.2021 • 2 minutes to read

Summary

Allow `await` to be used in an application's Main / entrypoint method by allowing the entrypoint to return `Task` / `Task<int>` and be marked `async`.

Motivation

It is very common when learning C#, when writing console-based utilities, and when writing small test apps to want to call and `await` `async` methods from Main. Today we add a level of complexity here by forcing such `await`'ing to be done in a separate async method, which causes developers to need to write boilerplate like the following just to get started:

```
public static void Main()
{
    MainAsync().GetAwaiter().GetResult();
}

private static async Task MainAsync()
{
    ... // Main body here
}
```

We can remove the need for this boilerplate and make it easier to get started simply by allowing Main itself to be `async` such that `await`s can be used in it.

Detailed design

The following signatures are currently allowed entrypoints:

```
static void Main()
static void Main(string[])
static int Main()
static int Main(string[])
```

We extend the list of allowed entrypoints to include:

```
static Task Main()
static Task<int> Main()
static Task Main(string[])
static Task<int> Main(string[])
```

To avoid compatibility risks, these new signatures will only be considered as valid entrypoints if no overloads of the previous set are present. The language / compiler will not require that the entrypoint be marked as `async`, though we expect the vast majority of uses will be marked as such.

When one of these is identified as the entrypoint, the compiler will synthesize an actual entrypoint method that calls one of these coded methods:

- `static Task Main()` will result in the compiler emitting the equivalent of

`private static void $GeneratedMain() => Main().GetAwaiter().GetResult();`

- `static Task Main(string[])` will result in the compiler emitting the equivalent of `private static void $GeneratedMain(string[] args) => Main(args).GetAwaiter().GetResult();`
- `static Task<int> Main()` will result in the compiler emitting the equivalent of `private static int $GeneratedMain() => Main().GetAwaiter().GetResult();`
- `static Task<int> Main(string[])` will result in the compiler emitting the equivalent of `private static int $GeneratedMain(string[] args) => Main(args).GetAwaiter().GetResult();`

Example usage:

```
using System;
using System.Net.Http;

class Test
{
    static async Task Main(string[] args) =>
     Console.WriteLine(await new HttpClient().GetStringAsync(args[0]));
}
```

Drawbacks

The main drawback is simply the additional complexity of supporting additional entrypoint signatures.

Alternatives

Other variants considered:

Allowing `async void`. We need to keep the semantics the same for code calling it directly, which would then make it difficult for a generated entrypoint to call it (no Task returned). We could solve this by generating two other methods, e.g.

```
public static async void Main()
{
   ... // await code
}
```

becomes

```
public static async void Main() => await $MainTask();

private static void $EntrypointMain() => Main().GetAwaiter().GetResult();

private static async Task $MainTask()
{
    ... // await code
}
```

There are also concerns around encouraging usage of `async void`.

Using "MainAsync" instead of "Main" as the name. While the async suffix is recommended for Task-returning methods, that's primarily about library functionality, which Main is not, and supporting additional entrypoint names beyond "Main" is not worth it.

Unresolved questions

n/a

Design meetings

n/a

Target-typed "default" literal

18.12.2021 • 2 minutes to read

Summary

The target-typed `default` feature is a shorter form variation of the `default(T)` operator, which allows the type to be omitted. Its type is inferred by target-typing instead. Aside from that, it behaves like `default(T)`.

Motivation

The main motivation is to avoid typing redundant information.

For instance, when invoking `void Method(ImmutableArray<SomeType> array)`, the *default* literal allows `M(default)` in place of `M(default(ImmutableArray<SomeType>))`.

This is applicable in a number of scenarios, such as:

- declaring locals (`ImmutableArray<SomeType> x = default;`)
- ternary operations (`var x = flag ? default : ImmutableArray<SomeType>.Empty;`)
- returning in methods and lambdas (`return default;`)
- declaring default values for optional parameters (`void Method(ImmutableArray<SomeType> arrayOpt = default)`)
- including default values in array creation expressions (`var x = new[] { default, ImmutableArray.Create(y) };`)

Detailed design

A new expression is introduced, the *default* literal. An expression with this classification can be implicitly converted to any type, by a *default literal conversion*.

The inference of the type for the *default* literal works the same as that for the *null* literal, except that any type is allowed (not just reference types).

This conversion produces the default value of the inferred type.

The *default* literal may have a constant value, depending on the inferred type. So `const int x = default;` is legal, but `const int? y = default;` is not.

The *default* literal can be the operand of equality operators, as long as the other operand has a type. So `default == x` and `x == default` are valid expressions, but `default == default` is illegal.

Drawbacks

A minor drawback is that *default* literal can be used in place of *null* literal in most contexts. Two of the exceptions are `throw null;` and `null == null`, which are allowed for the *null* literal, but not the *default* literal.

Alternatives

There are a couple of alternatives to consider:

- The status quo: The feature is not justified on its own merits and developers continue to use the default operator with an explicit type.
- Extending the null literal: This is the VB approach with `Nothing`. We could allow `int x = null;`.

Unresolved questions

- [x] Should *default* be allowed as the operand of the *is* or *as* operators? Answer: disallow `default is T`, allow `x is default`, allow `default as RefType` (with always-null warning)

Design meetings

- LDM 3/7/2017
- LDM 3/28/2017
- LDM 5/31/2017

Infer tuple names (aka. tuple projection initializers)

18.12.2021 • 2 minutes to read

Summary

In a number of common cases, this feature allows the tuple element names to be omitted and instead be inferred. For instance, instead of typing `(f1: x.f1, f2: x?.f2)`, the element names "f1" and "f2" can be inferred from `(x.f1, x?.f2)`.

This parallels the behavior of anonymous types, which allow inferring member names during creation. For instance, `new { x.f1, y?.f2 }` declares members "f1" and "f2".

This is particularly handy when using tuples in LINQ:

```
// "c" and "result" have element names "f1" and "f2"
var result = list.Select(c => (c.f1, c.f2)).Where(t => t.f2 == 1);
```

Detailed design

There are two parts to the change:

1. Try to infer a candidate name for each tuple element which does not have an explicit name:
   - Using same rules as name inference for anonymous types.
     - In C#, this allows three cases: `y` (identifier), `x.y` (simple member access) and `x?.y` (conditional access).
     - In VB, this allows for additional cases, such as `x.y()`.
   - Rejecting reserved tuple names (case-sensitive in C#, case-insensitive in VB), as they are either forbidden or already implicit. For instance, such as `ItemN`, `Rest`, and `ToString`.
   - If any candidate names are duplicates (case-sensitive in C#, case-insensitive in VB) within the entire tuple, we drop those candidates,
2. During conversions (which check and warn about dropping names from tuple literals), inferred names would not produce any warnings. This avoids breaking existing tuple code.

Note that the rule for handling duplicates is different than that for anonymous types. For instance, `new { x.f1, x.f1 }` produces an error, but `(x.f1, x.f1)` would still be allowed (just without any inferred names). This avoids breaking existing tuple code.

For consistency, the same would apply to tuples produced by deconstruction-assignments (in C#):

```
// tuple has element names "f1" and "f2"
var tuple = ((x.f1, x?.f2) = (1, 2));
```

The same would also apply to VB tuples, using the VB-specific rules for inferring name from expression and case-insensitive name comparisons.

When using the C# 7.1 compiler (or later) with language version "7.0", the element names will be inferred (despite the feature not being available), but there will be a use-site error for trying to access them. This will limit additions of new code that would later face the compatibility issue (described below).

Drawbacks

The main drawback is that this introduces a compatibility break from C# 7.0:

```
Action y = () => M();
var t = (x: x, y);
t.y(); // this might have previously picked up an extension method called “y”, but would now call the
lambda.
```

The compatibility council found this break acceptable, given that it is limited and the time window since tuples shipped (in C# 7.0) is short.

References

- LDM April 4th 2017
- Github discussion (thanks @alrz for bringing this issue up)
- Tuples design

pattern-matching with generics

18.12.2021 • 2 minutes to read

Summary

The specification for the existing C# as operator permits there to be no conversion between the type of the operand and the specified type when either is an open type. However, in C# 7 the `Type identifier` pattern requires there be a conversion between the type of the input and the given type.

We propose to relax this and change `expression is Type identifier`, in addition to being permitted in the conditions when it is permitted in C# 7, to also be permitted when `expression as Type` would be allowed. Specifically, the new cases are cases where the type of the expression or the specified type is an open type.

Motivation

Cases where pattern-matching should "obviously" be permitted currently fail to compile. See, for example, https://github.com/dotnet/roslyn/issues/16195.

Detailed design

We change the paragraph in the pattern-matching specification (the proposed addition is shown in bold):

> Certain combinations of static type of the left-hand-side and the given type are considered incompatible and result in compile-time error. A value of static type `E` is said to be *pattern compatible* with the type `T` if there exists an identity conversion, an implicit reference conversion, a boxing conversion, an explicit reference conversion, or an unboxing conversion from `E` to `T`, **or if either `E` or `T` is an open type**. It is a compile-time error if an expression of type `E` is not pattern compatible with the type in a type pattern that it is matched with.

Drawbacks

None.

Alternatives

None.

Unresolved questions

None.

Design meetings

LDM considered this question and felt it was a bug-fix level change. We are treating it as a separate language feature because just making the change after the language has been released would introduce a forward incompatibility. Using the proposed change requires that the programmer specify language version 7.1.

Readonly references

18.12.2021 • 24 minutes to read

Summary

The "readonly references" feature is actually a group of features that leverage the efficiency of passing variables by reference, but without exposing the data to modifications:

- `in` parameters
- `ref readonly` returns
- `readonly` structs
- `ref` / `in` extension methods
- `ref readonly` locals
- `ref` conditional expressions

Passing arguments as readonly references.

There is an existing proposal that touches this topic https://github.com/dotnet/roslyn/issues/115 as a special case of readonly parameters without going into many details. Here I just want to acknowledge that the idea by itself is not very new.

Motivation

Prior to this feature C# did not have an efficient way of expressing a desire to pass struct variables into method calls for readonly purposes with no intention of modifying. Regular by-value argument passing implies copying, which adds unnecessary costs. That drives users to use by-ref argument passing and rely on comments/documentation to indicate that the data is not supposed to be mutated by the callee. It is not a good solution for many reasons.
The examples are numerous - vector/matrix math operators in graphics libraries like XNA are known to have ref operands purely because of performance considerations. There is code in Roslyn compiler itself that uses structs to avoid allocations and then passes them by reference to avoid copying costs.

Solution (`in` parameters)

Similarly to the `out` parameters, `in` parameters are passed as managed references with additional guarantees from the callee.
Unlike `out` parameters which *must* be assigned by the callee before any other use, `in` parameters cannot be assigned by the callee at all.

As a result `in` parameters allow for effectiveness of indirect argument passing without exposing arguments to mutations by the callee.

Declaring `in` parameters

`in` parameters are declared by using `in` keyword as a modifier in the parameter signature.

For all purposes the `in` parameter is treated as a `readonly` variable. Most of the restrictions on the use of `in` parameters inside the method are the same as with `readonly` fields.

> Indeed an `in` parameter may represent a `readonly` field. Similarity of restrictions is not a coincidence.

For example fields of an `in` parameter which has a struct type are all recursively classified as `readonly`

variables .

```
static Vector3 Add (in Vector3 v1, in Vector3 v2)
{
    // not OK!!
    v1 = default(Vector3);

    // not OK!!
    v1.X = 0;

    // not OK!!
    foo(ref v1.X);

    // OK
    return new Vector3(v1.X + v2.X, v1.Y + v2.Y, v1.Z + v2.Z);
}
```

- `in` parameters are allowed anywhere where ordinary byval parameters are allowed. This includes indexers, operators (including conversions), delegates, lambdas, local functions.

```
(in int x) => x                                                 // lambda expression
TValue this[in TKey index];                                     // indexer
public static Vector3 operator +(in Vector3 x, in Vector3 y) => ... // operator
```

- `in` is not allowed in combination with `out` or with anything that `out` does not combine with.
- It is not permitted to overload on `ref` / `out` / `in` differences.
- It is permitted to overload on ordinary byval and `in` differences.
- For the purpose of OHI (Overloading, Hiding, Implementing), `in` behaves similarly to an `out` parameter. All the same rules apply. For example the overriding method will have to match `in` parameters with `in` parameters of an identity-convertible type.
- For the purpose of delegate/lambda/method group conversions, `in` behaves similarly to an `out` parameter. Lambdas and applicable method group conversion candidates will have to match `in` parameters of the target delegate with `in` parameters of an identity-convertible type.
- For the purpose of generic variance, `in` parameters are nonvariant.

> NOTE: There are no warnings on `in` parameters that have reference or primitives types. It may be pointless in general, but in some cases user must/want to pass primitives as `in`. Examples - overriding a generic method like `Method(in T param)` when `T` was substituted to be `int`, or when having methods like `Volatile.Read(in int location)`
>
> It is conceivable to have an analyzer that warns in cases of inefficient use of `in` parameters, but the rules for such analysis would be too fuzzy to be a part of a language specification.

Use of `in` at call sites. (`in` arguments)

There are two ways to pass arguments to `in` parameters.

`in` arguments can match `in` parameters:

An argument with an `in` modifier at the call site can match `in` parameters.

```
int x = 1;

void M1<T>(in T x)
{
  // . . .
}

var x = M1(in x);  // in argument to a method

class D
{
    public string this[in Guid index];
}

D dictionary = . . . ;
var y = dictionary[in Guid.Empty]; // in argument to an indexer
```

- `in` argument must be a *readable* LValue(*). Example: `M1(in 42)` is invalid

> (*) The notion of LValue/RValue vary between languages.
> Here, by LValue I mean an expression that represent a location that can be referred to directly. And RValue means an expression that yields a temporary result which does not persist on its own.

- In particular it is valid to pass `readonly` fields, `in` parameters or other formally `readonly` variables as `in` arguments. Example: `dictionary[in Guid.Empty]` is legal. `Guid.Empty` is a static readonly field.
- `in` argument must have type *identity-convertible* to the type of the parameter. Example: `M1<object>(in Guid.Empty)` is invalid. `Guid.Empty` is not *identity-convertible* to `object`

The motivation for the above rules is that `in` arguments guarantee *aliasing* of the argument variable. The callee always receives a direct reference to the same location as represented by the argument.

- in rare situations when `in` arguments must be stack-spilled due to `await` expressions used as operands of the same call, the behavior is the same as with `out` and `ref` arguments - if the variable cannot be spilled in referentially-transparent manner, an error is reported.

Examples:

1. `M1(in staticField, await SomethingAsync())` is valid. `staticField` is a static field which can be accessed more than once without observable side effects. Therefore both the order of side effects and aliasing requirements can be provided.
2. `M1(in RefReturningMethod(), await SomethingAsync())` will produce an error. `RefReturningMethod()` is a `ref` returning method. A method call may have observable side effects, therefore it must be evaluated before the `SomethingAsync()` operand. However the result of the invocation is a reference that cannot be preserved across the `await` suspension point which make the direct reference requirement impossible.

> NOTE: the stack spilling errors are considered to be implementation-specific limitations. Therefore they do not have effect on overload resolution or lambda inference.

Ordinary byval arguments can match `in` parameters:

Regular arguments without modifiers can match `in` parameters. In such case the arguments have the same relaxed constraints as an ordinary byval arguments would have.

The motivation for this scenario is that `in` parameters in APIs may result in inconveniences for the user when arguments cannot be passed as a direct reference - ex: literals, computed or `await`-ed results or arguments that happen to have more specific types.

All these cases have a trivial solution of storing the argument value in a temporary local of appropriate type and passing that local as an `in` argument.
To reduce the need for such boilerplate code compiler can perform the same transformation, if needed, when `in` modifier is not present at the call site.

In addition, in some cases, such as invocation of operators, or `in` extension methods, there is no syntactical way to specify `in` at all. That alone requires specifying the behavior of ordinary byval arguments when they match `in` parameters.

In particular:

- it is valid to pass RValues. A reference to a temporary is passed in such case. Example:

```
Print("hello");      // not an error.

void Print<T>(in T x)
{
  //. . .
}
```

- implicit conversions are allowed.

> This is actually a special case of passing an RValue

A reference to a temporary holding converted value is passed in such case. Example:

```
Print<int>(Short.MaxValue)     // not an error.
```

- in a case of a receiver of an `in` extension method (as opposed to `ref` extension methods), RValues or implicit *this-argument-conversions* are allowed. A reference to a temporary holding converted value is passed in such case. Example:

```
public static IEnumerable<T> Concat<T>(in this (IEnumerable<T>, IEnumerable<T>) arg)  => . . .;

("aa", "bb").Concat<char>()    // not an error.
```

More information on `ref` / `in` extension methods is provided further in this document.

- argument spilling due to `await` operands could spill "by-value", if necessary. In scenarios where providing a direct reference to the argument is not possible due to intervening `await` a copy of the argument's value is spilled instead.
  Example:

```
M1(RefReturningMethod(), await SomethingAsync())   // not an error.
```

Since the result of a side-effecting invocation is a reference that cannot be preserved across `await` suspension, a temporary containing the actual value will be preserved instead (as it would in an ordinary byval parameter case).

Omitted optional arguments

It is permitted for an `in` parameter to specify a default value. That makes the corresponding argument optional.

Omitting optional argument at the call site results in passing the default value via a temporary.

```
Print("hello");      // not an error, same as
Print("hello", c: Color.Black);

void Print(string s, in Color c = Color.Black)
{
    // . . .
}
```

Aliasing behavior in general

Just like `ref` and `out` variables, `in` variables are references/aliases to existing locations.

While callee is not allowed to write into them, reading an `in` parameter can observe different values as a side effect of other evaluations.

Example:

```
static Vector3 v = Vector3.UnitY;

static void Main()
{
    Test(v);
}

static void Test(in Vector3 v1)
{
    Debug.Assert(v1 == Vector3.UnitY);
    // changes v1 deterministically (no races required)
    ChangeV();
    Debug.Assert(v1 == Vector3.UnitX);
}

static void ChangeV()
{
    v = Vector3.UnitX;
}
```

`in` parameters and capturing of local variables.

For the purpose of lambda/async capturing `in` parameters behave the same as `out` and `ref` parameters.

- `in` parameters cannot be captured in a closure
- `in` parameters are not allowed in iterator methods
- `in` parameters are not allowed in async methods

Temporary variables.

Some uses of `in` parameter passing may require indirect use of a temporary local variable:

- `in` arguments are always passed as direct aliases when call-site uses `in`. Temporary is never used in such case.
- `in` arguments are not required to be direct aliases when call-site does not use `in`. When argument is not an LValue, a temporary may be used.
- `in` parameter may have default value. When corresponding argument is omitted at the call site, the default value are passed via a temporary.
- `in` arguments may have implicit conversions, including those that do not preserve identity. A temporary is used in those cases.
- receivers of ordinary struct calls may not be writeable LValues (**existing case!**). A temporary is used in those cases.

The life time of the argument temporaries matches the closest encompassing scope of the call-site.

The formal life time of temporary variables is semantically significant in scenarios involving escape analysis of variables returned by reference.

Metadata representation of `in` parameters.

When `System.Runtime.CompilerServices.IsReadOnlyAttribute` is applied to a byref parameter, it means that the parameter is an `in` parameter.

In addition, if the method is *abstract* or *virtual*, then the signature of such parameters (and only such parameters) must have `modreq[System.Runtime.InteropServices.InAttribute]`.

Motivation: this is done to ensure that in a case of method overriding/implementing the `in` parameters match.

Same requirements apply to `Invoke` methods in delegates.

Motivation: this is to ensure that existing compilers cannot simply ignore `readonly` when creating or assigning delegates.

Returning by readonly reference.

Motivation

The motivation for this sub-feature is roughly symmetrical to the reasons for the `in` parameters - avoiding copying, but on the returning side. Prior to this feature, a method or an indexer had two options: 1) return by reference and be exposed to possible mutations or 2) return by value which results in copying.

Solution (`ref readonly` returns)

The feature allows a member to return variables by reference without exposing them to mutations.

Declaring `ref readonly` returning members

A combination of modifiers `ref readonly` on the return signature is used to to indicate that the member returns a readonly reference.

For all purposes a `ref readonly` member is treated as a `readonly` variable - similar to `readonly` fields and `in` parameters.

For example fields of `ref readonly` member which has a struct type are all recursively classified as `readonly` variables. - It is permitted to pass them as `in` arguments, but not as `ref` or `out` arguments.

```
ref readonly Guid Method1()
{
}

Method2(in Method1()); // valid. Can pass as `in` argument.

Method3(ref Method1()); // not valid. Cannot pass as `ref` argument
```

- `ref readonly` returns are allowed in the same places were `ref` returns are allowed. This includes indexers, delegates, lambdas, local functions.
- It is not permitted to overload on `ref` / `ref readonly` / differences.
- It is permitted to overload on ordinary byval and `ref readonly` return differences.
- For the purpose of OHI (Overloading, Hiding, Implementing), `ref readonly` is similar but distinct from `ref`. For example the a method that overrides `ref readonly` one, must itself be `ref readonly` and have

identity-convertible type.

- For the purpose of delegate/lambda/method group conversions, `ref readonly` is similar but distinct from `ref`. Lambdas and applicable method group conversion candidates have to match `ref readonly` return of the target delegate with `ref readonly` return of the type that is identity-convertible.
- For the purpose of generic variance, `ref readonly` returns are nonvariant.

> NOTE: There are no warnings on `ref readonly` returns that have reference or primitives types. It may be pointless in general, but in some cases user must/want to pass primitives as `in`. Examples - overriding a generic method like `ref readonly T Method()` when `T` was substituted to be `int`.
>
> It is conceivable to have an analyzer that warns in cases of inefficient use of `ref readonly` returns, but the rules for such analysis would be too fuzzy to be a part of a language specification.

Returning from `ref readonly` members

Inside the method body the syntax is the same as with regular ref returns. The `readonly` will be inferred from the containing method.

The motivation is that `return ref readonly <expression>` is unnecessary long and only allows for mismatches on the `readonly` part that would always result in errors. The `ref` is, however, required for consistency with other scenarios where something is passed via strict aliasing vs. by value.

> Unlike the case with `in` parameters, `ref readonly` returns never return via a local copy. Considering that the copy would cease to exist immediately upon returning such practice would be pointless and dangerous. Therefore `ref readonly` returns are always direct references.

Example:

```
struct ImmutableArray<T>
{
    private readonly T[] array;

    public ref readonly T ItemRef(int i)
    {
        // returning a readonly reference to an array element
        return ref this.array[i];
    }
}
```

- An argument of `return ref` must be an LValue (**existing rule**)
- An argument of `return ref` must be "safe to return" (**existing rule**)
- In a `ref readonly` member an argument of `return ref` is *not required to be writeable*. For example such member can ref-return a readonly field or one of its `in` parameters.

Safe to Return rules.

Normal safe to return rules for references will apply to readonly references as well.

Note that a `ref readonly` can be obtained from a regular `ref` local/parameter/return, but not the other way around. Otherwise the safety of `ref readonly` returns is inferred the same way as for regular `ref` returns.

Considering that RValues can be passed as `in` parameter and returned as `ref readonly` we need one more rule - **RValues are not safe-to-return by reference**.

> Consider the situation when an RValue is passed to an `in` parameter via a copy and then returned back in a

> form of a `ref readonly`. In the context of the caller the result of such invocation is a reference to local data and as such is unsafe to return. Once RValues are not safe to return, the existing rule `#6` already handles this case.

Example:

```
ref readonly Vector3 Test1()
{
    // can pass an RValue as "in" (via a temp copy)
    // but the result is not safe to return
    // because the RValue argument was not safe to return by reference
    return ref Test2(default(Vector3));
}

ref readonly Vector3 Test2(in Vector3 r)
{
    // this is ok, r is returnable
    return ref r;
}
```

Updated `safe to return` rules:

1. **refs to variables on the heap are safe to return**
2. **ref/in parameters are safe to return** `in` parameters naturally can only be returned as readonly.
3. **out parameters are safe to return** (but must be definitely assigned, as is already the case today)
4. **instance struct fields are safe to return as long as the receiver is safe to return**
5. **'this' is not safe to return from struct members**
6. **a ref, returned from another method is safe to return if all refs/outs passed to that method as formal parameters were safe to return.** *Specifically it is irrelevant if receiver is safe to return, regardless whether receiver is a struct, class or typed as a generic type parameter.*
7. **RValues are not safe to return by reference.** *Specifically RValues are safe to pass as in parameters.*

> NOTE: There are additional rules regarding safety of returns that come into play when ref-like types and ref-reassignments are involved. The rules equally apply to `ref` and `ref readonly` members and therefore are not mentioned here.

Aliasing behavior.

`ref readonly` members provide the same aliasing behavior as ordinary `ref` members (except for being readonly). Therefore for the purpose of capturing in lambdas, async, iterators, stack spilling etc... the same restrictions apply. - I.E. due to inability to capture the actual references and due to side-effecting nature of member evaluation such scenarios are disallowed.

> It is permitted and required to make a copy when `ref readonly` return is a receiver of regular struct methods, which take `this` as an ordinary writeable reference. Historically in all cases where such invocations are applied to readonly variable a local copy is made.

Metadata representation.

When `System.Runtime.CompilerServices.IsReadOnlyAttribute` is applied to the return of a byref returning method, it means that the method returns a readonly reference.

In addition, the result signature of such methods (and only those methods) must have `modreq[System.Runtime.CompilerServices.IsReadOnlyAttribute]`.

Motivation: this is to ensure that existing compilers cannot simply ignore `readonly` when invoking methods

with `ref readonly` returns

Readonly structs

In short - a feature that makes `this` parameter of all instance members of a struct, except for constructors, an `in` parameter.

Motivation

Compiler must assume that any method call on a struct instance may modify the instance. Indeed a writeable reference is passed to the method as `this` parameter and fully enables this behavior. To allow such invocations on `readonly` variables, the invocations are applied to temp copies. That could be unintuitive and sometimes forces people to abandon `readonly` for performance reasons.
Example: https://codeblog.jonskeet.uk/2014/07/16/micro-optimization-the-surprising-inefficiency-of-readonly-fields/

After adding support for `in` parameters and `ref readonly` returns the problem of defensive copying will get worse since readonly variables will become more common.

Solution

Allow `readonly` modifier on struct declarations which would result in `this` being treated as `in` parameter on all struct instance methods except for constructors.

```
static void Test(in Vector3 v1)
{
    // no need to make a copy of v1 since Vector3 is a readonly struct
    System.Console.WriteLine(v1.ToString());
}

readonly struct Vector3
{
    . . .

    public override string ToString()
    {
        // not OK!!  `this` is an `in` parameter
        foo(ref this.X);

        // OK
        return $"X: {X}, Y: {Y}, Z: {Z}";
    }
}
```

Restrictions on members of readonly struct

- Instance fields of a readonly struct must be readonly.
  Motivation: can only be written to externally, but not through members.
- Instance autoproperties of a readonly struct must be get-only.
  Motivation: consequence of restriction on instance fields.
- Readonly struct may not declare field-like events.
  Motivation: consequence of restriction on instance fields.

Metadata representation.

When `System.Runtime.CompilerServices.IsReadOnlyAttribute` is applied to a value type, it means that the type is a `readonly struct`.

In particular:

- The identity of the `IsReadOnlyAttribute` type is unimportant. In fact it can be embedded by the compiler in

the containing assembly if needed.

`ref` / `in` extension methods

There is actually an existing proposal (https://github.com/dotnet/roslyn/issues/165) and corresponding prototype PR (https://github.com/dotnet/roslyn/pull/15650). I just want to acknowledge that this idea is not entirely new. It is, however, relevant here since `ref readonly` elegantly removes the most contentious issue about such methods - what to do with RValue receivers.

The general idea is allowing extension methods to take the `this` parameter by reference, as long as the type is known to be a struct type.

```
public static void Extension(ref this Guid self)
{
    // do something
}
```

The reasons for writing such extension methods are primarily:

1. Avoid copying when receiver is a large struct
2. Allow mutating extension methods on structs

The reasons why we do not want to allow this on classes

1. It would be of very limited purpose.
2. It would break long standing invariant that a method call cannot turn non-`null` receiver to become `null` after invocation.

> In fact, currently a non-`null` variable cannot become `null` unless *explicitly* assigned or passed by `ref` or `out`. That greatly aids readability or other forms of "can this be a null here" analysis. 3. It would be hard to reconcile with "evaluate once" semantics of null-conditional accesses. Example: `obj.stringField?.RefExtension(...)` - need to capture a copy of `stringField` to make the null check meaningful, but then assignments to `this` inside RefExtension would not be reflected back to the field.

An ability to declare extension methods on **structs** that take the first argument by reference was a long-standing request. One of the blocking consideration was "what happens if receiver is not an LValue?".

- There is a precedent that any extension method could also be called as a static method (sometimes it is the only way to resolve ambiguity). It would dictate that RValue receivers should be disallowed.
- On the other hand there is a practice of making invocation on a copy in similar situations when struct instance methods are involved.

The reason why the "implicit copying" exists is because the majority of struct methods do not actually modify the struct while not being able to indicate that. Therefore the most practical solution was to just make the invocation on a copy, but this practice is known for harming performance and causing bugs.

Now, with availability of `in` parameters, it is possible for an extension to signal the intent. Therefore the conundrum can be resolved by requiring `ref` extensions to be called with writeable receivers while `in` extensions permit implicit copying if necessary.

```
// this can be called on either RValue or an LValue
public static void Reader(in this Guid self)
{
    // do something nonmutating.
    WriteLine(self == default(Guid));
}

// this can be called only on an LValue
public static void Mutator(ref this Guid self)
{
    // can mutate self
    self = new Guid();
}
```

`in` extensions and generics.

The purpose of `ref` extension methods is to mutate the receiver directly or by invoking mutating members. Therefore `ref this T` extensions are allowed as long as `T` is constrained to be a struct.

On the other hand `in` extension methods exist specifically to reduce implicit copying. However any use of an `in T` parameter will have to be done through an interface member. Since all interface members are considered mutating, any such use would require a copy. - Instead of reducing copying, the effect would be the opposite. Therefore `in this T` is not allowed when `T` is a generic type parameter regardless of constraints.

Valid kinds of extension methods (recap):

The following forms of `this` declaration in an extension method are now allowed:

1. `this T arg` - regular byval extension. (**existing case**)

- T can be any type, including reference types or type parameters. Instance will be the same variable after the call. Allows implicit conversions of *this-argument-conversion* kind. Can be called on RValues.
- `in this T self` - `in` extension. T must be an actual struct type. Instance will be the same variable after the call. Allows implicit conversions of *this-argument-conversion* kind. Can be called on RValues (may be invoked on a temp if needed).
- `ref this T self` - `ref` extension. T must be a struct type or a generic type parameter constrained to be a struct. Instance may be written to by the invocation. Allows only identity conversions. Must be called on writeable LValue. (never invoked via a temp).

Readonly ref locals.

Motivation.

Once `ref readonly` members were introduced, it was clear from the use that they need to be paired with appropriate kind of local. Evaluation of a member may produce or observe side effects, therefore if the result must be used more than once, it needs to be stored. Ordinary `ref` locals do not help here since they cannot be assigned a `readonly` reference.

Solution.

Allow declaring `ref readonly` locals. This is a new kind of `ref` locals that is not writeable. As a result `ref readonly` locals can accept references to readonly variables without exposing these variables to writes.

Declaring and using `ref readonly` locals.

The syntax of such locals uses `ref readonly` modifiers at declaration site (in that specific order). Similarly to ordinary `ref` locals, `ref readonly` locals must be ref-initialized at declaration. Unlike regular `ref` locals, `ref readonly` locals can refer to `readonly` LValues like `in` parameters, `readonly` fields, `ref readonly`

methods.

For all purposes a `ref readonly` local is treated as a `readonly` variable. Most of the restrictions on the use are the same as with `readonly` fields or `in` parameters.

For example fields of an `in` parameter which has a struct type are all recursively classified as `readonly` variables .

```
static readonly ref Vector3 M1() => . . .

static readonly ref Vector3 M1_Trace()
{
    // OK
    ref readonly var r1 = ref M1();

    // Not valid. Need an LValue
    ref readonly Vector3 r2 = ref default(Vector3);

    // Not valid. r1 is readonly.
    Mutate(ref r1);

    // OK.
    Print(in r1);

    // OK.
    return ref r1;
}
```

Restrictions on use of `ref readonly` locals

Except for their `readonly` nature, `ref readonly` locals behave like ordinary `ref` locals and are subject to exactly same restrictions.
For example restrictions related to capturing in closures, declaring in `async` methods or the `safe-to-return` analysis equally applies to `ref readonly` locals.

Ternary `ref` expressions. (aka "Conditional LValues")

Motivation

Use of `ref` and `ref readonly` locals exposed a need to ref-initialize such locals with one or another target variable based on a condition.

A typical workaround is to introduce a method like:

```
ref T Choice(bool condition, ref T consequence, ref T alternative)
{
    if (condition)
    {
         return ref consequence;
    }
    else
    {
         return ref alternative;
    }
}
```

Note that `Choice` is not an exact replacement of a ternary since *all* arguments must be evaluated at the call site, which was leading to unintuitive behavior and bugs.

The following will not work as expected:

```
// will crash with NRE because 'arr[0]' will be executed unconditionally
ref var r = ref Choice(arr != null, ref arr[0], ref otherArr[0]);
```

Solution

Allow special kind of conditional expression that evaluates to a reference to one of LValue argument based on a condition.

Using `ref` ternary expression.

The syntax for the `ref` flavor of a conditional expression is
`<condition> ? ref <consequence> : ref <alternative>;`

Just like with the ordinary conditional expression only `<consequence>` or `<alternative>` is evaluated depending on result of the boolean condition expression.

Unlike ordinary conditional expression, `ref` conditional expression:

- requires that `<consequence>` and `<alternative>` are LValues.
- `ref` conditional expression itself is an LValue and
- `ref` conditional expression is writeable if both `<consequence>` and `<alternative>` are writeable LValues

Examples:
`ref` ternary is an LValue and as such it can be passed/assigned/returned by reference;

```
// pass by reference
foo(ref (arr != null ? ref arr[0]: ref otherArr[0]));

// return by reference
return ref (arr != null ? ref arr[0]: ref otherArr[0]);
```

Being an LValue, it can also be assigned to.

```
// assign to
(arr != null ? ref arr[0]: ref otherArr[0]) = 1;

// error. readOnlyField is readonly and thus conditional expression is readonly
(arr != null ? ref arr[0]: ref obj.readOnlyField) = 1;
```

Can be used as a receiver of a method call and skip copying if necessary.

```
// no copies
(arr != null ? ref arr[0]: ref otherArr[0]).StructMethod();

// invoked on a copy.
// The receiver is `readonly` because readOnlyField is readonly.
(arr != null ? ref arr[0]: ref obj.readOnlyField).StructMethod();

// no copies. `ReadonlyStructMethod` is a method on a `readonly` struct
// and can be invoked directly on a readonly receiver
(arr != null ? ref arr[0]: ref obj.readOnlyField).ReadonlyStructMethod();
```

`ref` ternary can be used in a regular (not ref) context as well.

```
// only an example
// a regular ternary could work here just the same
int x = (arr != null ? ref arr[0]: ref otherArr[0]);
```

Drawbacks

I can see two major arguments against enhanced support for references and readonly references:

1. The problems that are solved here are very old. Why suddenly solve them now, especially since it would not help existing code?

As we find C# and .Net used in new domains, some problems become more prominent.
As examples of environments that are more critical than average about computation overheads, I can list

- cloud/datacenter scenarios where computation is billed for and responsiveness is a competitive advantage.
- Games/VR/AR with soft-realtime requirements on latencies

This feature does not sacrifice any of the existing strengths such as type-safety, while allowing to lower overheads in some common scenarios.

2. Can we reasonably guarantee that the callee will play by the rules when it opts into `readonly` contracts?

We have similar trust when using `out`. Incorrect implementation of `out` can cause unspecified behavior, but in reality it rarely happens.

Making the formal verification rules familiar with `ref readonly` would further mitigate the trust issue.

Alternatives

The main competing design is really "do nothing".

Unresolved questions

Design meetings

https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-02-22.md
https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-03-01.md
https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-08-28.md
https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-09-25.md
https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-09-27.md

Compile time enforcement of safety for ref-like types

18.12.2021 • 16 minutes to read

Introduction

The main reason for the additional safety rules when dealing with types like `Span<T>` and `ReadOnlySpan<T>` is that such types must be confined to the execution stack.

There are two reasons why `Span<T>` and similar types must be a stack-only types.

1. `Span<T>` is semantically a struct containing a reference and a range - `(ref T data, int length)`. Regardless of actual implementation, writes to such struct would not be atomic. Concurrent "tearing" of such struct would lead to the possibility of `length` not matching the `data`, causing out-of-range accesses and type-safety violations, which ultimately could result in GC heap corruption in seemingly "safe" code.
2. Some implementations of `Span<T>` literally contain a managed pointer in one of its fields. Managed pointers are not supported as fields of heap objects and code that manages to put a managed pointer on the GC heap typically crashes at JIT time.

All the above problems would be alleviated if instances of `Span<T>` are constrained to exist only on the execution stack.

An additional problem arises due to composition. It would be generally desirable to build more complex data types that would embed `Span<T>` and `ReadOnlySpan<T>` instances. Such composite types would have to be structs and would share all the hazards and requirements of `Span<T>`. As a result the safety rules described here should be viewed as applicable to the whole range of ***ref-like types***.

The draft language specification is intended to ensure that values of a ref-like type occur only on the stack.

Generalized `ref-like` types in source code

`ref-like` structs are explicitly marked in the source code using `ref` modifier:

```
ref struct TwoSpans<T>
{
 // can have ref-like instance fields
 public Span<T> first;
 public Span<T> second;
}

// error: arrays of ref-like types are not allowed.
TwoSpans<T>[] arr = null;
```

Designating a struct as ref-like will allow the struct to have ref-like instance fields and will also make all the requirements of ref-like types applicable to the struct.

Metadata representation or ref-like structs

Ref-like structs will be marked with **System.Runtime.CompilerServices.IsRefLikeAttribute** attribute.

The attribute will be added to common base libraries such as `mscorlib`. In a case if the attribute is not available,

compiler will generate an internal one similarly to other embedded-on-demand attributes such as `IsReadOnlyAttribute`.

An additional measure will be taken to prevent the use of ref-like structs in compilers not familiar with the safety rules (this includes C# compilers prior to the one in which this feature is implemented).

Having no other good alternatives that work in old compilers without servicing, an `Obsolete` attribute with a known string will be added to all ref-like structs. Compilers that know how to use ref-like types will ignore this particular form of `Obsolete`.

A typical metadata representation:

```
[IsRefLike]
[Obsolete("Types with embedded references are not supported in this version of your compiler.")]
public struct TwoSpans<T>
{
   // . . . .
}
```

NOTE: it is not the goal to make it so that any use of ref-like types on old compilers fails 100%. That is hard to achieve and is not strictly necessary. For example there would always be a way to get around the `Obsolete` using dynamic code or, for example, creating an array of ref-like types through reflection.

In particular, if user wants to actually put an `Obsolete` or `Deprecated` attribute on a ref-like type, we will have no choice other than not emitting the predefined one since `Obsolete` attribute cannot be applied more than once..

Examples:

```
SpanLikeType M1(ref SpanLikeType x, Span<byte> y)
{
    // this is all valid, unconcerned with stack-referring stuff
    var local = new SpanLikeType(y);
    x = local;
    return x;
}

void Test1(ref SpanLikeType param1, Span<byte> param2)
{
    Span<byte> stackReferring1 = stackalloc byte[10];
    var stackReferring2 = new SpanLikeType(stackReferring1);

    // this is allowed
    stackReferring2 = M1(ref stackReferring2, stackReferring1);

    // this is NOT allowed
    stackReferring2 = M1(ref param1, stackReferring1);

    // this is NOT allowed
    param1 = M1(ref stackReferring2, stackReferring1);

    // this is NOT allowed
    param2 = stackReferring1.Slice(10);

    // this is allowed
    param1 = new SpanLikeType(param2);

    // this is allowed
    stackReferring2 = param1;
}

ref SpanLikeType M2(ref SpanLikeType x)
{
    return ref x;
}

ref SpanLikeType Test2(ref SpanLikeType param1, Span<byte> param2)
{
    Span<byte> stackReferring1 = stackalloc byte[10];
    var stackReferring2 = new SpanLikeType(stackReferring1);

    ref var stackReferring3 = M2(ref stackReferring2);

    // this is allowed
    stackReferring3 = M1(ref stackReferring2, stackReferring1);

    // this is allowed
    M2(ref stackReferring3) = stackReferring2;

    // this is NOT allowed
    M1(ref param1) = stackReferring2;

    // this is NOT allowed
    param1 = stackReferring3;

    // this is NOT allowed
    return ref stackReferring3;

    // this is allowed
    return ref param1;
}
```

Draft language specification

Below we describe a set of safety rules for ref-like types (`ref struct`s) to ensure that values of these types occur only on the stack. A different, simpler set of safety rules would be possible if locals cannot be passed by reference. This specification would also permit the safe reassignment of ref locals.

Overview

We associate with each expression at compile-time the concept of what scope that expression is permitted to escape to, "safe-to-escape". Similarly, for each lvalue we maintain a concept of what scope a reference to it is permitted to escape to, "ref-safe-to-escape". For a given lvalue expression, these may be different.

These are analogous to the "safe to return" of the ref locals feature, but it is more fine-grained. Where the "safe-to-return" of an expression records only whether (or not) it may escape the enclosing method as a whole, the safe-to-escape records which scope it may escape to (which scope it may not escape beyond). The basic safety mechanism is enforced as follows. Given an assignment from an expression E1 with a safe-to-escape scope S1, to an (lvalue) expression E2 with safe-to-escape scope S2, it is an error if S2 is a wider scope than S1. By construction, the two scopes S1 and S2 are in a nesting relationship, because a legal expression is always safe-to-return from some scope enclosing the expression.

For the time being it is sufficient, for the purpose of the analysis, to support just two scopes - external to the method, and top-level scope of the method. That is because ref-like values with inner scopes cannot be created and ref locals do not support re-assignment. The rules, however, can support more than two scope levels.

The precise rules for computing the *safe-to-return* status of an expression, and the rules governing the legality of expressions, follow.

ref-safe-to-escape

The *ref-safe-to-escape* is a scope, enclosing an lvalue expression, to which it is safe for a ref to the lvalue to escape to. If that scope is the entire method, we say that a ref to the lvalue is *safe to return* from the method.

safe-to-escape

The *safe-to-escape* is a scope, enclosing an expression, to which it is safe for the value to escape to. If that scope is the entire method, we say that the value is *safe to return* from the method.

An expression whose type is not a `ref struct` type is *safe-to-return* from the entire enclosing method. Otherwise we refer to the rules below.

Parameters

An lvalue designating a formal parameter is *ref-safe-to-escape* (by reference) as follows:

- If the parameter is a `ref`, `out`, or `in` parameter, it is *ref-safe-to-escape* from the entire method (e.g. by a `return ref` statement); otherwise
- If the parameter is the `this` parameter of a struct type, it is *ref-safe-to-escape* to the top-level scope of the method (but not from the entire method itself); Sample
- Otherwise the parameter is a value parameter, and it is *ref-safe-to-escape* to the top-level scope of the method (but not from the method itself).

An expression that is an rvalue designating the use of a formal parameter is *safe-to-escape* (by value) from the entire method (e.g. by a `return` statement). This applies to the `this` parameter as well.

Locals

An lvalue designating a local variable is *ref-safe-to-escape* (by reference) as follows:

- If the variable is a `ref` variable, then its *ref-safe-to-escape* is taken from the *ref-safe-to-escape* of its initializing expression; otherwise
- The variable is *ref-safe-to-escape* the scope in which it was declared.

An expression that is an rvalue designating the use of a local variable is *safe-to-escape* (by value) as follows:

- But the general rule above, a local whose type is not a `ref struct` type is *safe-to-return* from the entire enclosing method.
- If the variable is an iteration variable of a `foreach` loop, then the variable's *safe-to-escape* scope is the same as the *safe-to-escape* of the `foreach` loop's expression.
- A local of `ref struct` type and uninitialized at the point of declaration is *safe-to-return* from the entire enclosing method.
- Otherwise the variable's type is a `ref struct` type, and the variable's declaration requires an initializer. The variable's *safe-to-escape* scope is the same as the *safe-to-escape* of its initializer.

Field reference

An lvalue designating a reference to a field, `e.F`, is *ref-safe-to-escape* (by reference) as follows:

- If `e` is of a reference type, it is *ref-safe-to-escape* from the entire method; otherwise
- If `e` is of a value type, its *ref-safe-to-escape* is taken from the *ref-safe-to-escape* of `e`.

An rvalue designating a reference to a field, `e.F`, has a *safe-to-escape* scope that is the same as the *safe-to-escape* of `e`.

Operators including `?:`

The application of a user-defined operator is treated as a method invocation.

For an operator that yields an rvalue, such as `e1 + e2` or `c ? e1 : e2`, the *safe-to-escape* of the result is the narrowest scope among the *safe-to-escape* of the operands of the operator. As a consequence, for a unary operator that yields an rvalue, such as `+e`, the *safe-to-escape* of the result is the *safe-to-escape* of the operand.

For an operator that yields an lvalue, such as `c ? ref e1 : ref e2`

- the *ref-safe-to-escape* of the result is the narrowest scope among the *ref-safe-to-escape* of the operands of the operator.
- the *safe-to-escape* of the operands must agree, and that is the *safe-to-escape* of the resulting lvalue.

Method invocation

An lvalue resulting from a ref-returning method invocation `e1.M(e2, ...)` is *ref-safe-to-escape* the smallest of the following scopes:

- The entire enclosing method
- the *ref-safe-to-escape* of all `ref` and `out` argument expressions (excluding the receiver)
- For each `in` parameter of the method, if there is a corresponding expression that is an lvalue, its *ref-safe-to-escape*, otherwise the nearest enclosing scope
- the *safe-to-escape* of all argument expressions (including the receiver)

> Note: the last bullet is necessary to handle code such as
>
> ```
> var sp = new Span(...)
> return ref sp[0];
> ```
>
> or
>
> ```
> return ref M(sp, 0);
> ```

An rvalue resulting from a method invocation `e1.M(e2, ...)` is *safe-to-escape* from the smallest of the following scopes:

- The entire enclosing method
- the *safe-to-escape* of all argument expressions (including the receiver)

An Rvalue

An rvalue is *ref-safe-to-escape* from the nearest enclosing scope. This occurs for example in an invocation such as `M(ref d.Length)` where `d` is of type `dynamic`. It is also consistent with (and perhaps subsumes) our handling of arguments corresponding to `in` parameters.

Property invocations

A property invocation (either `get` or `set`) it treated as a method invocation of the underlying method by the above rules.

`stackalloc`

A stackalloc expression is an rvalue that is *safe-to-escape* to the top-level scope of the method (but not from the entire method itself).

Constructor invocations

A `new` expression that invokes a constructor obeys the same rules as a method invocation that is considered to return the type being constructed.

In addition *safe-to-escape* is no wider than the smallest of the *safe-to-escape* of all arguments/operands of the object initializer expressions, recursively, if initializer is present.

Span constructor

The language relies on `Span<T>` not having a constructor of the following form:

```
void Example(ref int x)
{
    // Create a span of length one
    var span = new Span<int>(ref x);
}
```

Such a constructor makes `Span<T>` which are used as fields indistinguishable from a `ref` field. The safety rules described in this document depend on `ref` fields not being a valid construct in C# or .NET.

`default` **expressions**

A `default` expression is *safe-to-escape* from the entire enclosing method.

Language Constraints

We wish to ensure that no `ref` local variable, and no variable of `ref struct` type, refers to stack memory or variables that are no longer alive. We therefore have the following language constraints:

- Neither a ref parameter, nor a ref local, nor a parameter or local of a `ref struct` type can be lifted into a lambda or local function.
- Neither a ref parameter nor a parameter of a `ref struct` type may be an argument on an iterator method or an `async` method.
- Neither a ref local, nor a local of a `ref struct` type may be in scope at the point of a `yield return` statement or an `await` expression.
- A `ref struct` type may not be used as a type argument, or as an element type in a tuple type.
- A `ref struct` type may not be the declared type of a field, except that it may be the declared type of an instance field of another `ref struct`.

- A `ref struct` type may not be the element type of an array.
- A value of a `ref struct` type may not be boxed:
  - There is no conversion from a `ref struct` type to the type `object` or the type `System.ValueType`.
  - A `ref struct` type may not be declared to implement any interface
  - No instance method declared in `object` or in `System.ValueType` but not overridden in a `ref struct` type may be called with a receiver of that `ref struct` type.
  - No instance method of a `ref struct` type may be captured by method conversion to a delegate type.
- For a ref reassignment `ref e1 = ref e2`, the *ref-safe-to-escape* of `e2` must be at least as wide a scope as the *ref-safe-to-escape* of `e1`.
- For a ref return statement `return ref e1`, the *ref-safe-to-escape* of `e1` must be *ref-safe-to-escape* from the entire method. (TODO: Do we also need a rule that `e1` must be *safe-to-escape* from the entire method, or is that redundant?)
- For a return statement `return e1`, the *safe-to-escape* of `e1` must be *safe-to-escape* from the entire method.
- For an assignment `e1 = e2`, if the type of `e1` is a `ref struct` type, then the *safe-to-escape* of `e2` must be at least as wide a scope as the *safe-to-escape* of `e1`.
- For a method invocation if there is a `ref` or `out` argument of a `ref struct` type (including the receiver unless the type is `readonly`), with *safe-to-escape* E1, then no argument (including the receiver) may have a narrower *safe-to-escape* than E1. Sample
- A local function or anonymous function may not refer to a local or parameter of `ref struct` type declared in an enclosing scope.

> ***Open Issue:*** We need some rule that permits us to produce an error when needing to spill a stack value of a `ref struct` type at an await expression, for example in the code
>
> ```
> Foo(new Span<int>(...), await e2);
> ```

Explanations

These explanations and samples help explain why many of the safety rules above exist

Method Arguments Must Match

When invoking a method where there is an `out` or `ref` parameter that is a `ref struct` then all of the `ref struct` parameters need to have the same lifetime. This is necessary because C# must make all of its decisions around lifetime safety based on the information available in the signature of the method and the lifetime of the values at the call site.

When there are `ref` parameters that are `ref struct` then there is the potential that they could swap around their contents. Hence at the call site we must ensure all of these **potential** swaps are compatible. If the language didn't enforce that then it will allow for bad code like the following.

```
void M1(ref Span<int> s1)
{
    Span<int> s2 = stackalloc int[1];
    Swap(ref s1, ref s2);
}

void Swap(ref Span<int> x, ref Span<int> y)
{
    // This will effectively assign the stackalloc to the s1 parameter and allow it
    // to escape to the caller of M1
    ref x = ref y;
}
```

This analysis of `ref` parameters includes the receiver in instance methods. This is necessary because it can be used to store values passed in as parameters, just as a `ref` parameter could`. This means with mismatched lifetimes you could create a type safety hole in the following way:

```
ref struct S
{
    public Span<int> Span;

    public void Set(Span<int> span)
    {
        Span = span;
    }
}

void Broken(ref S s)
{
    Span<int> span = stackalloc int[1];

    // The result of a stackalloc is now stored in s.Span and escaped to the caller
    // of Broken
    s.Set(span);
}
```

For the purpose of this analysis the receiver is considered an `in`, not a `ref`, if the type is a `readonly struct`. In that case the receiver cannot be used to store values from other parameters, it is effectively an `in` parameter for analysis purposes. Hence the same example above is legal when `S` is `readonly` because the `span` cannot be stored anywhere.

Struct This Escape

When it comes to span safety rules, the `this` value in an instance member is modeled as a parameter to the member. Now for a `struct` the type of `this` is actually `ref S` where in a `class` it's simply `S` (for members of a `class / struct` named S).

Yet `this` has different escaping rules than other `ref` parameters. Specifically it is not ref-safe-to-escape while other parameters are:

```
ref struct S
{
    int Field;

    // Illegal because `this` isn't safe to escape as ref
    ref int Get() => ref Field;

    // Legal
    ref int GetParam(ref int p) => ref p;
}
```

The reason for this restriction actually has little to do with `struct` member invocation. There are some rules that need to be worked out with respect to member invocation on `struct` members where the receiver is an rvalue. But that is very approachable.

The reason for this restriction is actually about interface invocation. Specifically it comes down to whether or not the following sample should or should not compile;

```
interface I1
{
    ref int Get();
}

ref int Use<T>(T p)
    where T : I1
{
    return ref p.Get();
}
```

Consider the case where `T` is instantiated as a `struct`. If the `this` parameter is ref-safe-to-escape then the return of `p.Get` could point to the stack (specifically it could be a field inside of the instantiated type of `T`). That means the language could not allow this sample to compile as it could be returning a `ref` to a stack location. On the other hand if `this` is not ref-safe-to-escape then `p.Get` cannot refer to the stack and hence it's safe to return.

This is why the escapability of `this` in a `struct` is really all about interfaces. It can absolutely be made to work but it has a trade off. The design eventually came down in favor of making interfaces more flexible.

There is potential for us to relax this in the future though.

Future Considerations

Length one Span<T> over ref values

Though not legal today there are cases where creating a length one `Span<T>` instance over a value would be beneficial:

```
void RefExample()
{
    int x = ...;

    // Today creating a length one Span<int> requires a stackalloc and a new
    // local
    Span<int> span1 = stackalloc [] { x };
    Use(span1);
    x = span1[0];

    // Simpler to just allow length one span
    var span2 = new Span<int>(ref x);
    Use(span2);
}
```

This feature gets more compelling if we lift the restrictions on fixed sized buffers as it would allow for `Span<T>` instances of even greater length.

If there is ever a need to go down this path then the language could accommodate this by ensuring such `Span<T>` instances were downward facing only. That is they were only ever *safe-to-escape* to the scope in which they were created. This ensure the language never had to consider a `ref` value escaping a method via a `ref struct` return or field of `ref struct`. This would likely also require further changes to recognize such

constructors as capturing a `ref` parameter in this way though.

Non-trailing named arguments

18.12.2021 • 3 minutes to read

Summary

Allow named arguments to be used in non-trailing position, as long as they are used in their correct position. For example: `DoSomething(isEmployed:true, name, age);`.

Motivation

The main motivation is to avoid typing redundant information. It is common to name an argument that is a literal (such as `null`, `true`) for the purpose of clarifying the code, rather than of passing arguments out-of-order. That is currently disallowed (`CS1738`) unless all the following arguments are also named.

```
DoSomething(isEmployed:true, name, age); // currently disallowed, even though all arguments are in position
// CS1738 "Named argument specifications must appear after all fixed arguments have been specified"
```

Some additional examples:

```
public void DoSomething(bool isEmployed, string personName, int personAge) { ... }

DoSomething(isEmployed:true, name, age); // currently CS1738, but would become legal
DoSomething(true, personName:name, age); // currently CS1738, but would become legal
DoSomething(name, isEmployed:true, age); // remains illegal
DoSomething(name, age, isEmployed:true); // remains illegal
DoSomething(true, personAge:age, personName:name); // already legal
```

This would also work with params:

```
public class Task
{
    public static Task When(TaskStatus all, TaskStatus any, params Task[] tasks);
}
Task.When(all: TaskStatus.RanToCompletion, any: TaskStatus.Faulted, task1, task2)
```

Detailed design

In §7.5.1 (Argument lists), the spec currently says:

> An *argument* with an *argument-name* is referred to as a **named argument**, whereas an *argument* without an *argument-name* is a **positional argument**. It is an error for a positional argument to appear after a named argument in an *argument-list*.

The proposal is to remove this error and update the rules for finding the corresponding parameter for an argument (§7.5.1.1):

Arguments in the argument-list of instance constructors, methods, indexers and delegates:

- [existing rules]
- An unnamed argument corresponds to no parameter when it is after an out-of-position named argument or

a named params argument.

In particular, this prevents invoking `void M(bool a = true, bool b = true, bool c = true, );` with `M(c: false, valueB);`. The first argument is used out-of-position (the argument is used in first position, but the parameter named "c" is in third position), so the following arguments should be named.

In other words, non-trailing named arguments are only allowed when the name and the position result in finding the same corresponding parameter.

Drawbacks

This proposal exacerbates existing subtleties with named arguments in overload resolution. For instance:

```
void M(int x, int y) { }
void M<T>(T y, int x) { }

void M2()
{
    M(3, 4);
    M(y: 3, x: 4); // Invokes M(int, int)
    M(y: 3, 4); // Invokes M<T>(T, int)
}
```

You could get this situation today by swapping the parameters:

```
void M(int y, int x) { }
void M<T>(int x, T y) { }

void M2()
{
    M(3, 4);
    M(x: 3, y: 4); // Invokes M(int, int)
    M(3, y: 4); // Invokes M<T>(int, T)
}
```

Similarly, if you have two methods `void M(int a, int b)` and `void M(int x, string y)`, the mistaken invocation `M(x: 1, 2)` will produce a diagnostic based on the second overload ("cannot convert from 'int' to 'string'"). This problem already exists when the named argument is used in a trailing position.

Alternatives

There are a couple of alternatives to consider:

- The status quo
- Providing IDE assistance to fill-in all the names of trailing arguments when you type specific a name in the middle.

Both of those suffer from more verbosity, as they introduce multiple named arguments even if you just need one name of a literal at the beginning of the argument list.

Unresolved questions

Design meetings

The feature was briefly discussed in LDM on May 16th 2017, with approval in principle (ok to move to proposal/prototype). It was also briefly discussed on June 28th 2017.

Relates to initial discussion https://github.com/dotnet/csharplang/issues/518 Relates to championed issue https://github.com/dotnet/csharplang/issues/570

private protected

18.12.2021 • 9 minutes to read

Summary

Expose the CLR `protectedAndInternal` accessibility level in C# as `private protected`.

Motivation

There are many circumstances in which an API contains members that are only intended to be implemented and used by subclasses contained in the assembly that provides the type. While the CLR provides an accessibility level for that purpose, it is not available in C#. Consequently API owners are forced to either use `internal` protection and self-discipline or a custom analyzer, or to use `protected` with additional documentation explaining that, while the member appears in the public documentation for the type, it is not intended to be part of the public API. For examples of the latter, see members of Roslyn's `CSharpCompilationOptions` whose names start with `Common`.

Directly providing support for this access level in C# enables these circumstances to be expressed naturally in the language.

Detailed design

`private protected` access modifier

We propose to add a new access modifier combination `private protected` (which can appear in any order among the modifiers). This maps to the CLR notion of protectedAndInternal, and borrows the same syntax currently used in C++/CLI.

A member declared `private protected` can be accessed within a subclass of its container if that subclass is in the same assembly as the member.

We modify the language specification as follows (additions in bold). Section numbers are not shown below as they may vary depending on which version of the specification it is integrated into.

> The declared accessibility of a member can be one of the following:

- Public, which is selected by including a public modifier in the member declaration. The intuitive meaning of public is "access not limited".
- Protected, which is selected by including a protected modifier in the member declaration. The intuitive meaning of protected is "access limited to the containing class or types derived from the containing class".
- Internal, which is selected by including an internal modifier in the member declaration. The intuitive meaning of internal is "access limited to this assembly".
- Protected internal, which is selected by including both a protected and an internal modifier in the member declaration. The intuitive meaning of protected internal is "accessible within this assembly as well as types derived from the containing class".
- **Private protected, which is selected by including both a private and a protected modifier in the member declaration. The intuitive meaning of private protected is "accessible within this assembly by types derived from the containing class".**

> Depending on the context in which a member declaration takes place, only certain types of declared accessibility are permitted. Furthermore, when a member declaration does not include any access modifiers, the context in which the declaration takes place determines the default declared accessibility.

- Namespaces implicitly have public declared accessibility. No access modifiers are allowed on namespace declarations.
- Types declared directly in compilation units or namespaces (as opposed to within other types) can have public or internal declared accessibility and default to internal declared accessibility.
- Class members can have any of the five kinds of declared accessibility and default to private declared accessibility. [Note: A type declared as a member of a class can have any of the five kinds of declared accessibility, whereas a type declared as a member of a namespace can have only public or internal declared accessibility. end note]
- Struct members can have public, internal, or private declared accessibility and default to private declared accessibility because structs are implicitly sealed. Struct members introduced in a struct (that is, not inherited by that struct) cannot have protected*,* ~~or~~ protected internal**, or private protected** declared accessibility. [Note: A type declared as a member of a struct can have public, internal, or private declared accessibility, whereas a type declared as a member of a namespace can have only public or internal declared accessibility. end note]
- Interface members implicitly have public declared accessibility. No access modifiers are allowed on interface member declarations.
- Enumeration members implicitly have public declared accessibility. No access modifiers are allowed on enumeration member declarations.

> The accessibility domain of a nested member M declared in a type T within a program P, is defined as follows (noting that M itself might possibly be a type):

- If the declared accessibility of M is public, the accessibility domain of M is the accessibility domain of T.
- If the declared accessibility of M is protected internal, let D be the union of the program text of P and the program text of any type derived from T, which is declared outside P. The accessibility domain of M is the intersection of the accessibility domain of T with D.
- **If the declared accessibility of M is private protected, let D be the intersection of the program text of P and the program text of any type derived from T. The accessibility domain of M is the intersection of the accessibility domain of T with D.**
- If the declared accessibility of M is protected, let D be the union of the program text of T and the program text of any type derived from T. The accessibility domain of M is the intersection of the accessibility domain of T with D.
- If the declared accessibility of M is internal, the accessibility domain of M is the intersection of the accessibility domain of T with the program text of P.
- If the declared accessibility of M is private, the accessibility domain of M is the program text of T.

> When a protected **or private protected** instance member is accessed outside the program text of the class in which it is declared, and when a protected internal instance member is accessed outside the program text of the program in which it is declared, the access shall take place within a class declaration that derives from the class in which it is declared. Furthermore, the access is required to take place through an instance of that derived class type or a class type constructed from it. This restriction prevents one derived class from accessing protected members of other derived classes, even when the members are inherited from the same base class.

> The permitted access modifiers and the default access for a type declaration depend on the context in which the declaration takes place (§9.5.2):

- Types declared in compilation units or namespaces can have public or internal access. The default is internal access.
- Types declared in classes can have public, protected internal, **private protected**, protected, internal, or private access. The default is private access.
- Types declared in structs can have public, internal, or private access. The default is private access.

> A static class declaration is subject to the following restrictions:

- A static class shall not include a sealed or abstract modifier. (However, since a static class cannot be instantiated or derived from, it behaves as if it was both sealed and abstract.)
- A static class shall not include a class-base specification (§16.2.5) and cannot explicitly specify a base class or a list of implemented interfaces. A static class implicitly inherits from type object.
- A static class shall only contain static members (§16.4.8). [Note: All constants and nested types are classified as static members. end note]
- A static class shall not have members with protected**, private protected** or protected internal declared accessibility.

> It is a compile-time error to violate any of these restrictions.

> A class-member-declaration can have any one of the ~~five~~**six** possible kinds of declared accessibility (§9.5.2): public, **private protected**, protected internal, protected, internal, or private. Except for the protected internal **and private protected** combinations, it is a compile-time error to specify more than one access modifier. When a class-member-declaration does not include any access modifiers, private is assumed.

> Non-nested types can have public or internal declared accessibility and have internal declared accessibility by default. Nested types can have these forms of declared accessibility too, plus one or more additional forms of declared accessibility, depending on whether the containing type is a class or struct:

- A nested type that is declared in a class can have any of ~~five~~**six** forms of declared accessibility (public, **private protected**, protected internal, protected, internal, or private) and, like other class members, defaults to private declared accessibility.
- A nested type that is declared in a struct can have any of three forms of declared accessibility (public, internal, or private) and, like other struct members, defaults to private declared accessibility.

> The method overridden by an override declaration is known as the overridden base method For an override method M declared in a class C, the overridden base method is determined by examining each base class type of C, starting with the direct base class type of C and continuing with each successive direct base class type, until in a given base class type at least one accessible method is located which has the same signature as M after substitution of type arguments. For the purposes of locating the overridden base method, a method is considered accessible if it is public, if it is protected, if it is protected internal, or if it is **either** internal **or private protected** and declared in the same program as C.

> The use of accessor-modifiers is governed by the following restrictions:

- An accessor-modifier shall not be used in an interface or in an explicit interface member implementation.
- For a property or indexer that has no override modifier, an accessor-modifier is permitted only if the property or indexer has both a get and set accessor, and then is permitted only on one of those accessors.
- For a property or indexer that includes an override modifier, an accessor shall match the accessor-modifier, if any, of the accessor being overridden.
- The accessor-modifier shall declare an accessibility that is strictly more restrictive than the declared accessibility of the property or indexer itself. To be precise:
  - If the property or indexer has a declared accessibility of public, the accessor-modifier may be either **private protected**, , protected internal, internal, protected, or private.
  - If the property or indexer has a declared accessibility of protected internal, the accessor-modifier may be either **private protected**, internal, protected, or private.
  - If the property or indexer has a declared accessibility of internal or protected, the accessor-modifier shall be **either private protected or** private.
  - **If the property or indexer has a declared accessibility of private protected, the accessor-modifier shall be private.**
  - If the property or indexer has a declared accessibility of private, no accessor-modifier may be used.

> Since inheritance isn't supported for structs, the declared accessibility of a struct member cannot be protected, **private protected**, or protected internal.

Drawbacks

As with any language feature, we must question whether the additional complexity to the language is repaid in the additional clarity offered to the body of C# programs that would benefit from the feature.

Alternatives

An alternative would be the provision of an API combining an attribute and an analyzer. The attribute is placed by the programmer on an `internal` member to indicates that the member is intended to be used only in subclasses, and the analyzer checks that those restrictions are obeyed.

Unresolved questions

The implementation is largely complete. The only open work item is drafting a corresponding specification for VB.

Design meetings

TBD

Conditional ref expressions

18.12.2021 • 2 minutes to read

The pattern of binding a ref variable to one or another expression conditionally is not currently expressible in C#.

The typical workaround is to introduce a method like:

```
ref T Choice(bool condition, ref T consequence, ref T alternative)
{
    if (condition)
    {
         return ref consequence;
    }
    else
    {
         return ref alternative;
    }
}
```

Note that this is not an exact replacement of a ternary since all arguments must be evaluated at the call site.

The following will not work as expected:

```
       // will crash with NRE because 'arr[0]' will be executed unconditionally
      ref var r = ref Choice(arr != null, ref arr[0], ref otherArr[0]);
```

The proposed syntax would look like:

```
     <condition> ? ref <consequence> : ref <alternative>;
```

The above attempt with "Choice" can be *correctly* written using ref ternary as:

```
     ref var r = ref (arr != null ? ref arr[0]: ref otherArr[0]);
```

The difference from Choice is that consequence and alternative expressions are accessed in a *truly* conditional manner, so we do not see a crash if `arr == null`

The ternary ref is just a ternary where both alternative and consequence are refs. It will naturally require that consequence/alternative operands are LValues. It will also require that consequence and alternative have types that are identity convertible to each other.

The type of the expression will be computed similarly to the one for the regular ternary. I.E. in a case if consequence and alternative have identity convertible, but different types, the existing type-merging rules will apply.

Safe-to-return will be assumed conservatively from the conditional operands. If either is unsafe to return the whole thing is unsafe to return.

Ref ternary is an LValue and as such it can be passed/assigned/returned by reference;

```
// pass by reference
foo(ref (arr != null ? ref arr[0]: ref otherArr[0]));

// return by reference
return ref (arr != null ? ref arr[0]: ref otherArr[0]);
```

Being an LValue, it can also be assigned to.

```
// assign to
(arr != null ? ref arr[0]: ref otherArr[0]) = 1;
```

Ref ternary can be used in a regular (not ref) context as well. Although it would not be common since you could as well just use a regular ternary.

```
int x = (arr != null ? ref arr[0]: ref otherArr[0]);
```

Implementation notes:

The complexity of the implementation would seem to be the size of a moderate-to-large bug fix. - I.E not very expensive. I do not think we need any changes to the syntax or parsing. There is no effect on metadata or interop. The feature is completely expression based. No effect on debugging/PDB either

Allow digit separator after 0b or 0x

18.12.2021 • 2 minutes to read

In C# 7.2, we extend the set of places that digit separators (the underscore character) can appear in integral literals. Beginning in C# 7.0, separators are permitted between the digits of a literal. Now, in C# 7.2, we also permit digit separators before the first significant digit of a binary or hexadecimal literal, after the prefix.

```
123      // permitted in C# 1.0 and later
1_2_3    // permitted in C# 7.0 and later
0x1_2_3  // permitted in C# 7.0 and later
0b101    // binary literals added in C# 7.0
0b1_0_1  // permitted in C# 7.0 and later

// in C# 7.2, _ is permitted after the `0x` or `0b`
0x_1_2   // permitted in C# 7.2 and later
0b_1_0_1 // permitted in C# 7.2 and later
```

We do not permit a decimal integer literal to have a leading underscore. A token such as `_123` is an identifier.

Unmanaged type constraint

18.12.2021 • 4 minutes to read

Summary

The unmanaged constraint feature will give language enforcement to the class of types known as "unmanaged types" in the C# language spec. This is defined in section 18.2 as a type which is not a reference type and doesn't contain reference type fields at any level of nesting.

Motivation

The primary motivation is to make it easier to author low level interop code in C#. Unmanaged types are one of the core building blocks for interop code, yet the lack of support in generics makes it impossible to create re-usable routines across all unmanaged types. Instead developers are forced to author the same boiler plate code for every unmanaged type in their library:

```
int Hash(Point point) { ... }
int Hash(TimeSpan timeSpan) { ... }
```

To enable this type of scenario the language will be introducing a new constraint: unmanaged:

```
void Hash<T>(T value) where T : unmanaged
{
    ...
}
```

This constraint can only be met by types which fit into the unmanaged type definition in the C# language spec. Another way of looking at it is that a type satisfies the unmanaged constraint if it can also be used as a pointer.

```
Hash(new Point()); // Okay
Hash(42); // Okay
Hash("hello") // Error: Type string does not satisfy the unmanaged constraint
```

Type parameters with the unmanaged constraint can use all the features available to unmanaged types: pointers, fixed, etc ...

```
void Hash<T>(T value) where T : unmanaged
{
    // Okay
    fixed (T* p = &value)
    {
        ...
    }
}
```

This constraint will also make it possible to have efficient conversions between structured data and streams of bytes. This is an operation that is common in networking stacks and serialization layers:

```
Span<byte> Convert<T>(ref T value) where T : unmanaged
{
    ...
}
```

Such routines are advantageous because they are provably safe at compile time and allocation free. Interop authors today can not do this (even though it's at a layer where perf is critical). Instead they need to rely on allocating routines that have expensive runtime checks to verify values are correctly unmanaged.

Detailed design

The language will introduce a new constraint named `unmanaged`. In order to satisfy this constraint a type must be a struct and all the fields of the type must fall into one of the following categories:

- Have the type `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, `IntPtr` or `UIntPtr`.
- Be any `enum` type.
- Be a pointer type.
- Be a user defined struct that satisfies the `unmanaged` constraint.

Compiler generated instance fields, such as those backing auto-implemented properties, must also meet these constraints.

For example:

```
// Unmanaged type
struct Point
{
    int X;
    int Y {get; set;}
}

// Not an unmanaged type
struct Student
{
    string FirstName;
    string LastName;
}
```

The `unmanaged` constraint cannot be combined with `struct`, `class` or `new()`. This restriction derives from the fact that `unmanaged` implies `struct` hence the other constraints do not make sense.

The `unmanaged` constraint is not enforced by CLR, only by the language. To prevent mis-use by other languages, methods which have this constraint will be protected by a mod-req. This will prevent other languages from using type arguments which are not unmanaged types.

The token `unmanaged` in the constraint is not a keyword, nor a contextual keyword. Instead it is like `var` in that it is evaluated at that location and will either:

- Bind to user defined or referenced type named `unmanaged`: This will be treated just as any other named type constraint is treated.
- Bind to no type: This will be interpreted as the `unmanaged` constraint.

In the case there is a type named `unmanaged` and it is available without qualification in the current context, then there will be no way to use the `unmanaged` constraint. This parallels the rules surrounding the feature `var` and user defined types of the same name.

Drawbacks

The primary drawback of this feature is that it serves a small number of developers: typically low level library authors or frameworks. Hence it's spending precious language time for a small number of developers.

Yet these frameworks are often the basis for the majority of .NET applications out there. Hence performance / correctness wins at this level can have a ripple effect on the .NET ecosystem. This makes the feature worth considering even with the limited audience.

Alternatives

There are a couple of alternatives to consider:

- The status quo: The feature is not justified on its own merits and developers continue to use the implicit opt in behavior.

Questions

Metadata Representation

The F# language encodes the constraint in the signature file which means C# cannot re-use their representation. A new attribute will need to be chosen for this constraint. Additionally a method which has this constraint must be protected by a mod-req.

Blittable vs. Unmanaged

The F# language has a very similar feature which uses the keyword unmanaged. The blittable name comes from the use in Midori. May want to look to precedence here and use unmanaged instead.

Resolution The language decide to use unmanaged

Verifier

Does the verifier / runtime need to be updated to understand the use of pointers to generic type parameters? Or can it simply work as is without changes?

Resolution No changes needed. All pointer types are simply unverifiable.

Design meetings

n/a

Indexing `fixed` fields should not require pinning regardless of the movable/unmovable context.

18.12.2021 • 2 minutes to read

The change has the size of a bug fix. It can be in 7.3 and does not conflict with whatever direction we take further. This change is only about allowing the following scenario to work even though `s` is moveable. It is already valid when `s` is not moveable.

NOTE: in either case, it still requires `unsafe` context. It is possible to read uninitialized data or even out of range. That is not changing.

```
unsafe struct S
{
    public fixed int myFixedField[10];
}

class Program
{
    static S s;

    unsafe static void Main()
    {
        int p = s.myFixedField[5]; // indexing fixed-size array fields would be ok
    }
}
```

The main "challenge" that I see here is how to explain the relaxation in the spec. In particular, since the following would still need pinning. (because `s` is moveable and we explicitly use the field as a pointer)

```
unsafe struct S
{
    public fixed int myFixedField[10];
}

class Program
{
    static S s;

    unsafe static void Main()
    {
        int* ptr = s.myFixedField; // taking a pointer explicitly still requires pinning.
        int p = ptr[5];
    }
}
```

One reason why we require pinning of the target when it is movable is the artifact of our code generation strategy, - we always convert to an unmanaged pointer and thus force the user to pin via `fixed` statement. However, conversion to unmanaged is unnecessary when doing indexing. The same unsafe pointer math is equally applicable when we have the receiver in the form of a managed pointer. If we do that, then the intermediate ref is managed (GC-tracked) and the pinning is unnecessary.

The change https://github.com/dotnet/roslyn/pull/24966 is a prototype PR that relaxes this requirement.

Pattern-based `fixed` statement

18.12.2021 • 3 minutes to read

Summary

Introduce a pattern that would allow types to participate in `fixed` statements.

Motivation

The language provides a mechanism for pinning managed data and obtain a native pointer to the underlying buffer.

```
fixed(byte* ptr = byteArray)
{
   // ptr is a native pointer to the first element of the array
   // byteArray is protected from being moved/collected by the GC for the duration of this block
}
```

The set of types that can participate in `fixed` is hardcoded and limited to arrays and `System.String`. Hardcoding "special" types does not scale when new primitives such as `ImmutableArray<T>`, `Span<T>`, `Utf8String` are introduced.

In addition, the current solution for `System.String` relies on a fairly rigid API. The shape of the API implies that `System.String` is a contiguous object that embeds UTF16 encoded data at a fixed offset from the object header. Such approach has been found problematic in several proposals that could require changes to the underlying layout. It would be desirable to be able to switch to something more flexible that decouples `System.String` object from its internal representation for the purpose of unmanaged interop.

Detailed design

Pattern

A viable pattern-based “fixed” need to:

- Provide the managed references to pin the instance and to initialize the pointer (preferably this is the same reference)
- Convey unambiguously the type of the unmanaged element (i.e. “char” for “string”)
- Prescribe the behavior in "empty" case when there is nothing to refer to.
- Should not push API authors toward design decisions that hurt the use of the type outside of `fixed`.

I think the above could be satisfied by recognizing a specially named ref-returning member: `ref [readonly] T GetPinnableReference()`.

In order to be used by the `fixed` statement the following conditions must be met:

1. There is only one such member provided for a type.
2. Returns by `ref` or `ref readonly`. (`readonly` is permitted so that authors of immutable/readonly types could implement the pattern without adding writeable API that could be used in safe code)
3. T is an unmanaged type. (since `T*` becomes the pointer type. The restriction will naturally expand if/when

the notion of "unmanaged" is expanded)

4. Returns managed `nullptr` when there is no data to pin – probably the cheapest way to convey emptiness. (note that "" string returns a ref to '\0' since strings are null-terminated)

Alternatively for the `#3` we can allow the result in empty cases be undefined or implementation-specific. That, however, may make the API more dangerous and prone to abuse and unintended compatibility burdens.

Translation

```
fixed(byte* ptr = thing)
{
    // <BODY>
}
```

becomes the following pseudocode (not all expressible in C#)

```
byte* ptr;
// specially decorated "pinned" IL local slot, not visible to user code.
pinned ref byte _pinned;

try
{
    // NOTE: null check is omitted for value types
    // NOTE: `thing` is evaluated only once (temporary is introduced if necessary)
    if (thing != null)
    {
        // obtain and "pin" the reference
        _pinned = ref thing.GetPinnableReference();

        // unsafe cast in IL
        ptr = (byte*)_pinned;
    }
    else
    {
        ptr = default(byte*);
    }

    // <BODY>
}
finally   // finally can be omitted when not observable
{
    // "unpin" the object
    _pinned = nullptr;
}
```

Drawbacks

- GetPinnableReference is intended to be used only in `fixed`, but nothing prevents its use in safe code, so implementor must keep that in mind.

Alternatives

Users can introduce GetPinnableReference or similar member and use it as

```
fixed(byte* ptr = thing.GetPinnableReference())
{
    // <BODY>
}
```

There is no solution for `System.String` if alternative solution is desired.

Unresolved questions

- [ ] Behavior in "empty" state. - `nullptr` or `undefined` ?
- [ ] Should the extension methods be considered ?
- [ ] If a pattern is detected on `System.String`, should it win over ?

Design meetings

None yet.

Ref Local Reassignment

18.12.2021 • 2 minutes to read

In C# 7.3, we add support for rebinding the referent of a ref local variable or a ref parameter.

We add the following to the set of `assignment_operator`s.

```
assignment_operator
    : '=' 'ref'
    ;
```

The `=ref` operator is called the ***ref assignment operator***. It is not a *compound assignment operator*. The left operand must be an expression that binds to a ref local variable, a ref parameter (other than `this`), or an out parameter. The right operand must be an expression that yields an lvalue designating a value of the same type as the left operand.

The right operand must be definitely assigned at the point of the ref assignment.

When the left operand binds to an `out` parameter, it is an error if that `out` parameter has not been definitely assigned at the beginning of the ref assignment operator.

If the left operand is a writeable ref (i.e. it designates anything other than a `ref readonly` local or `in` parameter), then the right operand must be a writeable lvalue.

The ref assignment operator yields an lvalue of the assigned type. It is writeable if the left operand is writeable (i.e. not `ref readonly` or `in`).

The safety rules for this operator are:

- For a ref reassignment `e1 = ref e2`, the *ref-safe-to-escape* of `e2` must be at least as wide a scope as the *ref-safe-to-escape* of `e1`.

Where *ref-safe-to-escape* is defined in Safety for ref-like types

Stackalloc array initializers

18.12.2021 • 2 minutes to read

Summary

Allow array initializer syntax to be used with `stackalloc`.

Motivation

Ordinary arrays can have their elements initialized at creation time. It seems reasonable to allow that in `stackalloc` case.

The question of why such syntax is not allowed with `stackalloc` arises fairly frequently.
See, for example, #1112

Detailed design

Ordinary arrays can be created through the following syntax:

```
new int[3]
new int[3] { 1, 2, 3 }
new int[] { 1, 2, 3 }
new[] { 1, 2, 3 }
```

We should allow stack allocated arrays be created through:

```
stackalloc int[3]    // currently allowed
stackalloc int[3] { 1, 2, 3 }
stackalloc int[] { 1, 2, 3 }
stackalloc[] { 1, 2, 3 }
```

The semantics of all cases is roughly the same as with arrays.
For example: in the last case the element type is inferred from the initializer and must be an "unmanaged" type.

NOTE: the feature is not dependent on the target being a `Span<T>`. It is just as applicable in `T*` case, so it does not seem reasonable to predicate it on `Span<T>` case.

Translation

The naive implementation could just initialize the array right after creation through a series of element-wise assignments.

Similarly to the case with arrays, it might be possible and desirable to detect cases where all or most of the elements are blittable types and use more efficient techniques by copying over the pre-created state of all the constant elements.

Drawbacks

Alternatives

This is a convenience feature. It is possible to just do nothing.

Unresolved questions

Design meetings

None yet.

Auto-Implemented Property Field-Targeted Attributes

18.12.2021 • 2 minutes to read

Summary

This feature intends to allow developers to apply attributes directly to the backing fields of auto-implemented properties.

Motivation

Currently it is not possible to apply attributes to the backing fields of auto-implemented properties. In those cases where the developer must use a field-targeting attribute they are forced to declare the field manually and use the more verbose property syntax. Given that C# has always supported field-targeted attributes on the generated backing field for events it makes sense to extend the same functionality to their property kin.

Detailed design

In short, the following would be legal C# and not produce a warning:

```
[Serializable]
public class Foo
{
    [field: NonSerialized]
    public string MySecret { get; set; }
}
```

This would result in the field-targeted attributes being applied to the compiler-generated backing field:

```
[Serializable]
public class Foo
{
    [NonSerialized]
    private string _mySecretBackingField;

    public string MySecret
    {
        get { return _mySecretBackingField; }
        set { _mySecretBackingField = value; }
    }
}
```

As mentioned, this brings parity with event syntax from C# 1.0 as the following is already legal and behaves as expected:

```
[Serializable]
public class Foo
{
    [field: NonSerialized]
    public event EventHandler MyEvent;
}
```

Drawbacks

There are two potential drawbacks to implementing this change:

1. Attempting to apply an attribute to the field of an auto-implemented property produces a compiler warning that the attributes in that block will be ignored. If the compiler were changed to support those attributes they would be applied to the backing field on a subsequent recompilation which could alter the behavior of the program at runtime.
2. The compiler does not currently validate the AttributeUsage targets of the attributes when attempting to apply them to the field of the auto-implemented property. If the compiler were changed to support field-targeted attributes and the attribute in question cannot be applied to a field the compiler would emit an error instead of a warning, breaking the build.

Alternatives

Unresolved questions

Design meetings

Expression variables in initializers

18.12.2021 • 2 minutes to read

Summary

We extend the features introduced in C# 7 to permit expressions containing expression variables (out variable declarations and declaration patterns) in field initializers, property initializers, ctor-initializers, and query clauses.

Motivation

This completes a couple of the rough edges left in the C# language due to lack of time.

Detailed design

We remove the restriction preventing the declaration of expression variables (out variable declarations and declaration patterns) in a ctor-initializer. Such a declared variable is in scope throughout the body of the constructor.

We remove the restriction preventing the declaration of expression variables (out variable declarations and declaration patterns) in a field or property initializer. Such a declared variable is in scope throughout the initializing expression.

We remove the restriction preventing the declaration of expression variables (out variable declarations and declaration patterns) in a query expression clause that is translated into the body of a lambda. Such a declared variable is in scope throughout that expression of the query clause.

Drawbacks

None.

Alternatives

The appropriate scope for expression variables declared in these contexts is not obvious, and deserves further LDM discussion.

Unresolved questions

- [ ] What is the appropriate scope for these variables?

Design meetings

None.

Support for == and != on tuple types

18.12.2021 • 4 minutes to read

Allow expressions `t1 == t2` where `t1` and `t2` are tuple or nullable tuple types of same cardinality, and evaluate them roughly as `temp1.Item1 == temp2.Item1 && temp1.Item2 == temp2.Item2` (assuming `var temp1 = t1; var temp2 = t2;`).

Conversely it would allow `t1 != t2` and evaluate it as `temp1.Item1 != temp2.Item1 || temp1.Item2 != temp2.Item2`.

In the nullable case, additional checks for `temp1.HasValue` and `temp2.HasValue` are used. For instance, `nullableT1 == nullableT2` evaluates as `temp1.HasValue == temp2.HasValue ? (temp1.HasValue ? ... : true) : false`.

When an element-wise comparison returns a non-bool result (for instance, when a non-bool user-defined `operator ==` or `operator !=` is used, or in a dynamic comparison), then that result will be either converted to `bool` or run through `operator true` or `operator false` to get a `bool`. The tuple comparison always ends up returning a `bool`.

As of C# 7.2, such code produces an error (`error CS0019: Operator '==' cannot be applied to operands of type '(...)' and '(...)'`), unless there is a user-defined `operator==`.

Details

When binding the `==` (or `!=`) operator, the existing rules are: (1) dynamic case, (2) overload resolution, and (3) fail. This proposal adds a tuple case between (1) and (2): if both operands of a comparison operator are tuples (have tuple types or are tuple literals) and have matching cardinality, then the comparison is performed element-wise. This tuple equality is also lifted onto nullable tuples.

Both operands (and, in the case of tuple literals, their elements) are evaluated in order from left to right. Each pair of elements is then used as operands to bind the operator `==` (or `!=`), recursively. Any elements with compile-time type `dynamic` cause an error. The results of those element-wise comparisons are used as operands in a chain of conditional AND (or OR) operators.

For instance, in the context of `(int, (int, int)) t1, t2;`, `t1 == (1, (2, 3))` would evaluate as

```
temp1.Item1 == temp2.Item1 && temp1.Item2.Item1 == temp2.Item2.Item1 && temp1.Item2.Item2 ==
temp2.Item2.Item2
```

.

When a tuple literal is used as operand (on either side), it receives a converted tuple type formed by the element-wise conversions which are introduced when binding the operator `==` (or `!=`) element-wise.

For instance, in `(1L, 2, "hello") == (1, 2L, null)`, the converted type for both tuple literals is `(long, long, string)` and the second literal has no natural type.

Deconstruction and conversions to tuple

In `(a, b) == x`, the fact that `x` can deconstruct into two elements does not play a role. That could conceivably be in a future proposal, although it would raise questions about `x == y` (is this a simple comparison or an element-wise comparison, and if so using what cardinality?). Similarly, conversions to tuple play no role.

Tuple element names

When converting a tuple literal, we warn when an explicit tuple element name was provided in the literal, but it doesn't match the target tuple element name. We use the same rule in tuple comparison, so that assuming `(int a, int b) t` we warn on `d` in `t == (c, d: 0)`.

Non-bool element-wise comparison results

If an element-wise comparison is dynamic in a tuple equality, we use a dynamic invocation of the operator `false` and negate that to get a `bool` and continue with further element-wise comparisons.

If an element-wise comparison returns some other non-bool type in a tuple equality, there are two cases:

- if the non-bool type converts to `bool`, we apply that conversion,
- if there is no such conversion, but the type has an operator `false`, we'll use that and negate the result.

In a tuple inequality, the same rules apply except that we'll use the operator `true` (without negation) instead of the operator `false`.

Those rules are similar to the rules involved for using a non-bool type in an `if` statement and some other existing contexts.

Evaluation order and special cases

The left-hand-side value is evaluated first, then the right-hand-side value, then the element-wise comparisons from left to right (including conversions, and with early exit based on existing rules for conditional AND/OR operators).

For instance, if there is a conversion from type `A` to type `B` and a method `(A, A) GetTuple()`, evaluating `(new A(1), (new B(2), new B(3))) == (new B(4), GetTuple())` means:

- `new A(1)`
- `new B(2)`
- `new B(3)`
- `new B(4)`
- `GetTuple()`
- then the element-wise conversions and comparisons and conditional logic is evaluated (convert `new A(1)` to type `B`, then compare it with `new B(4)`, and so on).

Comparing `null` to `null`

This is a special case from regular comparisons, that carries over to tuple comparisons. The `null == null` comparison is allowed, and the `null` literals do not get any type. In tuple equality, this means, `(0, null) == (0, null)` is also allowed and the `null` and tuple literals don't get a type either.

Comparing a nullable struct to `null` without `operator==`

This is another special case from regular comparisons, that carries over to tuple comparisons. If you have a `struct S` without `operator==`, the `(S?)x == null` comparison is allowed, and it is interpreted as `((S?).x).HasValue`. In tuple equality, the same rule is applied, so `(0, (S?)x) == (0, null)` is allowed.

Compatibility

If someone wrote their own `ValueTuple` types with an implementation of the comparison operator, it would have previously been picked up by overload resolution. But since the new tuple case comes before overload resolution, we would handle this case with tuple comparison instead of relying on the user-defined comparison.

Relates to relational and type testing operators Relates to #190

Improved overload candidates

18.12.2021 • 2 minutes to read

Summary

The overload resolution rules have been updated in nearly every C# language update to improve the experience for programmers, making ambiguous invocations select the "obvious" choice. This has to be done carefully to preserve backward compatibility, but since we are usually resolving what would otherwise be error cases, these enhancements usually work out nicely.

1. When a method group contains both instance and static members, we discard the instance members if invoked without an instance receiver or context, and discard the static members if invoked with an instance receiver. When there is no receiver, we include only static members in a static context, otherwise both static and instance members. When the receiver is ambiguously an instance or type due to a color-color situation, we include both. A static context, where an implicit this instance receiver cannot be used, includes the body of members where no this is defined, such as static members, as well as places where this cannot be used, such as field initializers and constructor-initializers.
2. When a method group contains some generic methods whose type arguments do not satisfy their constraints, these members are removed from the candidate set.
3. For a method group conversion, candidate methods whose return type doesn't match up with the delegate's return type are removed from the set.

Nullable reference types in C#

18.12.2021 • 7 minutes to read

The goal of this feature is to:

- Allow developers to express whether a variable, parameter or result of a reference type is intended to be null or not.
- Provide warnings when such variables, parameters and results are not used according to that intent.

Expression of intent

The language already contains the `T?` syntax for value types. It is straightforward to extend this syntax to reference types.

It is assumed that the intent of an unadorned reference type `T` is for it to be non-null.

Checking of nullable references

A flow analysis tracks nullable reference variables. Where the analysis deems that they would not be null (e.g. after a check or an assignment), their value will be considered a non-null reference.

A nullable reference can also explicitly be treated as non-null with the postfix `x!` operator (the "damnit" operator), for when flow analysis cannot establish a non-null situation that the developer knows is there.

Otherwise, a warning is given if a nullable reference is dereferenced, or is converted to a non-null type.

A warning is given when converting from `S[]` to `T?[]` and from `S?[]` to `T[]`.

A warning is given when converting from `C<S>` to `C<T?>` except when the type parameter is covariant (`out`), and when converting from `C<S?>` to `C<T>` except when the type parameter is contravariant (`in`).

A warning is given on `C<T?>` if the type parameter has non-null constraints.

Checking of non-null references

A warning is given if a null literal is assigned to a non-null variable or passed as a non-null parameter.

A warning is also given if a constructor does not explicitly initialize non-null reference fields.

We cannot adequately track that all elements of an array of non-null references are initialized. However, we could issue a warning if no element of a newly created array is assigned to before the array is read from or passed on. That might handle the common case without being too noisy.

We need to decide whether `default(T)` generates a warning, or is simply treated as being of the type `T?`.

Metadata representation

Nullability adornments should be represented in metadata as attributes. This means that downlevel compilers will ignore them.

We need to decide if only nullable annotations are included, or there's also some indication of whether non-null was "on" in the assembly.

Generics

If a type parameter `T` has non-nullable constraints, it is treated as non-nullable within its scope.

If a type parameter is unconstrained or has only nullable constraints, the situation is a little more complex: this means that the corresponding type argument could be *either* nullable or non-nullable. The safe thing to do in that situation is to treat the type parameter as *both* nullable and non-nullable, giving warnings when either is violated.

It is worth considering whether explicit nullable reference constraints should be allowed. Note, however, that we cannot avoid having nullable reference types *implicitly* be constraints in certain cases (inherited constraints).

The `class` constraint is non-null. We can consider whether `class?` should be a valid nullable constraint denoting "nullable reference type".

Type inference

In type inference, if a contributing type is a nullable reference type, the resulting type should be nullable. In other words, nullness is propagated.

We should consider whether the `null` literal as a participating expression should contribute nullness. It doesn't today: for value types it leads to an error, whereas for reference types the null successfully converts to the plain type.

```
string? n = "world";
var x = b ? "Hello" : n; // string?
var y = b ? "Hello" : null; // string? or error
var z = b ? 7 : null; // Error today, could be int?
```

Null guard guidance

As a feature, nullable reference types allow developers to express their intent, and provide warnings through flow analysis if that intent is contradicted. There is a common question as to whether or not null guards are necessary.

Example of null guard

```
public void DoWork(Worker worker)
{
    // Guard against worker being null
    if (worker is null)
    {
        throw new ArgumentNullException(nameof(worker));
    }

    // Otherwise use worker argument
}
```

In the previous example, the `DoWork` function accepts a `Worker` and guards against it potentially being `null`. If the `worker` argument is `null`, the `DoWork` function will `throw`. With nullable reference types, the code in the previous example makes the intent that the `Worker` parameter would *not* be `null`. If the `DoWork` function was a public API, such as a NuGet package or a shared library - as guidance you should leave null guards in place. As a public API, the only guarantee that a caller isn't passing `null` is to guard against it.

Express intent

A more compelling use of the previous example is to express that the `Worker` parameter could be `null`, thus

making the null guard more appropriate. If you remove the null guard in the following example, the compiler warns that you may be dereferencing null. Regardless, both null guards are still valid.

```
public void DoWork(Worker? worker)
{
    // Guard against worker being null
    if (worker is null)
    {
        throw new ArgumentNullException(nameof(worker));
    }

    // Otherwise use worker argument
}
```

For non-public APIs, such as source code entirely in control by a developer or dev team - the nullable reference types could allow for the safe removal of null guards where the developers can guarantee it is not necessary. The feature can help with warnings, but it cannot guarantee that at runtime code execution could result in a `NullReferenceException`.

Breaking changes

Non-null warnings are an obvious breaking change on existing code, and should be accompanied with an opt-in mechanism.

Less obviously, warnings from nullable types (as described above) are a breaking change on existing code in certain scenarios where the nullability is implicit:

- Unconstrained type parameters will be treated as implicitly nullable, so assigning them to `object` or accessing e.g. `ToString` will yield warnings.
- if type inference infers nullness from `null` expressions, then existing code will sometimes yield nullable rather than non-nullable types, which can lead to new warnings.

So nullable warnings also need to be optional

Finally, adding annotations to an existing API will be a breaking change to users who have opted in to warnings, when they upgrade the library. This, too, merits the ability to opt in or out. "I want the bug fixes, but I am not ready to deal with their new annotations"

In summary, you need to be able to opt in/out of:

- Nullable warnings
- Non-null warnings
- Warnings from annotations in other files

The granularity of the opt-in suggests an analyzer-like model, where swaths of code can opt in and out with pragmas and severity levels can be chosen by the user. Additionally, per-library options ("ignore the annotations from JSON.NET until I'm ready to deal with the fall out") may be expressible in code as attributes.

The design of the opt-in/transition experience is crucial to the success and usefulness of this feature. We need to make sure that:

- Users can adopt nullability checking gradually as they want to
- Library authors can add nullability annotations without fear of breaking customers
- Despite these, there is not a sense of "configuration nightmare"

Tweaks

We could consider not using the `?` annotations on locals, but just observing whether they are used in accordance with what gets assigned to them. I don't favor this; I think we should uniformly let people express their intent.

We could consider a shorthand `T! x` on parameters, that auto-generates a runtime null check.

Certain patterns on generic types, such as `FirstOrDefault` or `TryGet`, have slightly weird behavior with non-nullable type arguments, because they explicitly yield default values in certain situations. We could try to nuance the type system to accommodate these better. For instance, we could allow `?` on unconstrained type parameters, even though the type argument could already be nullable. I doubt that it is worth it, and it leads to weirdness related to interaction with nullable *value* types.

Nullable value types

We could consider adopting some of the above semantics for nullable value types as well.

We already mentioned type inference, where we could infer `int?` from `(7, null)`, instead of just giving an error.

Another opportunity is to apply the flow analysis to nullable value types. When they are deemed non-null, we could actually allow using as the non-nullable type in certain ways (e.g. member access). We just have to be careful that the things that you can *already* do on a nullable value type will be preferred, for back compat reasons.

Recursive Pattern Matching

18.12.2021 • 12 minutes to read

Summary

Pattern matching extensions for C# enable many of the benefits of algebraic data types and pattern matching from functional languages, but in a way that smoothly integrates with the feel of the underlying language. Elements of this approach are inspired by related features in the programming languages F# and Scala.

Detailed design

Is Expression

The `is` operator is extended to test an expression against a *pattern*.

```
relational_expression
    : is_pattern_expression
    ;
is_pattern_expression
    : relational_expression 'is' pattern
    ;
```

This form of *relational_expression* is in addition to the existing forms in the C# specification. It is a compile-time error if the *relational_expression* to the left of the `is` token does not designate a value or does not have a type.

Every *identifier* of the pattern introduces a new local variable that is *definitely assigned* after the `is` operator is `true` (i.e. *definitely assigned when true*).

> Note: There is technically an ambiguity between *type* in an `is-expression` and *constant_pattern*, either of which might be a valid parse of a qualified identifier. We try to bind it as a type for compatibility with previous versions of the language; only if that fails do we resolve it as we do an expression in other contexts, to the first thing found (which must be either a constant or a type). This ambiguity is only present on the right-hand-side of an `is` expression.

Patterns

Patterns are used in the *is_pattern* operator, in a *switch_statement*, and in a *switch_expression* to express the shape of data against which incoming data (which we call the input value) is to be compared. Patterns may be recursive so that parts of the data may be matched against sub-patterns.

```
pattern
    : declaration_pattern
    | constant_pattern
    | var_pattern
    | positional_pattern
    | property_pattern
    | discard_pattern
    ;
declaration_pattern
    : type simple_designation
    ;
constant_pattern
    : constant_expression
    ;
var_pattern
    : 'var' designation
    ;
positional_pattern
    : type? '(' subpatterns? ')' property_subpattern? simple_designation?
    ;
subpatterns
    : subpattern
    | subpattern ',' subpatterns
    ;
subpattern
    : pattern
    | identifier ':' pattern
    ;
property_subpattern
    : '{' '}'
    | '{' subpatterns ','? '}'
    ;
property_pattern
    : type? property_subpattern simple_designation?
    ;
simple_designation
    : single_variable_designation
    | discard_designation
    ;
discard_pattern
    : '_'
    ;
```

Declaration Pattern

```
declaration_pattern
    : type simple_designation
    ;
```

The *declaration_pattern* both tests that an expression is of a given type and casts it to that type if the test succeeds. This may introduce a local variable of the given type named by the given identifier, if the designation is a *single_variable_designation*. That local variable is *definitely assigned* when the result of the pattern-matching operation is `true`.

The runtime semantic of this expression is that it tests the runtime type of the left-hand *relational_expression* operand against the *type* in the pattern. If it is of that runtime type (or some subtype) and not `null`, the result of the `is operator` is `true`.

Certain combinations of static type of the left-hand-side and the given type are considered incompatible and result in compile-time error. A value of static type `E` is said to be *pattern-compatible* with a type `T` if there exists an identity conversion, an implicit reference conversion, a boxing conversion, an explicit reference conversion, or an unboxing conversion from `E` to `T`, or if one of those types is an open type. It is a compile-

time error if an input of type `E` is not *pattern-compatible* with the *type* in a type pattern that it is matched with.

The type pattern is useful for performing run-time type tests of reference types, and replaces the idiom

```
var v = expr as Type;
if (v != null) { // code using v
```

With the slightly more concise

```
if (expr is Type v) { // code using v
```

It is an error if *type* is a nullable value type.

The type pattern can be used to test values of nullable types: a value of type `Nullable<T>` (or a boxed `T`) matches a type pattern `T2 id` if the value is non-null and the type of `T2` is `T`, or some base type or interface of `T`. For example, in the code fragment

```
int? x = 3;
if (x is int v) { // code using v
```

The condition of the `if` statement is `true` at runtime and the variable `v` holds the value `3` of type `int` inside the block. After the block the variable `v` is in scope but not definitely assigned.

Constant Pattern

```
constant_pattern
    : constant_expression
    ;
```

A constant pattern tests the value of an expression against a constant value. The constant may be any constant expression, such as a literal, the name of a declared `const` variable, or an enumeration constant. When the input value is not an open type, the constant expression is implicitly converted to the type of the matched expression; if the type of the input value is not *pattern-compatible* with the type of the constant expression, the pattern-matching operation is an error.

The pattern *c* is considered matching the converted input value *e* if `object.Equals(c, e)` would return `true`.

We expect to see `e is null` as the most common way to test for `null` in newly written code, as it cannot invoke a user-defined `operator==`.

Var Pattern

```
var_pattern
    : 'var' designation
    ;
designation
    : simple_designation
    | tuple_designation
    ;
simple_designation
    : single_variable_designation
    | discard_designation
    ;
single_variable_designation
    : identifier
    ;
discard_designation
    : _
    ;
tuple_designation
    : '(' designations? ')'
    ;
designations
    : designation
    | designations ',' designation
    ;
```

If the *designation* is a *simple_designation*, an expression *e* matches the pattern. In other words, a match to a *var pattern* always succeeds with a *simple_designation*. If the *simple_designation* is a *single_variable_designation*, the value of *e* is bounds to a newly introduced local variable. The type of the local variable is the static type of *e*.

If the *designation* is a *tuple_designation*, then the pattern is equivalent to a *positional_pattern* of the form `(var` *designation*, ... `)` where the *designation*s are those found within the *tuple_designation*. For example, the pattern `var (x, (y, z))` is equivalent to `(var x, (var y, var z))`.

It is an error if the name `var` binds to a type.

Discard Pattern

```
discard_pattern
    : '_'
    ;
```

An expression *e* matches the pattern `_` always. In other words, every expression matches the discard pattern.

A discard pattern may not be used as the pattern of an *is_pattern_expression*.

Positional Pattern

A positional pattern checks that the input value is not `null`, invokes an appropriate `Deconstruct` method, and performs further pattern matching on the resulting values. It also supports a tuple-like pattern syntax (without the type being provided) when the type of the input value is the same as the type containing `Deconstruct`, or if the type of the input value is a tuple type, or if the type of the input value is `object` or `ITuple` and the runtime type of the expression implements `ITuple`.

```
positional_pattern
    : type? '(' subpatterns? ')' property_subpattern? simple_designation?
    ;
subpatterns
    : subpattern
    | subpattern ',' subpatterns
    ;
subpattern
    : pattern
    | identifier ':' pattern
    ;
```

If the *type* is omitted, we take it to be the static type of the input value.

Given a match of an input value to the pattern *type* `(` *subpattern_list* `)`, a method is selected by searching in *type* for accessible declarations of `Deconstruct` and selecting one among them using the same rules as for the deconstruction declaration.

It is an error if a *positional_pattern* omits the type, has a single *subpattern* without an *identifier*, has no *property_subpattern* and has no *simple_designation*. This disambiguates between a *constant_pattern* that is parenthesized and a *positional_pattern*.

In order to extract the values to match against the patterns in the list,

- If *type* was omitted and the input value's type is a tuple type, then the number of subpatterns is required to be the same as the cardinality of the tuple. Each tuple element is matched against the corresponding *subpattern*, and the match succeeds if all of these succeed. If any *subpattern* has an *identifier*, then that must name a tuple element at the corresponding position in the tuple type.
- Otherwise, if a suitable `Deconstruct` exists as a member of *type*, it is a compile-time error if the type of the input value is not *pattern-compatible* with *type*. At runtime the input value is tested against *type*. If this fails then the positional pattern match fails. If it succeeds, the input value is converted to this type and `Deconstruct` is invoked with fresh compiler-generated variables to receive the `out` parameters. Each value that was received is matched against the corresponding *subpattern*, and the match succeeds if all of these succeed. If any *subpattern* has an *identifier*, then that must name a parameter at the corresponding position of `Deconstruct`.
- Otherwise if *type* was omitted, and the input value is of type `object` or `ITuple` or some type that can be converted to `ITuple` by an implicit reference conversion, and no *identifier* appears among the subpatterns, then we match using `ITuple`.
- Otherwise the pattern is a compile-time error.

The order in which subpatterns are matched at runtime is unspecified, and a failed match may not attempt to match all subpatterns.

Example

This example uses many of the features described in this specification

```
    var newState = (GetState(), action, hasKey) switch {
        (DoorState.Closed, Action.Open, _) => DoorState.Opened,
        (DoorState.Opened, Action.Close, _) => DoorState.Closed,
        (DoorState.Closed, Action.Lock, true) => DoorState.Locked,
        (DoorState.Locked, Action.Unlock, true) => DoorState.Closed,
        (var state, _, _) => state };
```

Property Pattern

A property pattern checks that the input value is not `null` and recursively matches values extracted by the use of accessible properties or fields.

```
property_pattern
    : type? property_subpattern simple_designation?
    ;
property_subpattern
    : '{' '}'
    | '{' subpatterns ','? '}'
    ;
```

It is an error if any *subpattern* of a *property_pattern* does not contain an *identifier* (it must be of the second form, which has an *identifier*). A trailing comma after the last subpattern is optional.

Note that a null-checking pattern falls out of a trivial property pattern. To check if the string `s` is non-null, you can write any of the following forms

```
if (s is object o) ... // o is of type object
if (s is string x) ... // x is of type string
if (s is {} x) ... // x is of type string
if (s is {}) ...
```

Given a match of an expression *e* to the pattern *type* `{` *property_pattern_list* `}`, it is a compile-time error if the expression *e* is not *pattern-compatible* with the type *T* designated by *type*. If the type is absent, we take it to be the static type of *e*. If the *identifier* is present, it declares a pattern variable of type *type*. Each of the identifiers appearing on the left-hand-side of its *property_pattern_list* must designate an accessible readable property or field of *T*. If the *simple_designation* of the *property_pattern* is present, it defines a pattern variable of type *T*.

At runtime, the expression is tested against *T*. If this fails then the property pattern match fails and the result is `false`. If it succeeds, then each *property_subpattern* field or property is read and its value matched against its corresponding pattern. The result of the whole match is `false` only if the result of any of these is `false`. The order in which subpatterns are matched is not specified, and a failed match may not match all subpatterns at runtime. If the match succeeds and the *simple_designation* of the *property_pattern* is a *single_variable_designation*, it defines a variable of type *T* that is assigned the matched value.

> Note: The property pattern can be used to pattern-match with anonymous types.

Example

```
if (o is string { Length: 5 } s)
```

Switch Expression

A *switch_expression* is added to support `switch`-like semantics for an expression context.

The C# language syntax is augmented with the following syntactic productions:

```
multiplicative_expression
    : switch_expression
    | multiplicative_expression '*' switch_expression
    | multiplicative_expression '/' switch_expression
    | multiplicative_expression '%' switch_expression
    ;
switch_expression
    : range_expression 'switch' '{' '}'
    | range_expression 'switch' '{' switch_expression_arms ','? '}'
    ;
switch_expression_arms
    : switch_expression_arm
    | switch_expression_arms ',' switch_expression_arm
    ;
switch_expression_arm
    : pattern case_guard? '=>' expression
    ;
case_guard
    : 'when' null_coalescing_expression
    ;
```

The *switch_expression* is not permitted as an *expression_statement*.

> We are looking at relaxing this in a future revision.

The type of the *switch_expression* is the *best common type* of the expressions appearing to the right of the `=>` tokens of the *switch_expression_arm*s if such a type exists and the expression in every arm of the switch expression can be implicitly converted to that type. In addition, we add a new *switch expression conversion*, which is a predefined implicit conversion from a switch expression to every type `T` for which there exists an implicit conversion from each arm's expression to `T`.

It is an error if some *switch_expression_arm*'s pattern cannot affect the result because some previous pattern and guard will always match.

A switch expression is said to be *exhaustive* if some arm of the switch expression handles every value of its input. The compiler shall produce a warning if a switch expression is not *exhaustive*.

At runtime, the result of the *switch_expression* is the value of the *expression* of the first *switch_expression_arm* for which the expression on the left-hand-side of the *switch_expression* matches the *switch_expression_arm*'s pattern, and for which the *case_guard* of the *switch_expression_arm*, if present, evaluates to `true`. If there is no such *switch_expression_arm*, the *switch_expression* throws an instance of the exception `System.Runtime.CompilerServices.SwitchExpressionException`.

Optional parens when switching on a tuple literal

In order to switch on a tuple literal using the *switch_statement*, you have to write what appear to be redundant parens

```
switch ((a, b))
{
```

To permit

```
switch (a, b)
{
```

the parentheses of the switch statement are optional when the expression being switched on is a tuple literal.

Order of evaluation in pattern-matching

Giving the compiler flexibility in reordering the operations executed during pattern-matching can permit flexibility that can be used to improve the efficiency of pattern-matching. The (unenforced) requirement would be that properties accessed in a pattern, and the Deconstruct methods, are required to be "pure" (side-effect free, idempotent, etc). That doesn't mean that we would add purity as a language concept, only that we would allow the compiler flexibility in reordering operations.

Resolution 2018-04-04 LDM: confirmed: the compiler is permitted to reorder calls to `Deconstruct`, property accesses, and invocations of methods in `ITuple`, and may assume that returned values are the same from multiple calls. The compiler should not invoke functions that cannot affect the result, and we will be very careful before making any changes to the compiler-generated order of evaluation in the future.

Some Possible Optimizations

The compilation of pattern matching can take advantage of common parts of patterns. For example, if the top-level type test of two successive patterns in a *switch_statement* is the same type, the generated code can skip the type test for the second pattern.

When some of the patterns are integers or strings, the compiler can generate the same kind of code it generates for a switch-statement in earlier versions of the language.

For more on these kinds of optimizations, see [Scott and Ramsey (2000)].

default interface methods

18.12.2021 • 27 minutes to read

Summary

Add support for *virtual extension methods* - methods in interfaces with concrete implementations. A class or struct that implements such an interface is required to have a single *most specific* implementation for the interface method, either implemented by the class or struct, or inherited from its base classes or interfaces. Virtual extension methods enable an API author to add methods to an interface in future versions without breaking source or binary compatibility with existing implementations of that interface.

These are similar to Java's "Default Methods".

(Based on the likely implementation technique) this feature requires corresponding support in the CLI/CLR. Programs that take advantage of this feature cannot run on earlier versions of the platform.

Motivation

The principal motivations for this feature are

- Default interface methods enable an API author to add methods to an interface in future versions without breaking source or binary compatibility with existing implementations of that interface.
- The feature enables C# to interoperate with APIs targeting Android (Java) and iOS (Swift), which support similar features.
- As it turns out, adding default interface implementations provides the elements of the "traits" language feature (https://en.wikipedia.org/wiki/Trait_(computer_programming)). Traits have proven to be a powerful programming technique (http://scg.unibe.ch/archive/papers/Scha03aTraits.pdf).

Detailed design

The syntax for an interface is extended to permit

- member declarations that declare constants, operators, static constructors, and nested types;
- a *body* for a method or indexer, property, or event accessor (that is, a "default" implementation);
- member declarations that declare static fields, methods, properties, indexers, and events;
- member declarations using the explicit interface implementation syntax; and
- Explicit access modifiers (the default access is `public`).

Members with bodies permit the interface to provide a "default" implementation for the method in classes and structs that do not provide an overriding implementation.

Interfaces may not contain instance state. While static fields are now permitted, instance fields are not permitted in interfaces. Instance auto-properties are not supported in interfaces, as they would implicitly declare a hidden field.

Static and private methods permit useful refactoring and organization of code used to implement the interface's public API.

A method override in an interface must use the explicit interface implementation syntax.

It is an error to declare a class type, struct type, or enum type within the scope of a type parameter that was declared with a *variance_annotation*. For example, the declaration of `C` below is an error.

```
interface IOuter<out T>
{
    class C { } // error: class declaration within the scope of variant type parameter 'T'
}
```

Concrete methods in interfaces

The simplest form of this feature is the ability to declare a *concrete method* in an interface, which is a method with a body.

```
interface IA
{
    void M() { WriteLine("IA.M"); }
}
```

A class that implements this interface need not implement its concrete method.

```
class C : IA { } // OK

IA i = new C();
i.M(); // prints "IA.M"
```

The final override for `IA.M` in class `C` is the concrete method `M` declared in `IA`. Note that a class does not inherit members from its interfaces; that is not changed by this feature:

```
new C().M(); // error: class 'C' does not contain a member 'M'
```

Within an instance member of an interface, `this` has the type of the enclosing interface.

Modifiers in interfaces

The syntax for an interface is relaxed to permit modifiers on its members. The following are permitted: `private`, `protected`, `internal`, `public`, `virtual`, `abstract`, `sealed`, `static`, `extern`, and `partial`.

> *TODO*: check what other modifiers exist.

An interface member whose declaration includes a body is a `virtual` member unless the `sealed` or `private` modifier is used. The `virtual` modifier may be used on a function member that would otherwise be implicitly `virtual`. Similarly, although `abstract` is the default on interface members without bodies, that modifier may be given explicitly. A non-virtual member may be declared using the `sealed` keyword.

It is an error for a `private` or `sealed` function member of an interface to have no body. A `private` function member may not have the modifier `sealed`.

Access modifiers may be used on interface members of all kinds of members that are permitted. The access level `public` is the default but it may be given explicitly.

> *Open Issue:* We need to specify the precise meaning of the access modifiers such as `protected` and `internal`, and which declarations do and do not override them (in a derived interface) or implement them (in a class that implements the interface).

Interfaces may declare `static` members, including nested types, methods, indexers, properties, events, and static constructors. The default access level for all interface members is `public`.

Interfaces may not declare instance constructors, destructors, or fields.

> ***Closed Issue:*** Should operator declarations be permitted in an interface? Probably not conversion operators, but what about others? ***Decision***: Operators are permitted *except* for conversion, equality, and inequality operators.

> ***Closed Issue:*** Should `new` be permitted on interface member declarations that hide members from base interfaces? ***Decision***: Yes.

> ***Closed Issue:*** We do not currently permit `partial` on an interface or its members. That would require a separate proposal. ***Decision***: Yes. https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#permit-partial-in-interface

Overrides in interfaces

Override declarations (i.e. those containing the `override` modifier) allow the programmer to provide a most specific implementation of a virtual member in an interface where the compiler or runtime would not otherwise find one. It also allows turning an abstract member from a super-interface into a default member in a derived interface. An override declaration is permitted to *explicitly* override a particular base interface method by qualifying the declaration with the interface name (no access modifier is permitted in this case). Implicit overrides are not permitted.

```
interface IA
{
    void M() { WriteLine("IA.M"); }
}
interface IB : IA
{
    override void IA.M() { WriteLine("IB.M"); } // explicitly named
}
interface IC : IA
{
    override void M() { WriteLine("IC.M"); } // implicitly named
}
```

Override declarations in interfaces may not be declared `sealed`.

Public `virtual` function members in an interface may be overridden in a derived interface explicitly (by qualifying the name in the override declaration with the interface type that originally declared the method, and omitting an access modifier).

`virtual` function members in an interface may only be overridden explicitly (not implicitly) in derived interfaces, and members that are not `public` may only be implemented in a class or struct explicitly (not implicitly). In either case, the overridden or implemented member must be *accessible* where it is overridden.

Reabstraction

A virtual (concrete) method declared in an interface may be overridden to be abstract in a derived interface

```
interface IA
{
    void M() { WriteLine("IA.M"); }
}
interface IB : IA
{
    abstract void IA.M();
}
class C : IB { } // error: class 'C' does not implement 'IA.M'.
```

The `abstract` modifier is not required in the declaration of `IB.M` (that is the default in interfaces), but it is probably good practice to be explicit in an override declaration.

This is useful in derived interfaces where the default implementation of a method is inappropriate and a more appropriate implementation should be provided by implementing classes.

> *Open Issue:* Should reabstraction be permitted?

The most specific override rule

We require that every interface and class have a *most specific override* for every virtual member among the overrides appearing in the type or its direct and indirect interfaces. The *most specific override* is a unique override that is more specific than every other override. If there is no override, the member itself is considered the most specific override.

One override `M1` is considered *more specific* than another override `M2` if `M1` is declared on type `T1`, `M2` is declared on type `T2`, and either

1. `T1` contains `T2` among its direct or indirect interfaces, or
2. `T2` is an interface type but `T1` is not an interface type.

For example:

```
interface IA
{
    void M() { WriteLine("IA.M"); }
}
interface IB : IA
{
    void IA.M() { WriteLine("IB.M"); }
}
interface IC : IA
{
    void IA.M() { WriteLine("IC.M"); }
}
interface ID : IB, IC { } // error: no most specific override for 'IA.M'
abstract class C : IB, IC { } // error: no most specific override for 'IA.M'
abstract class D : IA, IB, IC // ok
{
    public abstract void M();
}
```

The most specific override rule ensures that a conflict (i.e. an ambiguity arising from diamond inheritance) is resolved explicitly by the programmer at the point where the conflict arises.

Because we support explicit abstract overrides in interfaces, we could do so in classes as well

```
abstract class E : IA, IB, IC // ok
{
    abstract void IA.M();
}
```

> *Open issue*: should we support explicit interface abstract overrides in classes?

In addition, it is an error if in a class declaration the most specific override of some interface method is an abstract override that was declared in an interface. This is an existing rule restated using the new terminology.

```
interface IF
{
    void M();
}
abstract class F : IF { } // error: 'F' does not implement 'IF.M'
```

It is possible for a virtual property declared in an interface to have a most specific override for its `get` accessor in one interface and a most specific override for its `set` accessor in a different interface. This is considered a violation of the *most specific override* rule.

`static` and `private` methods

Because interfaces may now contain executable code, it is useful to abstract common code into private and static methods. We now permit these in interfaces.

> *Closed issue*: Should we support private methods? Should we support static methods? **Decision: YES**

> *Open issue*: should we permit interface methods to be `protected` or `internal` or other access? If so, what are the semantics? Are they `virtual` by default? If so, is there a way to make them non-virtual?

> *Open issue*: If we support static methods, should we support (static) operators?

Base interface invocations

Code in a type that derives from an interface with a default method can explicitly invoke that interface's "base" implementation.

```
interface I0
{
   void M() { Console.WriteLine("I0"); }
}
interface I1 : I0
{
   override void M() { Console.WriteLine("I1"); }
}
interface I2 : I0
{
   override void M() { Console.WriteLine("I2"); }
}
interface I3 : I1, I2
{
   // an explicit override that invoke's a base interface's default method
   void I0.M() { I2.base.M(); }
}
```

An instance (nonstatic) method is permitted to invoke the implementation of an accessible instance method in a direct base interface nonvirtually by naming it using the syntax `base(Type).M`. This is useful when an override that is required to be provided due to diamond inheritance is resolved by delegating to one particular base implementation.

```
interface IA
{
    void M() { WriteLine("IA.M"); }
}
interface IB : IA
{
    override void IA.M() { WriteLine("IB.M"); }
}
interface IC : IA
{
    override void IA.M() { WriteLine("IC.M"); }
}

class D : IA, IB, IC
{
    void IA.M() { base(IB).M(); }
}
```

When a `virtual` or `abstract` member is accessed using the syntax `base(Type).M`, it is required that `Type` contains a unique *most specific override* for `M`.

Binding base clauses

Interfaces now contain types. These types may be used in the base clause as base interfaces. When binding a base clause, we may need to know the set of base interfaces to bind those types (e.g. to lookup in them and to resolve protected access). The meaning of an interface's base clause is thus circularly defined. To break the cycle, we add a new language rules corresponding to a similar rule already in place for classes.

While determining the meaning of the *interface_base* of an interface, the base interfaces are temporarily assumed to be empty. Intuitively this ensures that the meaning of a base clause cannot recursively depend on itself.

We used to have the following rules:

"When a class B derives from a class A, it is a compile-time error for A to depend on B. A class **directly depends on** its direct base class (if any) and **directly depends on** the ~~**class**~~ within which it is immediately nested (if any). Given this definition, the complete set of ~~**classes**~~ upon which a class depends is the reflexive and transitive closure of the **directly depends on** relationship."

It is a compile-time error for an interface to directly or indirectly inherit from itself. The **base interfaces** of an interface are the explicit base interfaces and their base interfaces. In other words, the set of base interfaces is the complete transitive closure of the explicit base interfaces, their explicit base interfaces, and so on.

We are adjusting them as follows:

When a class B derives from a class A, it is a compile-time error for A to depend on B. A class **directly depends on** its direct base class (if any) and **directly depends on** the ***type*** within which it is immediately nested (if any).

When an interface IB extends an interface IA, it is a compile-time error for IA to depend on IB. An interface **directly depends on** its direct base interfaces (if any) and **directly depends on** the type within which it is immediately nested (if any).

Given these definitions, the complete set of **types** upon which a type depends is the reflexive and transitive closure of the **directly depends on** relationship.

Effect on existing programs

The rules presented here are intended to have no effect on the meaning of existing programs.

Example 1:

```
interface IA
{
    void M();
}
class C: IA // Error: IA.M has no concrete most specific override in C
{
    public static void M() { } // method unrelated to 'IA.M' because static
}
```

Example 2:

```
interface IA
{
    void M();
}
class Base: IA
{
    void IA.M() { }
}
class Derived: Base, IA // OK, all interface members have a concrete most specific override
{
    private void M() { } // method unrelated to 'IA.M' because private
}
```

The same rules give similar results to the analogous situation involving default interface methods:

```
interface IA
{
    void M() { }
}
class Derived: IA // OK, all interface members have a concrete most specific override
{
    private void M() { } // method unrelated to 'IA.M' because private
}
```

> *Closed issue*: confirm that this is an intended consequence of the specification. **Decision: YES**

Runtime method resolution

> *Closed Issue:* The spec should describe the runtime method resolution algorithm in the face of interface default methods. We need to ensure that the semantics are consistent with the language semantics, e.g. which declared methods do and do not override or implement an `internal` method.

CLR support API

In order for compilers to detect when they are compiling for a runtime that supports this feature, libraries for such runtimes are modified to advertise that fact through the API discussed in https://github.com/dotnet/corefx/issues/17116. We add

```
namespace System.Runtime.CompilerServices
{
    public static class RuntimeFeature
    {
        // Presence of the field indicates runtime support
        public const string DefaultInterfaceImplementation = nameof(DefaultInterfaceImplementation);
    }
}
```

> *Open issue*: Is that the best name for the *CLR* feature? The CLR feature does much more than just that (e.g. relaxes protection constraints, supports overrides in interfaces, etc). Perhaps it should be called something like "concrete methods in interfaces", or "traits"?

Further areas to be specified

- [ ] It would be useful to catalog the kinds of source and binary compatibility effects caused by adding default interface methods and overrides to existing interfaces.

Drawbacks

This proposal requires a coordinated update to the CLR specification (to support concrete methods in interfaces and method resolution). It is therefore fairly "expensive" and it may be worth doing in combination with other features that we also anticipate would require CLR changes.

Alternatives

None.

Unresolved questions

- Open questions are called out throughout the proposal, above.
- See also https://github.com/dotnet/csharplang/issues/406 for a list of open questions.
- The detailed specification must describe the resolution mechanism used at runtime to select the precise method to be invoked.
- The interaction of metadata produced by new compilers and consumed by older compilers needs to be worked out in detail. For example, we need to ensure that the metadata representation that we use does not cause the addition of a default implementation in an interface to break an existing class that implements that interface when compiled by an older compiler. This may affect the metadata representation that we can use.
- The design must consider interoperation with other languages and existing compilers for other languages.

Resolved Questions

Abstract Override

The earlier draft spec contained the ability to "reabstract" an inherited method:

```
interface IA
{
    void M();
}
interface IB : IA
{
    override void M() { }
}
interface IC : IB
{
    override void M(); // make it abstract again
}
```

My notes for 2017-03-20 showed that we decided not to allow this. However, there are at least two use cases for it:

1. The Java APIs, with which some users of this feature hope to interoperate, depend on this facility.
2. Programming with *traits* benefits from this. Reabstraction is one of the elements of the "traits" language feature (https://en.wikipedia.org/wiki/Trait_(computer_programming)). The following is permitted with

classes:

```
public abstract class Base
{
    public abstract void M();
}
public abstract class A : Base
{
    public override void M() { }
}
public abstract class B : A
{
    public override abstract void M(); // reabstract Base.M
}
```

Unfortunately this code cannot be refactored as a set of interfaces (traits) unless this is permitted. By the *Jared principle of greed*, it should be permitted.

> ***Closed issue:*** Should reabstraction be permitted? [YES] My notes were wrong. The LDM notes say that reabstraction is permitted in an interface. Not in a class.

Virtual Modifier vs Sealed Modifier

From Aleksey Tsingauz:

> We decided to allow modifiers explicitly stated on interface members, unless there is a reason to disallow some of them. This brings an interesting question around virtual modifier. Should it be required on members with default implementation?
>
> We could say that:
>
> - if there is no implementation and neither virtual, nor sealed are specified, we assume the member is abstract.
> - if there is an implementation and neither abstract, nor sealed are specified, we assume the member is virtual.
> - sealed modifier is required to make a method neither virtual, nor abstract.
>
> Alternatively, we could say that virtual modifier is required for a virtual member. I.e, if there is a member with implementation not explicitly marked with virtual modifier, it is neither virtual, nor abstract. This approach might provide better experience when a method is moved from a class to an interface:
>
> - an abstract method stays abstract.
> - a virtual method stays virtual.
> - a method without any modifier stays neither virtual, nor abstract.
> - sealed modifier cannot be applied to a method that is not an override.
>
> What do you think?

> ***Closed Issue:*** Should a concrete method (with implementation) be implicitly `virtual`? [YES]

Decisions: Made in the LDM 2017-04-05:

1. non-virtual should be explicitly expressed through `sealed` or `private`.
2. `sealed` is the keyword to make interface instance members with bodies non-virtual
3. We want to allow all modifiers in interfaces
4. Default accessibility for interface members is public, including nested types

5. private function members in interfaces are implicitly sealed, and `sealed` is not permitted on them.
6. Private classes (in interfaces) are permitted and can be sealed, and that means sealed in the class sense of sealed.
7. Absent a good proposal, partial is still not allowed on interfaces or their members.

Binary Compatibility 1

When a library provides a default implementation

```
interface I1
{
    void M() { Impl1 }
}
interface I2 : I1
{
}
class C : I2
{
}
```

We understand that the implementation of `I1.M` in `C` is `I1.M`. What if the assembly containing `I2` is changed as follows and recompiled

```
interface I2 : I1
{
    override void M() { Impl2 }
}
```

but `C` is not recompiled. What happens when the program is run? An invocation of `(C as I1).M()`

1. Runs `I1.M`
2. Runs `I2.M`
3. Throws some kind of runtime error

Decision: Made 2017-04-11: Runs `I2.M`, which is the unambiguously most specific override at runtime.

Event accessors (closed)

> *Closed Issue:* Can an event be overridden "piecewise"?

Consider this case:

```
public interface I1
{
    event T e1;
}
public interface I2 : I1
{
    override event T
    {
        add { }
        // error: "remove" accessor missing
    }
}
```

This "partial" implementation of the event is not permitted because, as in a class, the syntax for an event declaration does not permit only one accessor; both (or neither) must be provided. You could accomplish the same thing by permitting the abstract remove accessor in the syntax to be implicitly abstract by the absence of a

body:

```
public interface I1
{
    event T e1;
}
public interface I2 : I1
{
    override event T
    {
        add { }
        remove; // implicitly abstract
    }
}
```

Note that *this is a new (proposed) syntax*. In the current grammar, event accessors have a mandatory body.

> ***Closed Issue:*** Can an event accessor be (implicitly) abstract by the omission of a body, similarly to the way that methods in interfaces and property accessors are (implicitly) abstract by the omission of a body?

Decision: (2017-04-18) No, event declarations require both concrete accessors (or neither).

Reabstraction in a Class (closed)

Closed Issue: We should confirm that this is permitted (otherwise adding a default implementation would be a breaking change):

```
interface I1
{
    void M() { }
}
abstract class C : I1
{
    public abstract void M(); // implement I1.M with an abstract method in C
}
```

Decision: (2017-04-18) Yes, adding a body to an interface member declaration shouldn't break C.

Sealed Override (closed)

The previous question implicitly assumes that the `sealed` modifier can be applied to an `override` in an interface. This contradicts the draft specification. Do we want to permit sealing an override? Source and binary compatibility effects of sealing should be considered.

> ***Closed Issue:*** Should we permit sealing an override?

Decision: (2017-04-18) Let's not allowed `sealed` on overrides in interfaces. The only use of `sealed` on interface members is to make them non-virtual in their initial declaration.

Diamond inheritance and classes (closed)

The draft of the proposal prefers class overrides to interface overrides in diamond inheritance scenarios:

> We require that every interface and class have a *most specific override* for every interface method among the overrides appearing in the type or its direct and indirect interfaces. The *most specific override* is a unique override that is more specific than every other override. If there is no override, the method itself is considered the most specific override.
>
> One override `M1` is considered *more specific* than another override `M2` if `M1` is declared on type `T1`, `M2` is declared on type `T2`, and either

1. `T1` contains `T2` among its direct or indirect interfaces, or
2. `T2` is an interface type but `T1` is not an interface type.

The scenario is this

```
interface IA
{
    void M();
}
interface IB : IA
{
    override void M() { WriteLine("IB"); }
}
class Base : IA
{
    void IA.M() { WriteLine("Base"); }
}
class Derived : Base, IB // allowed?
{
    static void Main()
    {
        Ia a = new Derived();
        a.M();           // what does it do?
    }
}
```

We should confirm this behavior (or decide otherwise)

> ***Closed Issue:*** Confirm the draft spec, above, for *most specific override* as it applies to mixed classes and interfaces (a class takes priority over an interface). See https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-19.md#diamonds-with-classes.

Interface methods vs structs (closed)

There are some unfortunate interactions between default interface methods and structs.

```
interface IA
{
    public void M() { }
}
struct S : IA
{
}
```

Note that interface members are not inherited:

```
var s = default(S);
s.M(); // error: 'S' does not contain a member 'M'
```

Consequently, the client must box the struct to invoke interface methods

```
IA s = default(S); // an S, boxed
s.M(); // ok
```

Boxing in this way defeats the principal benefits of a `struct` type. Moreover, any mutation methods will have no apparent effect, because they are operating on a *boxed copy* of the struct:

```
interface IB
{
    public void Increment() { P += 1; }
    public int P { get; set; }
}
struct T : IB
{
    public int P { get; set; } // auto-property
}

T t = default(T);
Console.WriteLine(t.P); // prints 0
(t as IB).Increment();
Console.WriteLine(t.P); // prints 0
```

> *Closed Issue:* What can we do about this:
>
> 1. Forbid a `struct` from inheriting a default implementation. All interface methods would be treated as abstract in a `struct`. Then we may take time later to decide how to make it work better.
> 2. Come up with some kind of code generation strategy that avoids boxing. Inside a method like `IB.Increment`, the type of `this` would perhaps be akin to a type parameter constrained to `IB`. In conjunction with that, to avoid boxing in the caller, non-abstract methods would be inherited from interfaces. This may increase compiler and CLR implementation work substantially.
> 3. Not worry about it and just leave it as a wart.
> 4. Other ideas?

Decision: Not worry about it and just leave it as a wart. See https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-19.md#structs-and-default-implementations.

Base interface invocations (closed)

The draft spec suggests a syntax for base interface invocations inspired by Java: `Interface.base.M()`. We need to select a syntax, at least for the initial prototype. My favorite is `base<Interface>.M()`.

> *Closed Issue:* What is the syntax for a base member invocation?

Decision: The syntax is `base(Interface).M()`. See https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-19.md#base-invocation. The interface so named must be a base interface, but does not need to be a direct base interface.

> *Open Issue:* Should base interface invocations be permitted in class members?

Decision: Yes. https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-19.md#base-invocation

Overriding non-public interface members (closed)

In an interface, non-public members from base interfaces are overridden using the `override` modifier. If it is an "explicit" override that names the interface containing the member, the access modifier is omitted.

> *Closed Issue:* If it is an "implicit" override that does not name the interface, does the access modifier have to match?

Decision: Only public members may be implicitly overridden, and the access must match. See https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-18.md#dim-implementing-

a-non-public-interface-member-not-in-list.

> ***Open Issue:*** Is the access modifier required, optional, or omitted on an explicit override such as `override void IB.M() {}`?

> ***Open Issue:*** Is `override` required, optional, or omitted on an explicit override such as `void IB.M() {}`?

How does one implement a non-public interface member in a class? Perhaps it must be done explicitly?

```
interface IA
{
    internal void MI();
    protected void MP();
}
class C : IA
{
    // are these implementations?
    internal void MI() {}
    protected void MP() {}
}
```

> ***Closed Issue:*** How does one implement a non-public interface member in a class?

Decision: You can only implement non-public interface members explicitly. See https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-18.md#dim-implementing-a-non-public-interface-member-not-in-list.

Decision: No `override` keyword permitted on interface members. https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#does-an-override-in-an-interface-introduce-a-new-member

Binary Compatibility 2 (closed)

Consider the following code in which each type is in a separate assembly

```
interface I1
{
    void M() { Impl1 }
}
interface I2 : I1
{
    override void M() { Impl2 }
}
interface I3 : I1
{
}
class C : I2, I3
{
}
```

We understand that the implementation of `I1.M` in `C` is `I2.M`. What if the assembly containing `I3` is changed as follows and recompiled

```
interface I3 : I1
{
    override void M() { Impl3 }
}
```

but `C` is not recompiled. What happens when the program is run? An invocation of `(C as I1).M()`

1. Runs `I1.M`
2. Runs `I2.M`
3. Runs `I3.M`
4. Either 2 or 3, deterministically
5. Throws some kind of runtime exception

Decision: Throw an exception (5). See https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#issues-in-default-interface-methods.

Permit `partial` in interface? (closed)

Given that interfaces may be used in ways analogous to the way abstract classes are used, it may be useful to declare them `partial`. This would be particularly useful in the face of generators.

> *Proposal:* Remove the language restriction that interfaces and members of interfaces may not be declared `partial`.

Decision: Yes. See https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#permit-partial-in-interface.

`Main` in an interface? (closed)

> *Open Issue:* Is a `static Main` method in an interface a candidate to be the program's entry point?

Decision: Yes. See https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#main-in-an-interface.

Confirm intent to support public non-virtual methods (closed)

Can we please confirm (or reverse) our decision to permit non-virtual public methods in an interface?

```
interface IA
{
    public sealed void M() { }
}
```

> *Semi-Closed Issue:* (2017-04-18) We think it is going to be useful, but will come back to it. This is a mental model tripping block.

Decision: Yes. https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#confirm-that-we-support-public-non-virtual-methods.

Does an `override` in an interface introduce a new member? (closed)

There are a few ways to observe whether an override declaration introduces a new member or not.

```
interface IA
{
    void M(int x) { }
}
interface IB : IA
{
    override void M(int y) { }
}
interface IC : IB
{
    static void M2()
    {
        M(y: 3); // permitted?
    }
    override void IB.M(int z) { } // permitted? What does it override?
}
```

> *Open Issue:* Does an override declaration in an interface introduce a new member? (closed)

In a class, an overriding method is "visible" in some senses. For example, the names of its parameters take precedence over the names of parameters in the overridden method. It may be possible to duplicate that behavior in interfaces, as there is always a most specific override. But do we want to duplicate that behavior?

Also, it is possible to "override" an override method? [Moot]

Decision: No `override` keyword permitted on interface members. https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#does-an-override-in-an-interface-introduce-a-new-member.

Properties with a private accessor (closed)

We say that private members are not virtual, and the combination of virtual and private is disallowed. But what about a property with a private accessor?

```
interface IA
{
    public virtual int P
    {
        get => 3;
        private set => { }
    }
}
```

Is this allowed? Is the `set` accessor here `virtual` or not? Can it be overridden where it is accessible? Does the following implicitly implement only the `get` accessor?

```
class C : IA
{
    public int P
    {
        get => 4;
        set { }
    }
}
```

Is the following presumably an error because IA.P.set isn't virtual and also because it isn't accessible?

```
class C : IA
{
    int IA.P
    {
        get => 4;
        set { }
    }
}
```

Decision: The first example looks valid, while the last does not. This is resolved analogously to how it already works in C#. https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#properties-with-a-private-accessor

Base Interface Invocations, round 2 (closed)

Our previous "resolution" to how to handle base invocations doesn't actually provide sufficient expressiveness. It turns out that in C# and the CLR, unlike Java, you need to specify both the interface containing the method declaration and the location of the implementation you want to invoke.

I propose the following syntax for base calls in interfaces. I'm not in love with it, but it illustrates what any syntax must be able to express:

```
interface I1 { void M(); }
interface I2 { void M(); }
interface I3 : I1, I2 { void I1.M() { } void I2.M() { } }
interface I4 : I1, I2 { void I1.M() { } void I2.M() { } }
interface I5 : I3, I4
{
    void I1.M()
    {
        base<I3>(I1).M(); // calls I3's implementation of I1.M
        base<I4>(I1).M(); // calls I4's implementation of I1.M
    }
    void I2.M()
    {
        base<I3>(I2).M(); // calls I3's implementation of I2.M
        base<I4>(I2).M(); // calls I4's implementation of I2.M
    }
}
```

If there is no ambiguity, you can write it more simply

```
interface I1 { void M(); }
interface I3 : I1 { void I1.M() { } }
interface I4 : I1 { void I1.M() { } }
interface I5 : I3, I4
{
    void I1.M()
    {
        base<I3>.M(); // calls I3's implementation of I1.M
        base<I4>.M(); // calls I4's implementation of I1.M
    }
}
```

Or

```
interface I1 { void M(); }
interface I2 { void M(); }
interface I3 : I1, I2 { void I1.M() { } void I2.M() { } }
interface I5 : I3
{
    void I1.M()
    {
        base(I1).M(); // calls I3's implementation of I1.M
    }
    void I2.M()
    {
        base(I2).M(); // calls I3's implementation of I2.M
    }
}
```

Or

```
interface I1 { void M(); }
interface I3 : I1 { void I1.M() { } }
interface I5 : I3
{
    void I1.M()
    {
        base.M(); // calls I3's implementation of I1.M
    }
}
```

Decision: Decided on `base(N.I1<T>).M(s)`, conceding that if we have an invocation binding there may be problem here later on. https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-11-14.md#default-interface-implementations

Warning for struct not implementing default method? (closed)

@vancem asserts that we should seriously consider producing a warning if a value type declaration fails to override some interface method, even if it would inherit an implementation of that method from an interface. Because it causes boxing and undermines constrained calls.

Decision: This seems like something more suited for an analyzer. It also seems like this warning could be noisy, since it would fire even if the default interface method is never called and no boxing will ever occur. https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#warning-for-struct-not-implementing-default-method

Interface static constructors (closed)

When are interface static constructors run? The current CLI draft proposes that it occurs when the first static method or field is accessed. If there are neither of those then it might never be run??

[2018-10-09 The CLR team proposes "Going to mirror what we do for valuetypes (cctor check on access to each instance method)"]

Decision: Static constructors are also run on entry to instance methods, if the static constructor was not `beforefieldinit`, in which case static constructors are run before access to the first static field. https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#when-are-interface-static-constructors-run

Design meetings

2017-03-08 LDM Meeting Notes 2017-03-21 LDM Meeting Notes 2017-03-23 meeting "CLR Behavior for Default Interface Methods" 2017-04-05 LDM Meeting Notes 2017-04-11 LDM Meeting Notes 2017-04-18 LDM Meeting Notes 2017-04-19 LDM Meeting Notes 2017-05-17 LDM Meeting Notes 2017-05-31 LDM Meeting

Notes 2017-06-14 LDM Meeting Notes 2018-10-17 LDM Meeting Notes 2018-11-14 LDM Meeting Notes

Async Streams

18.12.2021 • 20 minutes to read

Summary

C# has support for iterator methods and async methods, but no support for a method that is both an iterator and an async method. We should rectify this by allowing for `await` to be used in a new form of `async` iterator, one that returns an `IAsyncEnumerable<T>` or `IAsyncEnumerator<T>` rather than an `IEnumerable<T>` or `IEnumerator<T>`, with `IAsyncEnumerable<T>` consumable in a new `await foreach`. An `IAsyncDisposable` interface is also used to enable asynchronous cleanup.

Related discussion

- https://github.com/dotnet/roslyn/issues/261
- https://github.com/dotnet/roslyn/issues/114

Detailed design

Interfaces

IAsyncDisposable

There has been much discussion of `IAsyncDisposable` (e.g. https://github.com/dotnet/roslyn/issues/114) and whether it's a good idea. However, it's a required concept to add in support of async iterators. Since `finally` blocks may contain `await`s, and since `finally` blocks need to be run as part of disposing of iterators, we need async disposal. It's also just generally useful any time cleaning up of resources might take any period of time, e.g. closing files (requiring flushes), deregistering callbacks and providing a way to know when deregistration has completed, etc.

The following interface is added to the core .NET libraries (e.g. System.Private.CoreLib / System.Runtime):

```
namespace System
{
    public interface IAsyncDisposable
    {
        ValueTask DisposeAsync();
    }
}
```

As with `Dispose`, invoking `DisposeAsync` multiple times is acceptable, and subsequent invocations after the first should be treated as nops, returning a synchronously completed successful task (`DisposeAsync` need not be thread-safe, though, and need not support concurrent invocation). Further, types may implement both `IDisposable` and `IAsyncDisposable`, and if they do, it's similarly acceptable to invoke `Dispose` and then `DisposeAsync` or vice versa, but only the first should be meaningful and subsequent invocations of either should be a nop. As such, if a type does implement both, consumers are encouraged to call once and only once the more relevant method based on the context, `Dispose` in synchronous contexts and `DisposeAsync` in asynchronous ones.

(I'm leaving discussion of how `IAsyncDisposable` interacts with `using` to a separate discussion. And coverage of how it interacts with `foreach` is handled later in this proposal.)

Alternatives considered:

- `DisposeAsync` *accepting a* `CancellationToken`: while in theory it makes sense that anything async can be canceled, disposal is about cleanup, closing things out, free'ing resources, etc., which is generally not something that should be canceled; cleanup is still important for work that's canceled. The same `CancellationToken` that caused the actual work to be canceled would typically be the same token passed to `DisposeAsync`, making `DisposeAsync` worthless because cancellation of the work would cause `DisposeAsync` to be a nop. If someone wants to avoid being blocked waiting for disposal, they can avoid waiting on the resulting `ValueTask`, or wait on it only for some period of time.
- `DisposeAsync` *returning a* `Task`: Now that a non-generic `ValueTask` exists and can be constructed from an `IValueTaskSource`, returning `ValueTask` from `DisposeAsync` allows an existing object to be reused as the promise representing the eventual async completion of `DisposeAsync`, saving a `Task` allocation in the case where `DisposeAsync` completes asynchronously.
- *Configuring* `DisposeAsync` *with a* `bool continueOnCapturedContext` *(*`ConfigureAwait`*)*: While there may be issues related to how such a concept is exposed to `using`, `foreach`, and other language constructs that consume this, from an interface perspective it's not actually doing any `await`'ing and there's nothing to configure... consumers of the `ValueTask` can consume it however they wish.
- `IAsyncDisposable` *inheriting* `IDisposable`: Since only one or the other should be used, it doesn't make sense to force types to implement both.
- `IDisposableAsync` *instead of* `IAsyncDisposable`: We've been following the naming that things/types are an "async something" whereas operations are "done async", so types have "Async" as a prefix and methods have "Async" as a suffix.

IAsyncEnumerable / IAsyncEnumerator

Two interfaces are added to the core .NET libraries:

```
namespace System.Collections.Generic
{
    public interface IAsyncEnumerable<out T>
    {
        IAsyncEnumerator<T> GetAsyncEnumerator(CancellationToken cancellationToken = default);
    }

    public interface IAsyncEnumerator<out T> : IAsyncDisposable
    {
        ValueTask<bool> MoveNextAsync();
        T Current { get; }
    }
}
```

Typical consumption (without additional language features) would look like:

```
IAsyncEnumerator<T> enumerator = enumerable.GetAsyncEnumerator();
try
{
    while (await enumerator.MoveNextAsync())
    {
        Use(enumerator.Current);
    }
}
finally { await enumerator.DisposeAsync(); }
```

Discarded options considered:

- `Task<bool> MoveNextAsync(); T current { get; }`: Using `Task<bool>` would support using a cached task object to represent synchronous, successful `MoveNextAsync` calls, but an allocation would still be required for

asynchronous completion. By returning `ValueTask<bool>`, we enable the enumerator object to itself implement `IValueTaskSource<bool>` and be used as the backing for the `ValueTask<bool>` returned from `MoveNextAsync`, which in turn allows for significantly reduced overheads.

- *`ValueTask<(bool, T)> MoveNextAsync();`*: It's not only harder to consume, but it means that `T` can no longer be covariant.
- *`ValueTask<T?> TryMoveNextAsync();`*: Not covariant.
- *`Task<T?> TryMoveNextAsync();`*: Not covariant, allocations on every call, etc.
- *`ITask<T?> TryMoveNextAsync();`*: Not covariant, allocations on every call, etc.
- *`ITask<(bool,T)> TryMoveNextAsync();`*: Not covariant, allocations on every call, etc.
- *`Task<bool> TryMoveNextAsync(out T result);`*: The `out` result would need to be set when the operation returns synchronously, not when it asynchronously completes the task potentially sometime long in the future, at which point there'd be no way to communicate the result.
- *`IAsyncEnumerator<T>` not implementing `IAsyncDisposable`*: We could choose to separate these. However, doing so complicates certain other areas of the proposal, as code must then be able to deal with the possibility that an enumerator doesn't provide disposal, which makes it difficult to write pattern-based helpers. Further, it will be common for enumerators to have a need for disposal (e.g. any C# async iterator that has a finally block, most things enumerating data from a network connection, etc.), and if one doesn't, it is simple to implement the method purely as `public ValueTask DisposeAsync() => default(ValueTask);` with minimal additional overhead.
- _ `IAsyncEnumerator<T> GetAsyncEnumerator()`: No cancellation token parameter.

Viable alternative:

```
namespace System.Collections.Generic
{
    public interface IAsyncEnumerable<out T>
    {
        IAsyncEnumerator<T> GetAsyncEnumerator();
    }

    public interface IAsyncEnumerator<out T> : IAsyncDisposable
    {
        ValueTask<bool> WaitForNextAsync();
        T TryGetNext(out bool success);
    }
}
```

`TryGetNext` is used in an inner loop to consume items with a single interface call as long as they're available synchronously. When the next item can't be retrieved synchronously, it returns false, and any time it returns false, a caller must subsequently invoke `WaitForNextAsync` to either wait for the next item to be available or to determine that there will never be another item. Typical consumption (without additional language features) would look like:

```
IAsyncEnumerator<T> enumerator = enumerable.GetAsyncEnumerator();
try
{
    while (await enumerator.WaitForNextAsync())
    {
        while (true)
        {
            int item = enumerator.TryGetNext(out bool success);
            if (!success) break;
            Use(item);
        }
    }
}
finally { await enumerator.DisposeAsync(); }
```

The advantage of this is two-fold, one minor and one major:

- *Minor: Allows for an enumerator to support multiple consumers*. There may be scenarios where it's valuable for an enumerator to support multiple concurrent consumers. That can't be achieved when `MoveNextAsync` and `Current` are separate such that an implementation can't make their usage atomic. In contrast, this approach provides a single method `TryGetNext` that supports pushing the enumerator forward and getting the next item, so the enumerator can enable atomicity if desired. However, it's likely that such scenarios could also be enabled by giving each consumer its own enumerator from a shared enumerable. Further, we don't want to enforce that every enumerator support concurrent usage, as that would add non-trivial overheads to the majority case that doesn't require it, which means a consumer of the interface generally couldn't rely on this any way.
- *Major: Performance*. The `MoveNextAsync` / `Current` approach requires two interface calls per operation, whereas the best case for `WaitForNextAsync` / `TryGetNext` is that most iterations complete synchronously, enabling a tight inner loop with `TryGetNext`, such that we only have one interface call per operation. This can have a measurable impact in situations where the interface calls dominate the computation.

However, there are non-trivial downsides, including significantly increased complexity when consuming these manually, and an increased chance of introducing bugs when using them. And while the performance benefits show up in microbenchmarks, we don't believe they'll be impactful in the vast majority of real usage. If it turns out they are, we can introduce a second set of interfaces in a light-up fashion.

Discarded options considered:

- `ValueTask<bool> WaitForNextAsync(); bool TryGetNext(out T result);`: `out` parameters can't be covariant. There's also a small impact here (an issue with the try pattern in general) that this likely incurs a runtime write barrier for reference type results.

Cancellation

There are several possible approaches to supporting cancellation:

1. `IAsyncEnumerable<T>` / `IAsyncEnumerator<T>` are cancellation-agnostic: `CancellationToken` doesn't appear anywhere. Cancellation is achieved by logically baking the `CancellationToken` into the enumerable and/or enumerator in whatever manner is appropriate, e.g. when calling an iterator, passing the `CancellationToken` as an argument to the iterator method and using it in the body of the iterator, as is done with any other parameter.
2. `IAsyncEnumerator<T>.GetAsyncEnumerator(CancellationToken)`: You pass a `CancellationToken` to `GetAsyncEnumerator`, and subsequent `MoveNextAsync` operations respect it however it can.
3. `IAsyncEnumerator<T>.MoveNextAsync(CancellationToken)`: You pass a `CancellationToken` to each individual `MoveNextAsync` call.
4. 1 && 2: You both embed `CancellationToken`s into your enumerable/enumerator and pass `CancellationToken`s into `GetAsyncEnumerator`.

5. 1 && 3: You both embed `CancellationToken`s into your enumerable/enumerator and pass `CancellationToken`s into `MoveNextAsync`.

From a purely theoretical perspective, (5) is the most robust, in that (a) `MoveNextAsync` accepting a `CancellationToken` enables the most fine-grained control over what's canceled, and (b) `CancellationToken` is just any other type that can passed as an argument into iterators, embedded in arbitrary types, etc.

However, there are multiple problems with that approach:

- How does a `CancellationToken` passed to `GetAsyncEnumerator` make it into the body of the iterator? We could expose a new `iterator` keyword that you could dot off of to get access to the `CancellationToken` passed to `GetEnumerator`, but a) that's a lot of additional machinery, b) we're making it a very first-class citizen, and c) the 99% case would seem to be the same code both calling an iterator and calling `GetAsyncEnumerator` on it, in which case it can just pass the `CancellationToken` as an argument into the method.
- How does a `CancellationToken` passed to `MoveNextAsync` get into the body of the method? This is even worse, as if it's exposed off of an `iterator` local object, its value could change across awaits, which means any code that registered with the token would need to unregister from it prior to awaits and then re-register after; it's also potentially quite expensive to need to do such registering and unregistering in every `MoveNextAsync` call, regardless of whether implemented by the compiler in an iterator or by a developer manually.
- How does a developer cancel a `foreach` loop? If it's done by giving a `CancellationToken` to an enumerable/enumerator, then either a) we need to support `foreach`'ing over enumerators, which raises them to being first-class citizens, and now you need to start thinking about an ecosystem built up around enumerators (e.g. LINQ methods) or b) we need to embed the `CancellationToken` in the enumerable anyway by having some `WithCancellation` extension method off of `IAsyncEnumerable<T>` that would store the provided token and then pass it into the wrapped enumerable's `GetAsyncEnumerator` when the `GetAsyncEnumerator` on the returned struct is invoked (ignoring that token). Or, you can just use the `CancellationToken` you have in the body of the foreach.
- If/when query comprehensions are supported, how would the `CancellationToken` supplied to `GetEnumerator` or `MoveNextAsync` be passed into each clause? The easiest way would simply be for the clause to capture it, at which point whatever token is passed to `GetAsyncEnumerator`/`MoveNextAsync` is ignored.

An earlier version of this document recommended (1), but we since switched to (4).

The two main problems with (1):

- producers of cancellable enumerables have to implement some boilerplate, and can only leverage the compiler's support for async-iterators to implement a `IAsyncEnumerator<T> GetAsyncEnumerator(CancellationToken)` method.
- it is likely that many producers would be tempted to just add a `CancellationToken` parameter to their async-enumerable signature instead, which will prevent consumers from passing the cancellation token they want when they are given an `IAsyncEnumerable` type.

There are two main consumption scenarios:

1. `await foreach (var i in GetData(token)) ...` where the consumer calls the async-iterator method,
2. `await foreach (var i in givenIAsyncEnumerable.WithCancellation(token)) ...` where the consumer deals with a given `IAsyncEnumerable` instance.

We find that a reasonable compromise to support both scenarios in a way that is convenient for both producers and consumers of async-streams is to use a specially annotated parameter in the async-iterator method. The `[EnumeratorCancellation]` attribute is used for this purpose. Placing this attribute on a parameter tells the compiler that if a token is passed to the `GetAsyncEnumerator` method, that token should be used instead of the

value originally passed for the parameter.

Consider `IAsyncEnumerable<int> GetData([EnumeratorCancellation] CancellationToken token = default)`. The implementer of this method can simply use the parameter in the method body. The consumer can use either consumption patterns above:

1. if you use `GetData(token)`, then the token is saved into the async-enumerable and will be used in iteration,
2. if you use `givenIAsyncEnumerable.WithCancellation(token)`, then the token passed to `GetAsyncEnumerator` will supersede any token saved in the async-enumerable.

foreach

`foreach` will be augmented to support `IAsyncEnumerable<T>` in addition to its existing support for `IEnumerable<T>`. And it will support the equivalent of `IAsyncEnumerable<T>` as a pattern if the relevant members are exposed publicly, falling back to using the interface directly if not, in order to enable struct-based extensions that avoid allocating as well as using alternative awaitables as the return type of `MoveNextAsync` and `DisposeAsync`.

Syntax

Using the syntax:

```
foreach (var i in enumerable)
```

C# will continue to treat `enumerable` as a synchronous enumerable, such that even if it exposes the relevant APIs for async enumerables (exposing the pattern or implementing the interface), it will only consider the synchronous APIs.

To force `foreach` to instead only consider the asynchronous APIs, `await` is inserted as follows:

```
await foreach (var i in enumerable)
```

No syntax would be provided that would support using either the async or the sync APIs; the developer must choose based on the syntax used.

Discarded options considered:

- *`foreach (var i in await enumerable)`*: This is already valid syntax, and changing its meaning would be a breaking change. This means to `await` the `enumerable`, get back something synchronously iterable from it, and then synchronously iterate through that.
- *`foreach (var i await in enumerable)`*, *`foreach (var await i in enumerable)`*, *`foreach (await var i in enumerable)`*: These all suggest that we're awaiting the next item, but there are other awaits involved in foreach, in particular if the enumerable is an `IAsyncDisposable`, we will be `await`'ing its async disposal. That await is as the scope of the foreach rather than for each individual element, and thus the `await` keyword deserves to be at the `foreach` level. Further, having it associated with the `foreach` gives us a way to describe the `foreach` with a different term, e.g. a "await foreach". But more importantly, there's value in considering `foreach` syntax at the same time as `using` syntax, so that they remain consistent with each other, and `using (await ...)` is already valid syntax.
- *`foreach await (var i in enumerable)`*

Still to consider:

- `foreach` today does not support iterating through an enumerator. We expect it will be more common to have `IAsyncEnumerator<T>`s handed around, and thus it's tempting to support `await foreach` with both

`IAsyncEnumerable<T>` and `IAsyncEnumerator<T>`. But once we add such support, it introduces the question of whether `IAsyncEnumerator<T>` is a first-class citizen, and whether we need to have overloads of combinators that operate on enumerators in addition to enumerables? Do we want to encourage methods to return enumerators rather than enumerables? We should continue to discuss this. If we decide we don't want to support it, we might want to introduce an extension method `public static IAsyncEnumerable<T> AsEnumerable<T>(this IAsyncEnumerator<T> enumerator);` that would allow an enumerator to still be `foreach`'d. If we decide we do want to support it, we'll need to also decide on whether the `await foreach` would be responsible for calling `DisposeAsync` on the enumerator, and the answer is likely "no, control over disposal should be handled by whoever called `GetEnumerator`."

Pattern-based Compilation

The compiler will bind to the pattern-based APIs if they exist, preferring those over using the interface (the pattern may be satisfied with instance methods or extension methods). The requirements for the pattern are:

- The enumerable must expose a `GetAsyncEnumerator` method that may be called with no arguments and that returns an enumerator that meets the relevant pattern.
- The enumerator must expose a `MoveNextAsync` method that may be called with no arguments and that returns something which may be `await`ed and whose `GetResult()` returns a `bool`.
- The enumerator must also expose `Current` property whose getter returns a `T` representing the kind of data being enumerated.
- The enumerator may optionally expose a `DisposeAsync` method that may be invoked with no arguments and that returns something that can be `await`ed and whose `GetResult()` returns `void`.

This code:

```
var enumerable = ...;
await foreach (T item in enumerable)
{
   ...
}
```

is translated to the equivalent of:

```
var enumerable = ...;
var enumerator = enumerable.GetAsyncEnumerator();
try
{
    while (await enumerator.MoveNextAsync())
    {
       T item = enumerator.Current;
       ...
    }
}
finally
{
    await enumerator.DisposeAsync(); // omitted, along with the try/finally, if the enumerator doesn't
expose DisposeAsync
}
```

If the iterated type doesn't expose the right pattern, the interfaces will be used.

ConfigureAwait

This pattern-based compilation will allow `ConfigureAwait` to be used on all of the awaits, via a `ConfigureAwait` extension method:

```
await foreach (T item in enumerable.ConfigureAwait(false))
{
   ...
}
```

This will be based on types we'll add to .NET as well, likely to System.Threading.Tasks.Extensions.dll:

```
// Approximate implementation, omitting arg validation and the like
namespace System.Threading.Tasks
{
    public static class AsyncEnumerableExtensions
    {
        public static ConfiguredAsyncEnumerable<T> ConfigureAwait<T>(this IAsyncEnumerable<T> enumerable,
bool continueOnCapturedContext) =>
            new ConfiguredAsyncEnumerable<T>(enumerable, continueOnCapturedContext);

        public struct ConfiguredAsyncEnumerable<T>
        {
            private readonly IAsyncEnumerable<T> _enumerable;
            private readonly bool _continueOnCapturedContext;

            internal ConfiguredAsyncEnumerable(IAsyncEnumerable<T> enumerable, bool
continueOnCapturedContext)
            {
                _enumerable = enumerable;
                _continueOnCapturedContext = continueOnCapturedContext;
            }

            public ConfiguredAsyncEnumerator<T> GetAsyncEnumerator() =>
                new ConfiguredAsyncEnumerator<T>(_enumerable.GetAsyncEnumerator(),
_continueOnCapturedContext);

            public struct Enumerator
            {
                private readonly IAsyncEnumerator<T> _enumerator;
                private readonly bool _continueOnCapturedContext;

                internal Enumerator(IAsyncEnumerator<T> enumerator, bool continueOnCapturedContext)
                {
                    _enumerator = enumerator;
                    _continueOnCapturedContext = continueOnCapturedContext;
                }

                public ConfiguredValueTaskAwaitable<bool> MoveNextAsync() =>
                    _enumerator.MoveNextAsync().ConfigureAwait(_continueOnCapturedContext);

                public T Current => _enumerator.Current;

                public ConfiguredValueTaskAwaitable DisposeAsync() =>
                    _enumerator.DisposeAsync().ConfigureAwait(_continueOnCapturedContext);
            }
        }
    }
}
```

Note that this approach will not enable `ConfigureAwait` to be used with pattern-based enumerables, but then again it's already the case that the `ConfigureAwait` is only exposed as an extension on `Task` / `Task<T>` / `ValueTask` / `ValueTask<T>` and can't be applied to arbitrary awaitable things, as it only makes sense when applied to Tasks (it controls a behavior implemented in Task's continuation support), and thus doesn't make sense when using a pattern where the awaitable things may not be tasks. Anyone returning awaitable things can provide their own custom behavior in such advanced scenarios.

(If we can come up with some way to support a scope- or assembly-level `ConfigureAwait` solution, then this

won't be necessary.)

Async Iterators

The language / compiler will support producing `IAsyncEnumerable<T>`s and `IAsyncEnumerator<T>`s in addition to consuming them. Today the language supports writing an iterator like:

```
static IEnumerable<int> MyIterator()
{
    try
    {
        for (int i = 0; i < 100; i++)
        {
            Thread.Sleep(1000);
            yield return i;
        }
    }
    finally
    {
        Thread.Sleep(200);
        Console.WriteLine("finally");
    }
}
```

but `await` can't be used in the body of these iterators. We will add that support.

Syntax

The existing language support for iterators infers the iterator nature of the method based on whether it contains any `yield`s. The same will be true for async iterators. Such async iterators will be demarcated and differentiated from synchronous iterators via adding `async` to the signature, and must then also have either `IAsyncEnumerable<T>` or `IAsyncEnumerator<T>` as its return type. For example, the above example could be written as an async iterator as follows:

```
static async IAsyncEnumerable<int> MyIterator()
{
    try
    {
        for (int i = 0; i < 100; i++)
        {
            await Task.Delay(1000);
            yield return i;
        }
    }
    finally
    {
        await Task.Delay(200);
        Console.WriteLine("finally");
    }
}
```

Alternatives considered:

- *Not using `async` in the signature*: Using `async` is likely technically required by the compiler, as it uses it to determine whether `await` is valid in that context. But even if it's not required, we've established that `await` may only be used in methods marked as `async`, and it seems important to keep the consistency.
- *Enabling custom builders for `IAsyncEnumerable<T>`*: That's something we could look at for the future, but the machinery is complicated and we don't support that for the synchronous counterparts.
- *Having an `iterator` keyword in the signature*: Async iterators would use `async iterator` in the signature, and `yield` could only be used in `async` methods that included `iterator`; `iterator` would then be made

optional on synchronous iterators. Depending on your perspective, this has the benefit of making it very clear by the signature of the method whether `yield` is allowed and whether the method is actually meant to return instances of type `IAsyncEnumerable<T>` rather than the compiler manufacturing one based on whether the code uses `yield` or not. But it is different from synchronous iterators, which don't and can't be made to require one. Plus some developers don't like the extra syntax. If we were designing it from scratch, we'd probably make this required, but at this point there's much more value in keeping async iterators close to sync iterators.

LINQ

There are over ~200 overloads of methods on the `System.Linq.Enumerable` class, all of which work in terms of `IEnumerable<T>`; some of these accept `IEnumerable<T>`, some of them produce `IEnumerable<T>`, and many do both. Adding LINQ support for `IAsyncEnumerable<T>` would likely entail duplicating all of these overloads for it, for another ~200. And since `IAsyncEnumerator<T>` is likely to be more common as a standalone entity in the asynchronous world than `IEnumerator<T>` is in the synchronous world, we could potentially need another ~200 overloads that work with `IAsyncEnumerator<T>`. Plus, a large number of the overloads deal with predicates (e.g. `Where` that takes a `Func<T, bool>`), and it may be desirable to have `IAsyncEnumerable<T>`-based overloads that deal with both synchronous and asynchronous predicates (e.g. `Func<T, ValueTask<bool>>` in addition to `Func<T, bool>`). While this isn't applicable to all of the now ~400 new overloads, a rough calculation is that it'd be applicable to half, which means another ~200 overloads, for a total of ~600 new methods.

That is a staggering number of APIs, with the potential for even more when extension libraries like Interactive Extensions (Ix) are considered. But Ix already has an implementation of many of these, and there doesn't seem to be a great reason to duplicate that work; we should instead help the community improve Ix and recommend it for when developers want to use LINQ with `IAsyncEnumerable<T>`.

There is also the issue of query comprehension syntax. The pattern-based nature of query comprehensions would allow them to "just work" with some operators, e.g. if Ix provides the following methods:

```
public static IAsyncEnumerable<TResult> Select<TSource, TResult>(this IAsyncEnumerable<TSource> source,
Func<TSource, TResult> func);
public static IAsyncEnumerable<T> Where(this IAsyncEnumerable<T> source, Func<T, bool> func);
```

then this C# code will "just work":

```
IAsyncEnumerable<int> enumerable = ...;
IAsyncEnumerable<int> result = from item in enumerable
                               where item % 2 == 0
                               select item * 2;
```

However, there is no query comprehension syntax that supports using `await` in the clauses, so if Ix added, for example:

```
public static IAsyncEnumerable<TResult> Select<TSource, TResult>(this IAsyncEnumerable<TSource> source,
Func<TSource, ValueTask<TResult>> func);
```

then this would "just work":

```
IAsyncEnumerable<string> result = from url in urls
                                  where item % 2 == 0
                                  select SomeAsyncMethod(item);

async ValueTask<int> SomeAsyncMethod(int item)
{
    await Task.Yield();
    return item * 2;
}
```

but there'd be no way to write it with the `await` inline in the `select` clause. As a separate effort, we could look into adding `async { ... }` expressions to the language, at which point we could allow them to be used in query comprehensions and the above could instead be written as:

```
IAsyncEnumerable<int> result = from item in enumerable
                               where item % 2 == 0
                               select async
                               {
                                   await Task.Yield();
                                   return item * 2;
                               };
```

or to enabling `await` to be used directly in expressions, such as by supporting `async from`. However, it's unlikely a design here would impact the rest of the feature set one way or the other, and this isn't a particularly high-value thing to invest in right now, so the proposal is to do nothing additional here right now.

Integration with other asynchronous frameworks

Integration with `IObservable<T>` and other asynchronous frameworks (e.g. reactive streams) would be done at the library level rather than at the language level. For example, all of the data from an `IAsyncEnumerator<T>` can be published to an `IObserver<T>` simply by `await foreach`'ing over the enumerator and `OnNext`'ing the data to the observer, so an `AsObservable<T>` extension method is possible. Consuming an `IObservable<T>` in a `await foreach` requires buffering the data (in case another item is pushed while the previous item is still being processing), but such a push-pull adapter can easily be implemented to enable an `IObservable<T>` to be pulled from with an `IAsyncEnumerator<T>`. Etc. Rx/Ix already provide prototypes of such implementations, and libraries like https://github.com/dotnet/corefx/tree/master/src/System.Threading.Channels provide various kinds of buffering data structures. The language need not be involved at this stage.

Ranges

18.12.2021 • 12 minutes to read

Summary

This feature is about delivering two new operators that allow constructing `System.Index` and `System.Range` objects, and using them to index/slice collections at runtime.

Overview

Well-known types and members

To use the new syntactic forms for `System.Index` and `System.Range`, new well-known types and members may be necessary, depending on which syntactic forms are used.

To use the "hat" operator (`^`), the following is required

```
namespace System
{
    public readonly struct Index
    {
        public Index(int value, bool fromEnd);
    }
}
```

To use the `System.Index` type as an argument in an array element access, the following member is required:

```
int System.Index.GetOffset(int length);
```

The `..` syntax for `System.Range` will require the `System.Range` type, as well as one or more of the following members:

```
namespace System
{
    public readonly struct Range
    {
        public Range(System.Index start, System.Index end);
        public static Range StartAt(System.Index start);
        public static Range EndAt(System.Index end);
        public static Range All { get; }
    }
}
```

The `..` syntax allows for either, both, or none of its arguments to be absent. Regardless of the number of arguments, the `Range` constructor is always sufficient for using the `Range` syntax. However, if any of the other members are present and one or more of the `..` arguments are missing, the appropriate member may be substituted.

Finally, for a value of type `System.Range` to be used in an array element access expression, the following member must be present:

```
namespace System.Runtime.CompilerServices
{
    public static class RuntimeHelpers
    {
        public static T[] GetSubArray<T>(T[] array, System.Range range);
    }
}
```

System.Index

C# has no way of indexing a collection from the end, but rather most indexers use the "from start" notion, or do a "length - i" expression. We introduce a new Index expression that means "from the end". The feature will introduce a new unary prefix "hat" operator. Its single operand must be convertible to `System.Int32`. It will be lowered into the appropriate `System.Index` factory method call.

We augment the grammar for *unary_expression* with the following additional syntax form:

```
unary_expression
    : '^' unary_expression
    ;
```

We call this the *index from end* operator. The predefined *index from end* operators are as follows:

```
System.Index operator ^(int fromEnd);
```

The behavior of this operator is only defined for input values greater than or equal to zero.

Examples:

```
var array = new int[] { 1, 2, 3, 4, 5 };
var thirdItem = array[2];    // array[2]
var lastItem = array[^1];    // array[new Index(1, fromEnd: true)]
```

System.Range

C# has no syntactic way to access "ranges" or "slices" of collections. Usually users are forced to implement complex structures to filter/operate on slices of memory, or resort to LINQ methods like `list.Skip(5).Take(2)`. With the addition of `System.Span<T>` and other similar types, it becomes more important to have this kind of operation supported on a deeper level in the language/runtime, and have the interface unified.

The language will introduce a new range operator `x..y`. It is a binary infix operator that accepts two expressions. Either operand can be omitted (examples below), and they have to be convertible to `System.Index`. It will be lowered to the appropriate `System.Range` factory method call.

We replace the C# grammar rules for *multiplicative_expression* with the following (in order to introduce a new precedence level):

```
range_expression
    : unary_expression
    | range_expression? '..' range_expression?
    ;

multiplicative_expression
    : range_expression
    | multiplicative_expression '*' range_expression
    | multiplicative_expression '/' range_expression
    | multiplicative_expression '%' range_expression
    ;
```

All forms of the *range operator* have the same precedence. This new precedence group is lower than the *unary operators* and higher than the *multiplicative arithmetic operators*.

We call the `..` operator the *range operator*. The built-in range operator can roughly be understood to correspond to the invocation of a built-in operator of this form:

```
System.Range operator ..(Index start = 0, Index end = ^0);
```

Examples:

```
var array = new int[] { 1, 2, 3, 4, 5 };
var slice1 = array[2..^3];    // array[new Range(2, new Index(3, fromEnd: true))]
var slice2 = array[..^3];     // array[Range.EndAt(new Index(3, fromEnd: true))]
var slice3 = array[2..];      // array[Range.StartAt(2)]
var slice4 = array[..];       // array[Range.All]
```

Moreover, `System.Index` should have an implicit conversion from `System.Int32`, in order to avoid the need to overload mixing integers and indexes over multi-dimensional signatures.

Adding Index and Range support to existing library types

Implicit Index support

The language will provide an instance indexer member with a single parameter of type `Index` for types which meet the following criteria:

- The type is Countable.
- The type has an accessible instance indexer which takes a single `int` as the argument.
- The type does not have an accessible instance indexer which takes an `Index` as the first parameter. The `Index` must be the only parameter or the remaining parameters must be optional.

A type is ***Countable*** if it has a property named `Length` or `Count` with an accessible getter and a return type of `int`. The language can make use of this property to convert an expression of type `Index` into an `int` at the point of the expression without the need to use the type `Index` at all. In case both `Length` and `Count` are present, `Length` will be preferred. For simplicity going forward, the proposal will use the name `Length` to represent `Count` or `Length`.

For such types, the language will act as if there is an indexer member of the form `T this[Index index]` where `T` is the return type of the `int` based indexer including any `ref` style annotations. The new member will have the same `get` and `set` members with matching accessibility as the `int` indexer.

The new indexer will be implemented by converting the argument of type `Index` into an `int` and emitting a call to the `int` based indexer. For discussion purposes, let's use the example of `receiver[expr]`. The conversion of `expr` to `int` will occur as follows:

- When the argument is of the form `^expr2` and the type of `expr2` is `int`, it will be translated to `receiver.Length - expr2`.
- Otherwise, it will be translated as `expr.GetOffset(receiver.Length)`.

Regardless of the specific conversion strategy, the order of evaluation should be equivalent to the following:

1. `receiver` is evaluated;
2. `expr` is evaluated;
3. `length` is evaluated, if needed;
4. the `int` based indexer is invoked.

This allows for developers to use the `Index` feature on existing types without the need for modification. For example:

```
List<char> list = ...;
var value = list[^1];

// Gets translated to
var value = list[list.Count - 1];
```

The `receiver` and `Length` expressions will be spilled as appropriate to ensure any side effects are only executed once. For example:

```
class Collection {
    private int[] _array = new[] { 1, 2, 3 };

    public int Length {
        get {
            Console.Write("Length ");
            return _array.Length;
        }
    }

    public int this[int index] => _array[index];
}

class SideEffect {
    Collection Get() {
        Console.Write("Get ");
        return new Collection();
    }

    void Use() {
        int i = Get()[^1];
        Console.WriteLine(i);
    }
}
```

This code will print "Get Length 3".

Implicit Range support

The language will provide an instance indexer member with a single parameter of type `Range` for types which meet the following criteria:

- The type is Countable.
- The type has an accessible member named `Slice` which has two parameters of type `int`.
- The type does not have an instance indexer which takes a single `Range` as the first parameter. The `Range` must be the only parameter or the remaining parameters must be optional.

For such types, the language will bind as if there is an indexer member of the form `T this[Range range]` where `T` is the return type of the `Slice` method including any `ref` style annotations. The new member will also have matching accessibility with `Slice`.

When the `Range` based indexer is bound on an expression named `receiver`, it will be lowered by converting the `Range` expression into two values that are then passed to the `Slice` method. For discussion purposes, let's use the example of `receiver[expr]`.

The first argument of `Slice` will be obtained by converting the range typed expression in the following way:

- When `expr` is of the form `expr1..expr2` (where `expr2` can be omitted) and `expr1` has type `int`, then it will be emitted as `expr1`.
- When `expr` is of the form `^expr1..expr2` (where `expr2` can be omitted), then it will be emitted as `receiver.Length - expr1`.
- When `expr` is of the form `..expr2` (where `expr2` can be omitted), then it will be emitted as `0`.
- Otherwise, it will be emitted as `expr.Start.GetOffset(receiver.Length)`.

This value will be re-used in the calculation of the second `Slice` argument. When doing so it will be referred to as `start`. The second argument of `Slice` will be obtained by converting the range typed expression in the following way:

- When `expr` is of the form `expr1..expr2` (where `expr1` can be omitted) and `expr2` has type `int`, then it will be emitted as `expr2 - start`.
- When `expr` is of the form `expr1..^expr2` (where `expr1` can be omitted), then it will be emitted as `(receiver.Length - expr2) - start`.
- When `expr` is of the form `expr1..` (where `expr1` can be omitted), then it will be emitted as `receiver.Length - start`.
- Otherwise, it will be emitted as `expr.End.GetOffset(receiver.Length) - start`.

Regardless of the specific conversion strategy, the order of evaluation should be equivalent to the following:

1. `receiver` is evaluated;
2. `expr` is evaluated;
3. `length` is evaluated, if needed;
4. the `Slice` method is invoked.

The `receiver`, `expr`, and `length` expressions will be spilled as appropriate to ensure any side effects are only executed once. For example:

```
class Collection {
    private int[] _array = new[] { 1, 2, 3 };

    public int Length {
        get {
            Console.Write("Length ");
            return _array.Length;
        }
    }

    public int[] Slice(int start, int length) {
        var slice = new int[length];
        Array.Copy(_array, start, slice, 0, length);
        return slice;
    }
}

class SideEffect {
    Collection Get() {
        Console.Write("Get ");
        return new Collection();
    }

    void Use() {
        var array = Get()[0..2];
        Console.WriteLine(array.Length);
    }
}
```

This code will print "Get Length 2".

The language will special case the following known types:

- `string`: the method `Substring` will be used instead of `Slice`.
- `array`: the method `System.Runtime.CompilerServices.RuntimeHelpers.GetSubArray` will be used instead of `Slice`.

Alternatives

The new operators (`^` and `..`) are syntactic sugar. The functionality can be implemented by explicit calls to `System.Index` and `System.Range` factory methods, but it will result in a lot more boilerplate code, and the experience will be unintuitive.

IL Representation

These two operators will be lowered to regular indexer/method calls, with no change in subsequent compiler layers.

Runtime behavior

- Compiler can optimize indexers for built-in types like arrays and strings, and lower the indexing to the appropriate existing methods.
- `System.Index` will throw if constructed with a negative value.
- `^0` does not throw, but it translates to the length of the collection/enumerable it is supplied to.
- `Range.All` is semantically equivalent to `0..^0`, and can be deconstructed to these indices.

Considerations

Detect Indexable based on ICollection

The inspiration for this behavior was collection initializers. Using the structure of a type to convey that it had opted into a feature. In the case of collection initializers types can opt into the feature by implementing the interface `IEnumerable` (non generic).

This proposal initially required that types implement `ICollection` in order to qualify as Indexable. That required a number of special cases though:

- `ref struct`: these cannot implement interfaces yet types like `Span<T>` are ideal for index / range support.
- `string`: does not implement `ICollection` and adding that `interface` has a large cost.

This means to support key types special casing is already needed. The special casing of `string` is less interesting as the language does this in other areas (`foreach` lowering, constants, etc ...). The special casing of `ref struct` is more concerning as it's special casing an entire class of types. They get labeled as Indexable if they simply have a property named `Count` with a return type of `int`.

After consideration the design was normalized to say that any type which has a property `Count` / `Length` with a return type of `int` is Indexable. That removes all special casing, even for `string` and arrays.

Detect just Count

Detecting on the property names `Count` or `Length` does complicate the design a bit. Picking just one to standardize though is not sufficient as it ends up excluding a large number of types:

- Use `Length`: excludes pretty much every collection in System.Collections and sub-namespaces. Those tend to derive from `ICollection` and hence prefer `Count` over length.
- Use `Count`: excludes `string`, arrays, `Span<T>` and most `ref struct` based types

The extra complication on the initial detection of Indexable types is outweighed by its simplification in other aspects.

Choice of Slice as a name

The name `Slice` was chosen as it's the de-facto standard name for slice style operations in .NET. Starting with netcoreapp2.1 all span style types use the name `Slice` for slicing operations. Prior to netcoreapp2.1 there really aren't any examples of slicing to look to for an example. Types like `List<T>`, `ArraySegment<T>`, `SortedList<T>` would've been ideal for slicing but the concept didn't exist when types were added.

Thus, `Slice` being the sole example, it was chosen as the name.

Index target type conversion

Another way to view the `Index` transformation in an indexer expression is as a target type conversion. Instead of binding as if there is a member of the form `return_type this[Index]`, the language instead assigns a target typed conversion to `int`.

This concept could be generalized to all member access on Countable types. Whenever an expression with type `Index` is used as an argument to an instance member invocation and the receiver is Countable then the expression will have a target type conversion to `int`. The member invocations applicable for this conversion include methods, indexers, properties, extension methods, etc ... Only constructors are excluded as they have no receiver.

The target type conversion will be implemented as follows for any expression which has a type of `Index`. For discussion purposes lets use the example of `receiver[expr]`:

- When `expr` is of the form `^expr2` and the type of `expr2` is `int`, it will be translated to `receiver.Length - expr2`.
- Otherwise, it will be translated as `expr.GetOffset(receiver.Length)`.

The `receiver` and `Length` expressions will be spilled as appropriate to ensure any side effects are only executed once. For example:

```
class Collection {
    private int[] _array = new[] { 1, 2, 3 };

    public int Length {
        get {
            Console.Write("Length ");
            return _array.Length;
        }
    }

    public int GetAt(int index) => _array[index];
}

class SideEffect {
    Collection Get() {
        Console.Write("Get ");
        return new Collection();
    }

    void Use() {
        int i = Get().GetAt(^1);
        Console.WriteLine(i);
    }
}
```

This code will print "Get Length 3".

This feature would be beneficial to any member which had a parameter that represented an index. For example `List<T>.InsertAt`. This also has the potential for confusion as the language can't give any guidance as to whether or not an expression is meant for indexing. All it can do is convert any `Index` expression to `int` when invoking a member on a Countable type.

Restrictions:

- This conversion is only applicable when the expression with type `Index` is directly an argument to the member. It would not apply to any nested expressions.

Decisions made during implementation

- All members in the pattern must be instance members
- If a Length method is found but it has the wrong return type, continue looking for Count
- The indexer used for the Index pattern must have exactly one int parameter
- The Slice method used for the Range pattern must have exactly two int parameters
- When looking for the pattern members, we look for original definitions, not constructed members

Design meetings

- Jan 10, 2018
- Jan 18, 2018
- Jan 22, 2018
- Dec 3, 2018
- Mar 25, 2019
- April 1st, 2019
- April 15, 2019

"pattern-based using" and "using declarations"

18.12.2021 • 4 minutes to read

Summary

The language will add two new capabilities around the `using` statement in order to make resource management simpler: `using` should recognize a disposable pattern in addition to `IDisposable` and add a `using` declaration to the language.

Motivation

The `using` statement is an effective tool for resource management today but it requires quite a bit of ceremony. Methods that have a number of resources to manage can get syntactically bogged down with a series of `using` statements. This syntax burden is enough that most coding style guidelines explicitly have an exception around braces for this scenario.

The `using` declaration removes much of the ceremony here and gets C# on par with other languages that include resource management blocks. Additionally the pattern-based `using` lets developers expand the set of types that can participate here. In many cases removing the need to create wrapper types that only exist to allow for a values use in a `using` statement.

Together these features allow developers to simplify and expand the scenarios where `using` can be applied.

Detailed Design

using declaration

The language will allow for `using` to be added to a local variable declaration. Such a declaration will have the same effect as declaring the variable in a `using` statement at the same location.

```
if (...)
{
   using FileStream f = new FileStream(@"C:\users\jaredpar\using.md");
   // statements
}

// Equivalent to
if (...)
{
   using (FileStream f = new FileStream(@"C:\users\jaredpar\using.md"))
   {
    // statements
   }
}
```

The lifetime of a `using` local will extend to the end of the scope in which it is declared. The `using` locals will then be disposed in the reverse order in which they are declared.

```
{
    using var f1 = new FileStream("...");
    using var f2 = new FileStream("..."), f3 = new FileStream("...");
    ...
    // Dispose f3
    // Dispose f2
    // Dispose f1
}
```

There are no restrictions around `goto`, or any other control flow construct in the face of a `using` declaration. Instead the code acts just as it would for the equivalent `using` statement:

```
{
    using var f1 = new FileStream("...");
  target:
    using var f2 = new FileStream("...");
    if (someCondition)
    {
        // Causes f2 to be disposed but has no effect on f1
        goto target;
    }
}
```

A local declared in a `using` local declaration will be implicitly read-only. This matches the behavior of locals declared in a `using` statement.

The language grammar for `using` declarations will be the following:

```
local-using-declaration:
  using type using-declarators

using-declarators:
  using-declarator
  using-declarators , using-declarator

using-declarator:
  identifier = expression
```

Restrictions around `using` declaration:

- May not appear directly inside a `case` label but instead must be within a block inside the `case` label.
- May not appear as part of an `out` variable declaration.
- Must have an initializer for each declarator.
- The local type must be implicitly convertible to `IDisposable` or fulfill the `using` pattern.

pattern-based using

The language will add the notion of a disposable pattern: that is a type which has an accessible `Dispose` instance method. Types which fit the disposable pattern can participate in a `using` statement or declaration without being required to implement `IDisposable`.

```
class Resource
{
    public void Dispose() { ... }
}

using (var r = new Resource())
{
    // statements
}
```

This will allow developers to leverage `using` in a number of new scenarios:

- `ref struct`: These types can't implement interfaces today and hence can't participate in `using` statements.
- Extension methods will allow developers to augment types in other assemblies to participate in `using` statements.

In the situation where a type can be implicitly converted to `IDisposable` and also fits the disposable pattern, then `IDisposable` will be preferred. While this takes the opposite approach of `foreach` (pattern preferred over interface) it is necessary for backwards compatibility.

The same restrictions from a traditional `using` statement apply here as well: local variables declared in the `using` are read-only, a `null` value will not cause an exception to be thrown, etc ... The code generation will be different only in that there will not be a cast to `IDisposable` before calling Dispose:

```
{
   Resource r = new Resource();
   try {
      // statements
   }
   finally {
      if (r != null) r.Dispose();
   }
}
```

In order to fit the disposable pattern the `Dispose` method must be accessible, parameterless and have a `void` return type. There are no other restrictions. This explicitly means that extension methods can be used here.

Considerations

case labels without blocks

A `using declaration` is illegal directly inside a `case` label due to complications around its actual lifetime. One potential solution is to simply give it the same lifetime as an `out var` in the same location. It was deemed the extra complexity to the feature implementation and the ease of the work around (just add a block to the `case` label) didn't justify taking this route.

Future Expansions

fixed locals

A `fixed` statement has all of the properties of `using` statements that motivated the ability to have `using` locals. Consideration should be given to extending this feature to `fixed` locals as well. The lifetime and ordering rules should apply equally well for `using` and `fixed` here.

Static local functions

18.12.2021 • 2 minutes to read

Summary

Support local functions that disallow capturing state from the enclosing scope.

Motivation

Avoid unintentionally capturing state from the enclosing context. Allow local functions to be used in scenarios where a `static` method is required.

Detailed design

A local function declared `static` cannot capture state from the enclosing scope. As a result, locals, parameters, and `this` from the enclosing scope are not available within a `static` local function.

A `static` local function cannot reference instance members from an implicit or explicit `this` or `base` reference.

A `static` local function may reference `static` members from the enclosing scope.

A `static` local function may reference `constant` definitions from the enclosing scope.

`nameof()` in a `static` local function may reference locals, parameters, or `this` or `base` from the enclosing scope.

Accessibility rules for `private` members in the enclosing scope are the same for `static` and non-`static` local functions.

A `static` local function definition is emitted as a `static` method in metadata, even if only used in a delegate.

A non-`static` local function or lambda can capture state from an enclosing `static` local function but cannot capture state outside the enclosing `static` local function.

A `static` local function cannot be invoked in an expression tree.

A call to a local function is emitted as `call` rather than `callvirt`, regardless of whether the local function is `static`.

Overload resolution of a call within a local function not affected by whether the local function is `static`.

Removing the `static` modifier from a local function in a valid program does not change the meaning of the program.

Design meetings

https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-09-10.md#static-local-functions

null coalescing assignment

18.12.2021 • 2 minutes to read

Summary

Simplifies a common coding pattern where a variable is assigned a value if it is null.

As part of this proposal, we will also loosen the type requirements on `??` to allow an expression whose type is an unconstrained type parameter to be used on the left-hand side.

Motivation

It is common to see code of the form

```
if (variable == null)
{
    variable = expression;
}
```

This proposal adds a non-overloadable binary operator to the language that performs this function.

There have been at least eight separate community requests for this feature.

Detailed design

We add a new form of assignment operator

```
assignment_operator
    : '??='
    ;
```

Which follows the existing semantic rules for compound assignment operators, except that we elide the assignment if the left-hand side is non-null. The rules for this feature are as follows.

Given `a ??= b`, where `A` is the type of `a`, `B` is the type of `b`, and `A0` is the underlying type of `A` if `A` is a nullable value type:

1. If `A` does not exist or is a non-nullable value type, a compile-time error occurs.
2. If `B` is not implicitly convertible to `A` or `A0` (if `A0` exists), a compile-time error occurs.
3. If `A0` exists and `B` is implicitly convertible to `A0`, and `B` is not dynamic, then the type of `a ??= b` is `A0`. `a ??= b` is evaluated at runtime as:

   ```
   var tmp = a.GetValueOrDefault();
   if (!a.HasValue) { tmp = b; a = tmp; }
   tmp
   ```

   Except that `a` is only evaluated once.
4. Otherwise, the type of `a ??= b` is `A`. `a ??= b` is evaluated at runtime as `a ?? (a = b)`, except that `a` is only evaluated once.

For the relaxation of the type requirements of `??`, we update the spec where it currently states that, given

`a ?? b`, where `A` is the type of `a`:

> 1. If A exists and is not a nullable type or a reference type, a compile-time error occurs.

We relax this requirement to:

1. If A exists and is a non-nullable value type, a compile-time error occurs.

This allows the null coalescing operator to work on unconstrained type parameters, as the unconstrained type parameter T exists, is not a nullable type, and is not a reference type.

Drawbacks

As with any language feature, we must question whether the additional complexity to the language is repaid in the additional clarity offered to the body of C# programs that would benefit from the feature.

Alternatives

The programmer can write `(x = x ?? y)`, `if (x == null) x = y;`, or `x ?? (x = y)` by hand.

Unresolved questions

- [ ] Requires LDM review
- [ ] Should we also support `&&=` and `||=` operators?

Design meetings

None.

Readonly Instance Members

18.12.2021 • 4 minutes to read

Championed Issue: https://github.com/dotnet/csharplang/issues/1710

Summary

Provide a way to specify individual instance members on a struct do not modify state, in the same way that `readonly struct` specifies no instance members modify state.

It is worth noting that `readonly instance member` != `pure instance member`. A `pure` instance member guarantees no state will be modified. A `readonly` instance member only guarantees that instance state will not be modified.

All instance members on a `readonly struct` could be considered implicitly `readonly instance members`. Explicit `readonly instance members` declared on non-readonly structs would behave in the same manner. For example, they would still create hidden copies if you called an instance member (on the current instance or on a field of the instance) which was itself not-readonly.

Motivation

Today, users have the ability to create `readonly struct` types which the compiler enforces that all fields are readonly (and by extension, that no instance members modify the state). However, there are some scenarios where you have an existing API that exposes accessible fields or that has a mix of mutating and non-mutating members. Under these circumstances, you cannot mark the type as `readonly` (it would be a breaking change).

This normally doesn't have much impact, except in the case of `in` parameters. With `in` parameters for non-readonly structs, the compiler will make a copy of the parameter for each instance member invocation, since it cannot guarantee that the invocation does not modify internal state. This can lead to a multitude of copies and worse overall performance than if you had just passed the struct directly by value. For an example, see this code on sharplab

Some other scenarios where hidden copies can occur include `static readonly fields` and `literals`. If they are supported in the future, `blittable constants` would end up in the same boat; that is they all currently necessitate a full copy (on instance member invocation) if the struct is not marked `readonly`.

Design

Allow a user to specify that an instance member is, itself, `readonly` and does not modify the state of the instance (with all the appropriate verification done by the compiler, of course). For example:

```
public struct Vector2
{
    public float x;
    public float y;

    public readonly float GetLengthReadonly()
    {
        return MathF.Sqrt(LengthSquared);
    }

    public float GetLength()
    {
        return MathF.Sqrt(LengthSquared);
    }

    public readonly float GetLengthIllegal()
    {
        var tmp = MathF.Sqrt(LengthSquared);

        x = tmp;    // Compiler error, cannot write x
        y = tmp;    // Compiler error, cannot write y

        return tmp;
    }

    public readonly float LengthSquared
    {
        get
        {
            return (x * x) +
                   (y * y);
        }
    }
}

public static class MyClass
{
    public static float ExistingBehavior(in Vector2 vector)
    {
        // This code causes a hidden copy, the compiler effectively emits:
        //    var tmpVector = vector;
        //    return tmpVector.GetLength();
        //
        // This is done because the compiler doesn't know that `GetLength()`
        // won't mutate `vector`.

        return vector.GetLength();
    }

    public static float ReadonlyBehavior(in Vector2 vector)
    {
        // This code is emitted exactly as listed. There are no hidden
        // copies as the `readonly` modifier indicates that the method
        // won't mutate `vector`.

        return vector.GetLengthReadonly();
    }
}
```

Readonly can be applied to property accessors to indicate that `this` will not be mutated in the accessor. The following examples have readonly setters because those accessors modify the state of member field, but do not modify the value of that member field.

```
public readonly int Prop1
{
    get
    {
        return this._store["Prop1"];
    }
    set
    {
        this._store["Prop1"] = value;
    }
}
```

When `readonly` is applied to the property syntax, it means that all accessors are `readonly`.

```
public readonly int Prop2
{
    get
    {
        return this._store["Prop2"];
    }
    set
    {
        this._store["Prop2"] = value;
    }
}
```

Readonly can only be applied to accessors which do not mutate the containing type.

```
public int Prop3
{
    readonly get
    {
        return this._prop3;
    }
    set
    {
        this._prop3 = value;
    }
}
```

Readonly can be applied to some auto-implemented properties, but it won't have a meaningful effect. The compiler will treat all auto-implemented getters as readonly whether or not the `readonly` keyword is present.

```
// Allowed
public readonly int Prop4 { get; }
public int Prop5 { readonly get; }
public int Prop6 { readonly get; set; }

// Not allowed
public readonly int Prop7 { get; set; }
public int Prop8 { get; readonly set; }
```

Readonly can be applied to manually-implemented events, but not field-like events. Readonly cannot be applied to individual event accessors (add/remove).

```
// Allowed
public readonly event Action<EventArgs> Event1
{
    add { }
    remove { }
}

// Not allowed
public readonly event Action<EventArgs> Event2;
public event Action<EventArgs> Event3
{
    readonly add { }
    readonly remove { }
}
public static readonly event Event4
{
    add { }
    remove { }
}
```

Some other syntax examples:

- Expression bodied members: `public readonly float ExpressionBodiedMember => (x * x) + (y * y);`
- Generic constraints: `public readonly void GenericMethod<T>(T value) where T : struct { }`

The compiler would emit the instance member, as usual, and would additionally emit a compiler recognized attribute indicating that the instance member does not modify state. This effectively causes the hidden `this` parameter to become `in T` instead of `ref T`.

This would allow the user to safely call said instance method without the compiler needing to make a copy.

The restrictions would include:

- The `readonly` modifier cannot be applied to static methods, constructors or destructors.
- The `readonly` modifier cannot be applied to delegates.
- The `readonly` modifier cannot be applied to members of class or interface.

Drawbacks

Same drawbacks as exist with `readonly struct` methods today. Certain code may still cause hidden copies.

Notes

Using an attribute or another keyword may also be possible.

This proposal is somewhat related to (but is more a subset of) `functional purity` and/or `constant expressions`, both of which have had some existing proposals.

Permit `stackalloc` in nested contexts

18.12.2021 • 2 minutes to read

Stack allocation

We modify the section *Stack allocation* of the C# language specification to relax the places when a `stackalloc` expression may appear. We delete

```
local_variable_initializer_unsafe
    : stackalloc_initializer
    ;

stackalloc_initializer
    : 'stackalloc' unmanaged_type '[' expression ']'
    ;
```

and replace them with

```
primary_no_array_creation_expression
    : stackalloc_initializer
    ;

stackalloc_initializer
    : 'stackalloc' unmanaged_type '[' expression? ']' array_initializer?
    | 'stackalloc' '[' expression? ']' array_initializer
    ;
```

Note that the addition of an *array_initializer* to *stackalloc_initializer* (and making the index expression optional) was an extension in C# 7.3 and is not described here.

The *element type* of the `stackalloc` expression is the *unmanaged_type* named in the stackalloc expression, if any, or the common type among the elements of the *array_initializer* otherwise.

The type of the *stackalloc_initializer* with *element type* `K` depends on its syntactic context:

- If the *stackalloc_initializer* appears directly as the *local_variable_initializer* of a *local_variable_declaration* statement or a *for_initializer*, then its type is `K*`.
- Otherwise its type is `System.Span<K>`.

Stackalloc Conversion

The *stackalloc conversion* is a new built-in implicit conversion from expression. When the type of a *stackalloc_initializer* is `K*`, there is an implicit *stackalloc conversion* from the *stackalloc_initializer* to the type `System.Span<K>`.

Records

18.12.2021 • 14 minutes to read

This proposal tracks the specification for the C# 9 records feature, as agreed to by the C# language design team.

The syntax for a record is as follows:

```
record_declaration
    : attributes? class_modifier* 'partial'? 'record' identifier type_parameter_list?
      parameter_list? record_base? type_parameter_constraints_clause* record_body
    ;

record_base
    : ':' class_type argument_list?
    | ':' interface_type_list
    | ':' class_type argument_list? ',' interface_type_list
    ;

record_body
    : '{' class_member_declaration* '}' ';'?
    | ';'
    ;
```

Record types are reference types, similar to a class declaration. It is an error for a record to provide a `record_base` `argument_list` if the `record_declaration` does not contain a `parameter_list`. At most one partial type declaration of a partial record may provide a `parameter_list`.

Record parameters cannot use `ref`, `out` or `this` modifiers (but `in` and `params` are allowed).

Inheritance

Records cannot inherit from classes, unless the class is `object`, and classes cannot inherit from records. Records can inherit from other records.

Members of a record type

In addition to the members declared in the record body, a record type has additional synthesized members. Members are synthesized unless a member with a "matching" signature is declared in the record body or an accessible concrete non-virtual member with a "matching" signature is inherited. A matching member prevents the compiler from generating that member, not any other synthesized members. Two members are considered matching if they have the same signature or would be considered "hiding" in an inheritance scenario. It is an error for a member of a record to be named "Clone". It is an error for an instance field of a record to have an unsafe type.

The synthesized members are as follows:

Equality members

If the record is derived from `object`, the record type includes a synthesized readonly property equivalent to a property declared as follows:

```
Type EqualityContract { get; };
```

The property is `private` if the record type is `sealed`. Otherwise, the property is `virtual` and `protected`. The

property can be declared explicitly. It is an error if the explicit declaration does not match the expected signature or accessibility, or if the explicit declaration doesn't allow overriding it in a derived type and the record type is not `sealed`.

If the record type is derived from a base record type `Base`, the record type includes a synthesized readonly property equivalent to a property declared as follows:

```
protected override Type EqualityContract { get; };
```

The property can be declared explicitly. It is an error if the explicit declaration does not match the expected signature or accessibility, or if the explicit declaration doesn't allow overriding it in a derived type and the record type is not `sealed`. It is an error if either synthesized, or explicitly declared property doesn't override a property with this signature in the record type `Base` (for example, if the property is missing in the `Base`, or sealed, or not virtual, etc.). The synthesized property returns `typeof(R)` where `R` is the record type.

The record type implements `System.IEquatable<R>` and includes a synthesized strongly-typed overload of `Equals(R? other)` where `R` is the record type. The method is `public`, and the method is `virtual` unless the record type is `sealed`. The method can be declared explicitly. It is an error if the explicit declaration does not match the expected signature or accessibility, or the explicit declaration doesn't allow overriding it in a derived type and the record type is not `sealed`.

If `Equals(R? other)` is user-defined (not synthesized) but `GetHashCode` is not, a warning is produced.

```
public virtual bool Equals(R? other);
```

The synthesized `Equals(R?)` returns `true` if and only if each of the following are `true`:

- `other` is not `null`, and
- For each instance field `fieldN` in the record type that is not inherited, the value of `System.Collections.Generic.EqualityComparer<TN>.Default.Equals(fieldN, other.fieldN)` where `TN` is the field type, and
- If there is a base record type, the value of `base.Equals(other)` (a non-virtual call to `public virtual bool Equals(Base? other)`); otherwise the value of `EqualityContract == other.EqualityContract`.

The record type includes synthesized `==` and `!=` operators equivalent to operators declared as follows:

```
public static bool operator==(R? left, R? right)
    => (object)left == right || (left?.Equals(right) ?? false);
public static bool operator!=(R? left, R? right)
    => !(left == right);
```

The `Equals` method called by the `==` operator is the `Equals(R? other)` method specified above. The `!=` operator delegates to the `==` operator. It is an error if the operators are declared explicitly.

If the record type is derived from a base record type `Base`, the record type includes a synthesized override equivalent to a method declared as follows:

```
public sealed override bool Equals(Base? other);
```

It is an error if the override is declared explicitly. It is an error if the method doesn't override a method with same signature in record type `Base` (for example, if the method is missing in the `Base`, or sealed, or not virtual, etc.).

The synthesized override returns `Equals((object?)other)`.

The record type includes a synthesized override equivalent to a method declared as follows:

```
public override bool Equals(object? obj);
```

It is an error if the override is declared explicitly. It is an error if the method doesn't override `object.Equals(object? obj)` (for example, due to shadowing in intermediate base types, etc.). The synthesized override returns `Equals(other as R)` where `R` is the record type.

The record type includes a synthesized override equivalent to a method declared as follows:

```
public override int GetHashCode();
```

The method can be declared explicitly. It is an error if the explicit declaration doesn't allow overriding it in a derived type and the record type is not `sealed`. It is an error if either synthesized, or explicitly declared method doesn't override `object.GetHashCode()` (for example, due to shadowing in intermediate base types, etc.).

A warning is reported if one of `Equals(R?)` and `GetHashCode()` is explicitly declared but the other method is not explicit.

The synthesized override of `GetHashCode()` returns an `int` result of combining the following values:

- For each instance field `fieldN` in the record type that is not inherited, the value of `System.Collections.Generic.EqualityComparer<TN>.Default.GetHashCode(fieldN)` where `TN` is the field type, and
- If there is a base record type, the value of `base.GetHashCode()`; otherwise the value of `System.Collections.Generic.EqualityComparer<System.Type>.Default.GetHashCode(EqualityContract)`.

For example, consider the following record types:

```
record R1(T1 P1);
record R2(T1 P1, T2 P2) : R1(P1);
record R3(T1 P1, T2 P2, T3 P3) : R2(P1, P2);
```

For those record types, the synthesized equality members would be something like:

```
class R1 : IEquatable<R1>
{
    public T1 P1 { get; init; }
    protected virtual Type EqualityContract => typeof(R1);
    public override bool Equals(object? obj) => Equals(obj as R1);
    public virtual bool Equals(R1? other)
    {
        return !(other is null) &&
            EqualityContract == other.EqualityContract &&
            EqualityComparer<T1>.Default.Equals(P1, other.P1);
    }
    public static bool operator==(R1? left, R1? right)
        => (object)left == right || (left?.Equals(right) ?? false);
    public static bool operator!=(R1? left, R1? right)
        => !(left == right);
    public override int GetHashCode()
    {
        return Combine(EqualityComparer<Type>.Default.GetHashCode(EqualityContract),
            EqualityComparer<T1>.Default.GetHashCode(P1));
    }
}

class R2 : R1, IEquatable<R2>
{
    public T2 P2 { get; init; }
    protected override Type EqualityContract => typeof(R2);
    public override bool Equals(object? obj) => Equals(obj as R2);
    public sealed override bool Equals(R1? other) => Equals((object?)other);
    public virtual bool Equals(R2? other)
    {
        return base.Equals((R1?)other) &&
            EqualityComparer<T2>.Default.Equals(P2, other.P2);
    }
    public static bool operator==(R2? left, R2? right)
        => (object)left == right || (left?.Equals(right) ?? false);
    public static bool operator!=(R2? left, R2? right)
        => !(left == right);
    public override int GetHashCode()
    {
        return Combine(base.GetHashCode(),
            EqualityComparer<T2>.Default.GetHashCode(P2));
    }
}

class R3 : R2, IEquatable<R3>
{
    public T3 P3 { get; init; }
    protected override Type EqualityContract => typeof(R3);
    public override bool Equals(object? obj) => Equals(obj as R3);
    public sealed override bool Equals(R2? other) => Equals((object?)other);
    public virtual bool Equals(R3? other)
    {
        return base.Equals((R2?)other) &&
            EqualityComparer<T3>.Default.Equals(P3, other.P3);
    }
    public static bool operator==(R3? left, R3? right)
        => (object)left == right || (left?.Equals(right) ?? false);
    public static bool operator!=(R3? left, R3? right)
        => !(left == right);
    public override int GetHashCode()
    {
        return Combine(base.GetHashCode(),
            EqualityComparer<T3>.Default.GetHashCode(P3));
    }
}
```

Copy and Clone members

A record type contains two copying members:

- A constructor taking a single argument of the record type. It is referred to as a "copy constructor".
- A synthesized public parameterless instance "clone" method with a compiler-reserved name

The purpose of the copy constructor is to copy the state from the parameter to the new instance being created. This constructor doesn't run any instance field/property initializers present in the record declaration. If the constructor is not explicitly declared, a constructor will be synthesized by the compiler. If the record is sealed, the constructor will be private, otherwise it will be protected. An explicitly declared copy constructor must be either public or protected, unless the record is sealed. The first thing the constructor must do, is to call a copy constructor of the base, or a parameter-less object constructor if the record inherits from object. An error is reported if a user-defined copy constructor uses an implicit or explicit constructor initializer that doesn't fulfill this requirement. After a base copy constructor is invoked, a synthesized copy constructor copies values for all instance fields implicitly or explicitly declared within the record type. The sole presence of a copy constructor, whether explicit or implicit, doesn't prevent an automatic addition of a default instance constructor.

If a virtual "clone" method is present in the base record, the synthesized "clone" method overrides it and the return type of the method is the current containing type if the "covariant returns" feature is supported and the override return type otherwise. An error is produced if the base record clone method is sealed. If a virtual "clone" method is not present in the base record, the return type of the clone method is the containing type and the method is virtual, unless the record is sealed or abstract. If the containing record is abstract, the synthesized clone method is also abstract. If the "clone" method is not abstract, it returns the result of a call to a copy constructor.

Printing members: PrintMembers and ToString methods

If the record is derived from `object`, the record includes a synthesized method equivalent to a method declared as follows:

```
bool PrintMembers(System.Text.StringBuilder builder);
```

The method is `private` if the record type is `sealed`. Otherwise, the method is `virtual` and `protected`.

The method:

1. calls the method `System.Runtime.CompilerServices.RuntimeHelpers.EnsureSufficientExecutionStack()` if the method is present and the record has printable members.
2. for each of the record's printable members (non-static public field and readable property members), appends that member's name followed by " = " followed by the member's value separated with ", ",
3. return true if the record has printable members.

For a member that has a value type, we will convert its value to a string representation using the most efficient method available to the target platform. At present that means calling `ToString` before passing to `StringBuilder.Append`.

If the record type is derived from a base record `Base`, the record includes a synthesized override equivalent to a method declared as follows:

```
protected override bool PrintMembers(StringBuilder builder);
```

If the record has no printable members, the method calls the base `PrintMembers` method with one argument (its `builder` parameter) and returns the result.

Otherwise, the method:

1. calls the base `PrintMembers` method with one argument (its `builder` parameter),
2. if the `PrintMembers` method returned true, append ", " to the builder,
3. for each of the record's printable members, appends that member's name followed by " = " followed by the member's value: `this.member` (or `this.member.ToString()` for value types), separated with ", ",
4. return true.

The `PrintMembers` method can be declared explicitly. It is an error if the explicit declaration does not match the expected signature or accessibility, or if the explicit declaration doesn't allow overriding it in a derived type and the record type is not `sealed`.

The record includes a synthesized method equivalent to a method declared as follows:

```
public override string ToString();
```

The method can be declared explicitly. It is an error if the explicit declaration does not match the expected signature or accessibility, or if the explicit declaration doesn't allow overriding it in a derived type and the record type is not `sealed`. It is an error if either synthesized, or explicitly declared method doesn't override `object.ToString()` (for example, due to shadowing in intermediate base types, etc.).

The synthesized method:

1. creates a `StringBuilder` instance,
2. appends the record name to the builder, followed by " { ",
3. invokes the record's `PrintMembers` method giving it the builder, followed by " " if it returned true,
4. appends "}",
5. returns the builder's contents with `builder.ToString()`.

For example, consider the following record types:

```
record R1(T1 P1);
record R2(T1 P1, T2 P2, T3 P3) : R1(P1);
```

For those record types, the synthesized printing members would be something like:

```
class R1 : IEquatable<R1>
{
    public T1 P1 { get; init; }

    protected virtual bool PrintMembers(StringBuilder builder)
    {
        builder.Append(nameof(P1));
        builder.Append(" = ");
        builder.Append(this.P1); // or builder.Append(this.P1.ToString()); if P1 has a value type

        return true;
    }

    public override string ToString()
    {
        var builder = new StringBuilder();
        builder.Append(nameof(R1));
        builder.Append(" { ");

        if (PrintMembers(builder))
            builder.Append(" ");

        builder.Append("}");
        return builder.ToString();
    }
}

class R2 : R1, IEquatable<R2>
{
    public T2 P2 { get; init; }
    public T3 P3 { get; init; }

    protected override bool PrintMembers(StringBuilder builder)
    {
        if (base.PrintMembers(builder))
            builder.Append(", ");

        builder.Append(nameof(P2));
        builder.Append(" = ");
        builder.Append(this.P2); // or builder.Append(this.P2); if P2 has a value type

        builder.Append(", ");

        builder.Append(nameof(P3));
        builder.Append(" = ");
        builder.Append(this.P3); // or builder.Append(this.P3); if P3 has a value type

        return true;
    }

    public override string ToString()
    {
        var builder = new StringBuilder();
        builder.Append(nameof(R2));
        builder.Append(" { ");

        if (PrintMembers(builder))
            builder.Append(" ");

        builder.Append("}");
        return builder.ToString();
    }
}
```

Positional record members

In addition to the above members, records with a parameter list ("positional records") synthesize additional members with the same conditions as the members above.

Primary Constructor

A record type has a public constructor whose signature corresponds to the value parameters of the type declaration. This is called the primary constructor for the type, and causes the implicitly declared default class constructor, if present, to be suppressed. It is an error to have a primary constructor and a constructor with the same signature already present in the class.

At runtime the primary constructor

1. executes the instance initializers appearing in the class-body
2. invokes the base class constructor with the arguments provided in the `record_base` clause, if present

If a record has a primary constructor, any user-defined constructor, except "copy constructor" must have an explicit `this` constructor initializer.

Parameters of the primary constructor as well as members of the record are in scope within the `argument_list` of the `record_base` clause and within initializers of instance fields or properties. Instance members would be an error in these locations (similar to how instance members are in scope in regular constructor initializers today, but an error to use), but the parameters of the primary constructor would be in scope and useable and would shadow members. Static members would also be useable, similar to how base calls and initializers work in ordinary constructors today.

A warning is produced if a parameter of the primary constructor is not read.

Expression variables declared in the `argument_list` are in scope within the `argument_list`. The same shadowing rules as within an argument list of a regular constructor initializer apply.

Properties

For each record parameter of a record type declaration there is a corresponding public property member whose name and type are taken from the value parameter declaration.

For a record:

- A public `get` and `init` auto-property is created (see separate `init` accessor specification). An inherited `abstract` property with matching type is overridden. It is an error if the inherited property does not have `public` overridable `get` and `init` accessors. It is an error if the inherited property is hidden.
  The auto-property is initialized to the value of the corresponding primary constructor parameter. Attributes can be applied to the synthesized auto-property and its backing field by using `property:` or `field:` targets for attributes syntactically applied to the corresponding record parameter.

Deconstruct

A positional record with at least one parameter synthesizes a public void-returning instance method called Deconstruct with an out parameter declaration for each parameter of the primary constructor declaration. Each parameter of the Deconstruct method has the same type as the corresponding parameter of the primary constructor declaration. The body of the method assigns each parameter of the Deconstruct method to the value from an instance member access to a member of the same name. The method can be declared explicitly. It is an error if the explicit declaration does not match the expected signature or accessibility, or is static.

`with` expression

A `with` expression is a new expression using the following syntax.

```
with_expression
    : switch_expression
    | switch_expression 'with' '{' member_initializer_list? '}'
    ;

member_initializer_list
    : member_initializer (',' member_initializer)*
    ;

member_initializer
    : identifier '=' expression
    ;
```

A `with` expression is not permitted as a statement.

A `with` expression allows for "non-destructive mutation", designed to produce a copy of the receiver expression with modifications in assignments in the `member_initializer_list`.

A valid `with` expression has a receiver with a non-void type. The receiver type must be a record.

On the right hand side of the `with` expression is a `member_initializer_list` with a sequence of assignments to *identifier*, which must be an accessible instance field or property of the receiver's type.

First, receiver's "clone" method (specified above) is invoked and its result is converted to the receiver's type. Then, each `member_initializer` is processed the same way as an assignment to a field or property access of the result of the conversion. Assignments are processed in lexical order.

Top-level statements

18.12.2021 • 4 minutes to read

Summary

Allow a sequence of *statements* to occur right before the *namespace_member_declaration*s of a *compilation_unit* (i.e. source file).

The semantics are that if such a sequence of *statements* is present, the following type declaration, modulo the actual type name and the method name, would be emitted:

```
static class Program
{
    static async Task Main(string[] args)
    {
        // statements
    }
}
```

See also https://github.com/dotnet/csharplang/issues/3117.

Motivation

There's a certain amount of boilerplate surrounding even the simplest of programs, because of the need for an explicit `Main` method. This seems to get in the way of language learning and program clarity. The primary goal of the feature therefore is to allow C# programs without unnecessary boilerplate around them, for the sake of learners and the clarity of code.

Detailed design

Syntax

The only additional syntax is allowing a sequence of *statement*s in a compilation unit, just before the *namespace_member_declaration*s:

```
compilation_unit
    : extern_alias_directive* using_directive* global_attributes? statement* namespace_member_declaration*
    ;
```

Only one *compilation_unit* is allowed to have *statement*s.

Example:

```
if (args.Length == 0
    || !int.TryParse(args[0], out int n)
    || n < 0) return;
Console.WriteLine(Fib(n).curr);

(int curr, int prev) Fib(int i)
{
    if (i == 0) return (1, 0);
    var (curr, prev) = Fib(i - 1);
    return (curr + prev, curr);
}
```

Semantics

If any top-level statements are present in any compilation unit of the program, the meaning is as if they were combined in the block body of a `Main` method of a `Program` class in the global namespace, as follows:

```
static class Program
{
    static async Task Main(string[] args)
    {
        // statements
    }
}
```

Note that the names "Program" and "Main" are used only for illustrations purposes, actual names used by compiler are implementation dependent and neither the type, nor the method can be referenced by name from source code.

The method is designated as the entry point of the program. Explicitly declared methods that by convention could be considered as an entry point candidates are ignored. A warning is reported when that happens. It is an error to specify `-main:<type>` compiler switch when there are top-level statements.

The entry point method always has one formal parameter, `string[] args`. The execution environment creates and passes a `string[]` argument containing the command-line arguments that were specified when the application was started. The `string[]` argument is never null, but it may have a length of zero if no command-line arguments were specified. The 'args' parameter is in scope within top-level statements and is not in scope outside of them. Regular name conflict/shadowing rules apply.

Async operations are allowed in top-level statements to the degree they are allowed in statements within a regular async entry point method. However, they are not required, if `await` expressions and other async operations are omitted, no warning is produced.

The signature of the generated entry point method is determined based on operations used by the top level statements as follows:

ASYNC-OPERATIONS\RETURN-WITH-EXPRESSION	PRESENT	ABSENT
Present	`static Task<int> Main(string[] args)`	`static Task Main(string[] args)`
Absent	`static int Main(string[] args)`	`static void Main(string[] args)`

The example above would yield the following `$Main` method declaration:

```
static class $Program
{
    static void $Main(string[] args)
    {
        if (args.Length == 0
            || !int.TryParse(args[0], out int n)
            || n < 0) return;
        Console.WriteLine(Fib(n).curr);

        (int curr, int prev) Fib(int i)
        {
            if (i == 0) return (1, 0);
            var (curr, prev) = Fib(i - 1);
            return (curr + prev, curr);
        }
    }
}
```

At the same time an example like this:

```
await System.Threading.Tasks.Task.Delay(1000);
System.Console.WriteLine("Hi!");
```

would yield:

```
static class $Program
{
    static async Task $Main(string[] args)
    {
        await System.Threading.Tasks.Task.Delay(1000);
        System.Console.WriteLine("Hi!");
    }
}
```

An example like this:

```
await System.Threading.Tasks.Task.Delay(1000);
System.Console.WriteLine("Hi!");
return 0;
```

would yield:

```
static class $Program
{
    static async Task<int> $Main(string[] args)
    {
        await System.Threading.Tasks.Task.Delay(1000);
        System.Console.WriteLine("Hi!");
        return 0;
    }
}
```

And an example like this:

```
System.Console.WriteLine("Hi!");
return 2;
```

would yield:

```
static class $Program
{
    static int $Main(string[] args)
    {
        System.Console.WriteLine("Hi!");
        return 2;
    }
}
```

Scope of top-level local variables and local functions

Even though top-level local variables and functions are "wrapped" into the generated entry point method, they should still be in scope throughout the program in every compilation unit. For the purpose of simple-name evaluation, once the global namespace is reached:

- First, an attempt is made to evaluate the name within the generated entry point method and only if this attempt fails
- The "regular" evaluation within the global namespace declaration is performed.

This could lead to name shadowing of namespaces and types declared within the global namespace as well as to shadowing of imported names.

If the simple name evaluation occurs outside of the top-level statements and the evaluation yields a top-level local variable or function, that should lead to an error.

In this way we protect our future ability to better address "Top-level functions" (scenario 2 in https://github.com/dotnet/csharplang/issues/3117), and are able to give useful diagnostics to users who mistakenly believe them to be supported.

Nullable Reference Types Specification

18.12.2021 • 18 minutes to read

This is a work in progress - several parts are missing or incomplete.

This feature adds two new kinds of nullable types (nullable reference types and nullable generic types) to the existing nullable value types, and introduces a static flow analysis for purpose of null-safety.

Syntax

Nullable reference types and nullable type parameters

Nullable reference types and nullable type parameters have the same syntax `T?` as the short form of nullable value types, but do not have a corresponding long form.

For the purposes of the specification, the current `nullable_type` production is renamed to `nullable_value_type`, and `nullable_reference_type` and `nullable_type_parameter` productions are added:

```
type
    : value_type
    | reference_type
    | nullable_type_parameter
    | type_parameter
    | type_unsafe
    ;

reference_type
    : ...
    | nullable_reference_type
    ;

nullable_reference_type
    : non_nullable_reference_type '?'
    ;

non_nullable_reference_type
    : reference_type
    ;

nullable_type_parameter
    : non_nullable_non_value_type_parameter '?'
    ;

non_nullable_non_value_type_parameter
    : type_parameter
    ;
```

The `non_nullable_reference_type` in a `nullable_reference_type` must be a nonnullable reference type (class, interface, delegate or array).

The `non_nullable_non_value_type_parameter` in `nullable_type_parameter` must be a type parameter that isn't constrained to be a value type.

Nullable reference types and nullable type parameters cannot occur in the following positions:

- as a base class or interface
- as the receiver of a `member_access`

- as the `type` in an `object_creation_expression`
- as the `delegate_type` in a `delegate_creation_expression`
- as the `type` in an `is_expression`, a `catch_clause` or a `type_pattern`
- as the `interface` in a fully qualified interface member name

A warning is given on a `nullable_reference_type` and `nullable_type_parameter` in a *disabled* nullable annotation context.

`class` and `class?` constraint

The `class` constraint has a nullable counterpart `class?`:

```
primary_constraint
    : ...
    | 'class' '?'
    ;
```

A type parameter constrained with `class` (in an *enabled* annotation context) must be instantiated with a nonnullable reference type.

A type parameter constrained with `class?` (or `class` in a *disabled* annotation context) may either be instantiated with a nullable or nonnullable reference type.

A warning is given on a `class?` constraint in a *disabled* annotation context.

`notnull` constraint

A type parameter constrained with `notnull` may not be a nullable type (nullable value type, nullable reference type or nullable type parameter).

```
primary_constraint
    : ...
    | 'notnull'
    ;
```

`default` constraint

The `default` constraint can be used on a method override or explicit implementation to disambiguate `T?` meaning "nullable type parameter" from "nullable value type" (`Nullable<T>`). Lacking the `default` constraint a `T?` syntax in an override or explicit implementation will be interpreted as `Nullable<T>`

See https://github.com/dotnet/csharplang/blob/master/proposals/csharp-9.0/unconstrained-type-parameter-annotations.md#default-constraint

The null-forgiving operator

The post-fix `!` operator is called the null-forgiving operator. It can be applied on a *primary_expression* or within a *null_conditional_expression*:

```
primary_expression
    : ...
    | null_forgiving_expression
    ;

null_forgiving_expression
    : primary_expression '!'
    ;

null_conditional_expression
    : primary_expression null_conditional_operations_no_suppression suppression?
    ;

null_conditional_operations_no_suppression
    : null_conditional_operations? '?' '.' identifier type_argument_list?
    | null_conditional_operations? '?' '[' argument_list ']'
    | null_conditional_operations '.' identifier type_argument_list?
    | null_conditional_operations '[' argument_list ']'
    | null_conditional_operations '(' argument_list? ')'
    ;

null_conditional_operations
    : null_conditional_operations_no_suppression suppression?
    ;

suppression
    : '!'
    ;
```

For example:

```
var v = expr!;
expr!.M();
_ = a?.b!.c;
```

The postfix `!` operator has no runtime effect - it evaluates to the result of the underlying expression. Its only role is to change the null state of the expression to "not null", and to limit warnings given on its use.

Nullable compiler directives

`#nullable` directives control the nullable annotation and warning contexts.

```
pp_directive
    : ...
    | pp_nullable
    ;

pp_nullable
    : whitespace? '#' whitespace? 'nullable' whitespace nullable_action (whitespace nullable_target)?
pp_new_line
    ;

nullable_action
    : 'disable'
    | 'enable'
    | 'restore'
    ;

nullable_target
    : 'warnings'
    | 'annotations'
    ;
```

`#pragma warning` directives are expanded to allow changing the nullable warning context:

```
pragma_warning_body
    : ...
    | 'warning' whitespace warning_action whitespace 'nullable'
    ;
```

For example:

```
#pragma warning disable nullable
```

Nullable contexts

Every line of source code has a *nullable annotation context* and a *nullable warning context*. These control whether nullable annotations have effect, and whether nullability warnings are given. The annotation context of a given line is either *disabled* or *enabled*. The warning context of a given line is either *disabled* or *enabled*.

Both contexts can be specified at the project level (outside of C# source code), or anywhere within a source file via `#nullable` pre-processor directives. If no project level settings are provided the default is for both contexts to be *disabled*.

The `#nullable` directive controls the annotation and warning contexts within the source text, and take precedence over the project-level settings.

A directive sets the context(s) it controls for subsequent lines of code, until another directive overrides it, or until the end of the source file.

The effect of the directives is as follows:

- `#nullable disable`: Sets the nullable annotation and warning contexts to *disabled*
- `#nullable enable`: Sets the nullable annotation and warning contexts to *enabled*
- `#nullable restore`: Restores the nullable annotation and warning contexts to project settings
- `#nullable disable annotations`: Sets the nullable annotation context to *disabled*
- `#nullable enable annotations`: Sets the nullable annotation context to *enabled*
- `#nullable restore annotations`: Restores the nullable annotation context to project settings
- `#nullable disable warnings`: Sets the nullable warning context to *disabled*
- `#nullable enable warnings`: Sets the nullable warning context to *enabled*
- `#nullable restore warnings`: Restores the nullable warning context to project settings

Nullability of types

A given type can have one of three nullabilities: *oblivious*, *nonnullable*, and *nullable*.

Nonnullable types may cause warnings if a potential `null` value is assigned to them. *Oblivious* and *nullable* types, however, are "*null-assignable*" and can have `null` values assigned to them without warnings.

Values of *oblivious* and *nonnullable* types can be dereferenced or assigned without warnings. Values of *nullable* types, however, are "*null-yielding*" and may cause warnings when dereferenced or assigned without proper null checking.

The *default null state* of a null-yielding type is "maybe null" or "maybe default". The default null state of a non-null-yielding type is "not null".

The kind of type and the nullable annotation context it occurs in determine its nullability:

- A nonnullable value type `S` is always *nonnullable*
- A nullable value type `S?` is always *nullable*
- An unannotated reference type `C` in a *disabled* annotation context is *oblivious*
- An unannotated reference type `C` in an *enabled* annotation context is *nonnullable*
- A nullable reference type `C?` is *nullable* (but a warning may be yielded in a *disabled* annotation context)

Type parameters additionally take their constraints into account:

- A type parameter `T` where all constraints (if any) are either nullable types or the `class?` constraint is *nullable*
- A type parameter `T` where at least one constraint is either *oblivious* or *nonnullable* or one of the `struct` or `class` or `notnull` constraints is
  - *oblivious* in a *disabled* annotation context
  - *nonnullable* in an *enabled* annotation context
- A nullable type parameter `T?` is *nullable*, but a warning is yielded in a *disabled* annotation context if `T` isn't a value type

Oblivious vs nonnullable

A `type` is deemed to occur in a given annotation context when the last token of the type is within that context.

Whether a given reference type `C` in source code is interpreted as oblivious or nonnullable depends on the annotation context of that source code. But once established, it is considered part of that type, and "travels with it" e.g. during substitution of generic type arguments. It is as if there is an annotation like `?` on the type, but invisible.

Constraints

Nullable reference types can be used as generic constraints.

`class?` is a new constraint denoting "possibly nullable reference type", whereas `class` in an *enabled* annotation context denotes "nonnullable reference type".

`default` is a new constraint denoting a type parameter that isn't known to be a reference or value type. It can only be used on overridden and explicitly implemented methods. With this constraint, `T?` means a nullable type parameter, as opposed to being a shorthand for `Nullable<T>`.

`notnull` is a new constraint denoting a type parameter that is nonnullable.

The nullability of a type argument or of a constraint does not impact whether the type satisfies the constraint, except where that is already the case today (nullable value types do not satisfy the `struct` constraint). However, if the type argument does not satisfy the nullability requirements of the constraint, a warning may be given.

Null state and null tracking

Every expression in a given source location has a *null state*, which indicated whether it is believed to potentially evaluate to null. The null state is either "not null", "maybe null", or "maybe default". The null state is used to determine whether a warning should be given about null-unsafe conversions and dereferences.

The distinction between "maybe null" and "maybe default" is subtle and applies to type parameters. The distinction is that a type parameter `T` which has the state "maybe null" means the value is in the domain of legal values for `T` however that legal value may include `null`. Where as a "maybe default" means that the value may be outside the legal domain of values for `T`.

Example:

```
// The value `t` here has the state "maybe null". It's possible for `T` to be instantiated
// with `string?` in which case `null` would be within the domain of legal values here. The
// assumption though is the value provided here is within the legal values of `T`. Hence
// if `T` is `string` then `null` will not be a value, just as we assume that `null` is not
// provided for a normal `string` parameter
void M<T>(T t)
{
    // There is no guarantee that default(T) is within the legal values for T hence the
    // state *must* be "maybe-default" and hence `local` must be `T?`
    T? local = default(T);
}
```

Null tracking for variables

For certain expressions denoting variables, fields or properties, the null state is tracked between occurrences, based on assignments to them, tests performed on them and the control flow between them. This is similar to how definite assignment is tracked for variables. The tracked expressions are the ones of the following form:

```
tracked_expression
    : simple_name
    | this
    | base
    | tracked_expression '.' identifier
    ;
```

Where the identifiers denote fields or properties.

The null state for tracked variables is "not null" in unreachable code. This follows other decisions around unreachable code like considering all locals to be definitely assigned.

Describe null state transitions similar to definite assignment

Null state for expressions

The null state of an expression is derived from its form and type, and from the null state of variables involved in it.

Literals

The null state of a `null` literal depends on the target type of the expression. If the target type is a type parameter constrained to a reference type then it's "maybe default". Otherwise it is "maybe null".

The null state of a `default` literal depends on the target type of the `default` literal. A `default` literal with target type `T` has the same null state as the `default(T)` expression.

The null state of any other literal is "not null".

Simple names

If a `simple_name` is not classified as a value, its null state is "not null". Otherwise it is a tracked expression, and its null state is its tracked null state at this source location.

Member access

If a `member_access` is not classified as a value, its null state is "not null". Otherwise, if it is a tracked expression, its null state is its tracked null state at this source location. Otherwise its null state is the default null state of its type.

```
var person = new Person();

// The receiver is a tracked expression hence the member_access of the property
// is tracked as well
if (person.FirstName is not null)
{
    Use(person.FirstName);
}

// The return of an invocation is not a tracked expression hence the member_access
// of the return is also not tracked
if (GetAnonymous().FirstName is not null)
{
    // Warning: Cannot convert null literal to non-nullable reference type.
    Use(GetAnonymous().FirstName);
}

void Use(string s)
{
    // ...
}

public class Person
{
    public string? FirstName { get; set; }
    public string? LastName { get; set; }

    private static Person s_anonymous = new Person();
    public static Person GetAnonymous() => s_anonymous;
}
```

Invocation expressions

If an `invocation_expression` invokes a member that is declared with one or more attributes for special null behavior, the null state is determined by those attributes. Otherwise the null state of the expression is the default null state of its type.

The null state of an `invocation_expression` is not tracked by the compiler.

```
// The result of an invocation_expression is not tracked
if (GetText() is not null)
{
    // Warning: Converting null literal or possible null value to non-nullable type.
    string s = GetText();
    // Warning: Dereference of a possibly null reference.
    Use(s);
}

// Nullable friendly pattern
if (GetText() is string s)
{
    Use(s);
}

string? GetText() => ...
Use(string s) {  }
```

Element access

If an `element_access` invokes an indexer that is declared with one or more attributes for special null behavior, the null state is determined by those attributes. Otherwise the null state of the expression is the default null state of its type.

```
object?[] array = ...;
if (array[0] != null)
{
    // Warning: Converting null literal or possible null value to non-nullable type.
    object o = array[0];
    // Warning: Dereference of a possibly null reference.
    Console.WriteLine(o.ToString());
}

// Nullable friendly pattern
if (array[0] is {} o)
{
    Console.WriteLine(o.ToString());
}
```

Base access

If `B` denotes the base type of the enclosing type, `base.I` has the same null state as `((B)this).I` and `base[E]` has the same null state as `((B)this)[E]`.

Default expressions

`default(T)` has the null state based on the properties of the type `T`:

- If the type is a *nonnullable* type then it has the null state "not null"
- Else if the type is a type parameter then it has the null state "maybe default"
- Else it has the null state "maybe null"

Null-conditional expressions ?.

A `null_conditional_expression` has the null state based on the expression type. Note that this refers to the type of the `null_conditional_expression`, not the original type of the member being invoked:

- If the type is a *nullable* value type then it has the null state "maybe null"
- Else if the type is a *nullable* type parameter then it has the null state "maybe default"
- Else it has the null state "maybe null"

Cast expressions

If a cast expression `(T)E` invokes a user-defined conversion, then the null state of the expression is the default null state for the type of the user-defined conversion. Otherwise:

- If `T` is a *nonnullable* value type then `T` has the null state "not null"
- Else if `T` is a *nullable* value type then `T` has the null state "maybe null"
- Else if `T` is a *nullable* type in the form `U?` where `U` is a type parameter then `T` has the null state "maybe default"
- Else if `T` is a *nullable* type, and `E` has null state "maybe null" or "maybe default", then `T` has the null state "maybe null"
- Else if `T` is a type parameter, and `E` has null state "maybe null" or "maybe default", then `T` has the null state "maybe default"
- Else `T` has the same null state as `E`

Unary and binary operators

If a unary or binary operator invokes an user-defined operator then the null state of the expression is the default null state for the type of the user-defined operator. Otherwise it is the null state of the expression.

Something special to do for binary `+` over strings and delegates?

Await expressions

The null state of `await E` is the default null state of its type.

The `as` operator

The null state of an `E as T` expression depends first on properties of the type `T`. If the type of `T` is *nonnullable* then the null state is "not null". Otherwise the null state depends on the conversion from the type of `E` to type `T`:

- If the conversion is an identity, boxing, implicit reference, or implicit nullable conversion, then the null state is the null state of `E`
- Else if `T` is a type parameter then it has the null state "maybe default"
- Else it has the null state "maybe null"

The null-coalescing operator

The null state of `E1 ?? E2` is the null state of `E2`

The conditional operator

The null state of `E1 ? E2 : E3` is based on the null state of `E2` and `E3`:

- If both are "not null" then the null state is "not null"
- Else if either is "maybe default" then the null state is "maybe default"
- Else the null state is "not null"

Query expressions

The null state of a query expression is the default null state of its type.

Additional work needed here

Assignment operators

`E1 = E2` and `E1 op= E2` have the same null state as `E2` after any implicit conversions have been applied.

Expressions that propagate null state

`(E)`, `checked(E)` and `unchecked(E)` all have the same null state as `E`.

Expressions that are never null

The null state of the following expression forms is always "not null":

- `this` access
- interpolated strings
- `new` expressions (object, delegate, anonymous object and array creation expressions)
- `typeof` expressions
- `nameof` expressions
- anonymous functions (anonymous methods and lambda expressions)
- null-forgiving expressions
- `is` expressions

Nested functions

Nested functions (lambdas and local functions) are treated like methods, except in regards to their captured variables. The initial state of a captured variable inside a lambda or local function is the intersection of the nullable state of the variable at all the "uses" of that nested function or lambda. A use of a local function is either a call to that function, or where it is converted to a delegate. A use of a lambda is the point at which it is defined in source.

Type inference

nullable implicitly typed local variables

`var` infers an annotated type for reference types, and type parameters that aren't constrained to be a value type. For instance:

- in `var s = "";` the `var` is inferred as `string?`.
- in `var t = new T();` with an unconstrained `T` the `var` is inferred as `T?`.

Generic type inference

Generic type inference is enhanced to help decide whether inferred reference types should be nullable or not. This is a best effort. It may yield warnings regarding nullability constraints, and may lead to nullable warnings when the inferred types of the selected overload are applied to the arguments.

The first phase

Nullable reference types flow into the bounds from the initial expressions, as described below. In addition, two new kinds of bounds, namely `null` and `default` are introduced. Their purpose is to carry through occurrences of `null` or `default` in the input expressions, which may cause an inferred type to be nullable, even when it otherwise wouldn't.

The determination of what bounds to add in the first phase are enhanced as follows:

If an argument `Ei` has a reference type, the type `U` used for inference depends on the null state of `Ei` as well as its declared type:

- If the declared type is a nonnullable reference type `U0` or a nullable reference type `U0?` then
  - if the null state of `Ei` is "not null" then `U` is `U0`
  - if the null state of `Ei` is "maybe null" then `U` is `U0?`
- Otherwise if `Ei` has a declared type, `U` is that type
- Otherwise if `Ei` is `null` then `U` is the special bound `null`
- Otherwise if `Ei` is `default` then `U` is the special bound `default`
- Otherwise no inference is made.

Exact, upper-bound and lower-bound inferences

In inferences *from* the type `U` *to* the type `V`, if `V` is a nullable reference type `V0?`, then `V0` is used instead of `V` in the following clauses.

- If `V` is one of the unfixed type variables, `U` is added as an exact, upper or lower bound as before
- Otherwise, if `U` is `null` or `default`, no inference is made
- Otherwise, if `U` is a nullable reference type `U0?`, then `U0` is used instead of `U` in the subsequent clauses.

The essence is that nullability that pertains directly to one of the unfixed type variables is preserved into its bounds. For the inferences that recurse further into the source and target types, on the other hand, nullability is ignored. It may or may not match, but if it doesn't, a warning will be issued later if the overload is chosen and applied.

Fixing

The spec currently does not do a good job of describing what happens when multiple bounds are identity convertible to each other, but are different. This may happen between `object` and `dynamic`, between tuple types that differ only in element names, between types constructed thereof and now also between `C` and `C?` for reference types.

In addition we need to propagate "nullness" from the input expressions to the result type.

To handle these we add more phases to fixing, which is now:

1. Gather all the types in all the bounds as candidates, removing `?` from all that are nullable reference types

2. Eliminate candidates based on requirements of exact, lower and upper bounds (keeping `null` and `default` bounds)
3. Eliminate candidates that do not have an implicit conversion to all the other candidates
4. If the remaining candidates do not all have identity conversions to one another, then type inference fails
5. *Merge* the remaining candidates as described below
6. If the resulting candidate is a reference type and *all* of the exact bounds or *any* of the lower bounds are nullable reference types, `null` or `default`, then `?` is added to the resulting candidate, making it a nullable reference type.

Merging is described between two candidate types. It is transitive and commutative, so the candidates can be merged in any order with the same ultimate result. It is undefined if the two candidate types are not identity convertible to each other.

The *Merge* function takes two candidate types and a direction (*+* or *-*):

- *Merge*(`T`, `T`, *d*) = T
- *Merge*(`S`, `T?`, *+*) = *Merge*(`S?`, `T`, *+*) = *Merge*(`S`, `T`, *+*)`?`
- *Merge*(`S`, `T?`, *-*) = *Merge*(`S?`, `T`, *-*) = *Merge*(`S`, `T`, *-*)
- *Merge*(`C<S1,...,Sn>`, `C<T1,...,Tn>`, *+*) = `C<` *Merge*(`S1`, `T1`, *d1*)`,...,` *Merge*(`Sn`, `Tn`, *dn*)`>`, *where*
  - `di` = *+* if the `i`'th type parameter of `C<...>` is covariant
  - `di` = *-* if the `i`'th type parameter of `C<...>` is contra- or invariant
- *Merge*(`C<S1,...,Sn>`, `C<T1,...,Tn>`, *-*) = `C<` *Merge*(`S1`, `T1`, *d1*)`,...,` *Merge*(`Sn`, `Tn`, *dn*)`>`, *where*
  - `di` = *-* if the `i`'th type parameter of `C<...>` is covariant
  - `di` = *+* if the `i`'th type parameter of `C<...>` is contra- or invariant
- *Merge*(`(S1 s1,..., Sn sn)`, `(T1 t1,..., Tn tn)`, *d*) = `(` *Merge*(`S1`, `T1`, *d*)`n1,...,` *Merge*(`Sn`, `Tn`, *d*)`nn)`, *where*
  - `ni` is absent if `si` and `ti` differ, or if both are absent
  - `ni` is `si` if `si` and `ti` are the same
- *Merge*(`object`, `dynamic`) = *Merge*(`dynamic`, `object`) = `dynamic`

Warnings

Potential null assignment

Potential null dereference

Constraint nullability mismatch

Nullable types in disabled annotation context

Override and implementation nullability mismatch

Attributes for special null behavior

Pattern-matching changes for C# 9.0

18.12.2021 • 11 minutes to read

We are considering a small handful of enhancements to pattern-matching for C# 9.0 that have natural synergy and work well to address a number of common programming problems:

- https://github.com/dotnet/csharplang/issues/2925 Type patterns
- https://github.com/dotnet/csharplang/issues/1350 Parenthesized patterns to enforce or emphasize precedence of the new combinators
- https://github.com/dotnet/csharplang/issues/1350 Conjunctive `and` patterns that require both of two different patterns to match;
- https://github.com/dotnet/csharplang/issues/1350 Disjunctive `or` patterns that require either of two different patterns to match;
- https://github.com/dotnet/csharplang/issues/1350 Negated `not` patterns that require a given pattern *not* to match; and
- https://github.com/dotnet/csharplang/issues/812 Relational patterns that require the input value to be less than, less than or equal to, etc a given constant.

Parenthesized Patterns

Parenthesized patterns permit the programmer to put parentheses around any pattern. This is not so useful with the existing patterns in C# 8.0, however the new pattern combinators introduce a precedence that the programmer may want to override.

```
primary_pattern
    : parenthesized_pattern
    | // all of the existing forms
    ;
parenthesized_pattern
    : '(' pattern ')'
    ;
```

Type Patterns

We permit a type as a pattern:

```
primary_pattern
    : type-pattern
    | // all of the existing forms
    ;
type_pattern
    : type
    ;
```

This retcons the existing *is-type-expression* to be an *is-pattern-expression* in which the pattern is a *type-pattern*, though we would not change the syntax tree produced by the compiler.

One subtle implementation issue is that this grammar is ambiguous. A string such as `a.b` can be parsed either as a qualified name (in a type context) or a dotted expression (in an expression context). The compiler is already capable of treating a qualified name the same as a dotted expression in order to handle something like

`e is Color.Red`. The compiler's semantic analysis would be further extended to be capable of binding a (syntactic) constant pattern (e.g. a dotted expression) as a type in order to treat it as a bound type pattern in order to support this construct.

After this change, you would be able to write

```
void M(object o1, object o2)
{
    var t = (o1, o2);
    if (t is (int, string)) {} // test if o1 is an int and o2 is a string
    switch (o1) {
        case int: break; // test if o1 is an int
        case System.String: break; // test if o1 is a string
    }
}
```

Relational Patterns

Relational patterns permit the programmer to express that an input value must satisfy a relational constraint when compared to a constant value:

```
    public static LifeStage LifeStageAtAge(int age) => age switch
    {
        < 0 =>  LifeStage.Prenatal,
        < 2 =>  LifeStage.Infant,
        < 4 =>  LifeStage.Toddler,
        < 6 =>  LifeStage.EarlyChild,
        < 12 => LifeStage.MiddleChild,
        < 20 => LifeStage.Adolescent,
        < 40 => LifeStage.EarlyAdult,
        < 65 => LifeStage.MiddleAdult,
        _ =>    LifeStage.LateAdult,
    };
```

Relational patterns support the relational operators `<`, `<=`, `>`, and `>=` on all of the built-in types that support such binary relational operators with two operands of the same type in an expression. Specifically, we support all of these relational patterns for `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `nint`, and `nuint`.

```
primary_pattern
    : relational_pattern
    ;
relational_pattern
    : '<' relational_expression
    | '<=' relational_expression
    | '>' relational_expression
    | '>=' relational_expression
    ;
```

The expression is required to evaluate to a constant value. It is an error if that constant value is `double.NaN` or `float.NaN`. It is an error if the expression is a null constant.

When the input is a type for which a suitable built-in binary relational operator is defined that is applicable with the input as its left operand and the given constant as its right operand, the evaluation of that operator is taken as the meaning of the relational pattern. Otherwise we convert the input to the type of the expression using an explicit nullable or unboxing conversion. It is a compile-time error if no such conversion exists. The pattern is considered not to match if the conversion fails. If the conversion succeeds then the result of the pattern-

matching operation is the result of evaluating the expression `e OP v` where `e` is the converted input, `OP` is the relational operator, and `v` is the constant expression.

Pattern Combinators

Pattern *combinators* permit matching both of two different patterns using `and` (this can be extended to any number of patterns by the repeated use of `and`), either of two different patterns using `or` (ditto), or the *negation* of a pattern using `not`.

A common use of a combinator will be the idiom

```
if (e is not null) ...
```

More readable than the current idiom `e is object`, this pattern clearly expresses that one is checking for a non-null value.

The `and` and `or` combinators will be useful for testing ranges of values

```
bool IsLetter(char c) => c is >= 'a' and <= 'z' or >= 'A' and <= 'Z';
```

This example illustrates that `and` will have a higher parsing priority (i.e. will bind more closely) than `or`. The programmer can use the *parenthesized pattern* to make the precedence explicit:

```
bool IsLetter(char c) => c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z');
```

Like all patterns, these combinators can be used in any context in which a pattern is expected, including nested patterns, the *is-pattern-expression*, the *switch-expression*, and the pattern of a switch statement's case label.

```
pattern
    : disjunctive_pattern
    ;
disjunctive_pattern
    : disjunctive_pattern 'or' conjunctive_pattern
    | conjunctive_pattern
    ;
conjunctive_pattern
    : conjunctive_pattern 'and' negated_pattern
    | negated_pattern
    ;
negated_pattern
    : 'not' negated_pattern
    | primary_pattern
    ;
primary_pattern
    : // all of the patterns forms previously defined
    ;
```

Change to 7.5.4.2 Grammar Ambiguities

Due to the introduction of the *type pattern*, it is possible for a generic type to appear before the token `=>`. We therefore add `=>` to the set of tokens listed in *7.5.4.2 Grammar Ambiguities* to permit disambiguation of the `<` that begins the type argument list. See also https://github.com/dotnet/roslyn/issues/47614.

Open Issues with Proposed Changes

Syntax for relational operators

Are `and`, `or`, and `not` some kind of contextual keyword? If so, is there a breaking change (e.g. compared to their use as a designator in a *declaration-pattern*).

Semantics (e.g. type) for relational operators

We expect to support all of the primitive types that can be compared in an expression using a relational operator. The meaning in simple cases is clear

```
bool IsValidPercentage(int x) => x is >= 0 and <= 100;
```

But when the input is not such a primitive type, what type do we attempt to convert it to?

```
bool IsValidPercentage(object x) => x is >= 0 and <= 100;
```

We have proposed that when the input type is already a comparable primitive, that is the type of the comparison. However, when the input is not a comparable primitive, we treat the relational as including an implicit type test to the type of the constant on the right-hand-side of the relational. If the programmer intends to support more than one input type, that must be done explicitly:

```
bool IsValidPercentage(object x) => x is
    >= 0 and <= 100 or    // integer tests
    >= 0F and <= 100F or  // float tests
    >= 0D and <= 100D;    // double tests
```

Flowing type information from the left to the right of `and`

It has been suggested that when you write an `and` combinator, type information learned on the left about the top-level type could flow to the right. For example

```
bool isSmallByte(object o) => o is byte and < 100;
```

Here, the *input type* to the second pattern is narrowed by the *type narrowing* requirements of left of the `and`. We would define type narrowing semantics for all patterns as follows. The *narrowed type* of a pattern `P` is defined as follows:

1. If `P` is a type pattern, the *narrowed type* is the type of the type pattern's type.
2. If `P` is a declaration pattern, the *narrowed type* is the type of the declaration pattern's type.
3. If `P` is a recursive pattern that gives an explicit type, the *narrowed type* is that type.
4. If `P` is matched via the rules for `ITuple`, the *narrowed type* is the type `System.Runtime.CompilerServices.ITuple`.
5. If `P` is a constant pattern where the constant is not the null constant and where the expression has no *constant expression conversion* to the *input type*, the *narrowed type* is the type of the constant.
6. If `P` is a relational pattern where the constant expression has no *constant expression conversion* to the *input type*, the *narrowed type* is the type of the constant.
7. If `P` is an `or` pattern, the *narrowed type* is the common type of the *narrowed type* of the subpatterns if such a common type exists. For this purpose, the common type algorithm considers only identity, boxing, and implicit reference conversions, and it considers all subpatterns of a sequence of `or` patterns (ignoring parenthesized patterns).
8. If `P` is an `and` pattern, the *narrowed type* is the *narrowed type* of the right pattern. Moreover, the *narrowed type* of the left pattern is the *input type* of the right pattern.
9. Otherwise the *narrowed type* of `P` is `P`'s input type.

Variable definitions and definite assignment

The addition of `or` and `not` patterns creates some interesting new problems around pattern variables and definite assignment. Since variables can normally be declared at most once, it would seem any pattern variable declared on one side of an `or` pattern would not be definitely assigned when the pattern matches. Similarly, a variable declared inside a `not` pattern would not be expected to be definitely assigned when the pattern matches. The simplest way to address this is to forbid declaring pattern variables in these contexts. However, this may be too restrictive. There are other approaches to consider.

One scenario that is worth considering is this

```
if (e is not int i) return;
M(i); // is i definitely assigned here?
```

This does not work today because, for an *is-pattern-expression*, the pattern variables are considered *definitely assigned* only where the *is-pattern-expression* is true ("definitely assigned when true").

Supporting this would be simpler (from the programmer's perspective) than also adding support for a negated-condition `if` statement. Even if we add such support, programmers would wonder why the above snippet does not work. On the other hand, the same scenario in a `switch` makes less sense, as there is no corresponding point in the program where *definitely assigned when false* would be meaningful. Would we permit this in an *is-pattern-expression* but not in other contexts where patterns are permitted? That seems irregular.

Related to this is the problem of definite assignment in a *disjunctive-pattern*.

```
if (e is 0 or int i)
{
    M(i); // is i definitely assigned here?
}
```

We would only expect `i` to be definitely assigned when the input is not zero. But since we don't know whether the input is zero or not inside the block, `i` is not definitely assigned. However, what if we permit `i` to be declared in different mutually exclusive patterns?

```
if ((e1, e2) is (0, int i) or (int i, 0))
{
    M(i);
}
```

Here, the variable `i` is definitely assigned inside the block, and takes it value from the other element of the tuple when a zero element is found.

It has also been suggested to permit variables to be (multiply) defined in every case of a case block:

```
    case (0, int x):
    case (int x, 0):
        Console.WriteLine(x);
```

To make any of this work, we would have to carefully define where such multiple definitions are permitted and under what conditions such a variable is considered definitely assigned.

Should we elect to defer such work until later (which I advise), we could say in C# 9

- beneath a `not` or `or`, pattern variables may not be declared.

Then, we would have time to develop some experience that would provide insight into the possible value of

relaxing that later.

Diagnostics, subsumption, and exhaustiveness

These new pattern forms introduce many new opportunities for diagnosable programmer error. We will need to decide what kinds of errors we will diagnose, and how to do so. Here are some examples:

```
case >= 0 and <= 100D:
```

This case can never match (because the input cannot be both an `int` and a `double`). We already have an error when we detect a case that can never match, but its wording ("The switch case has already been handled by a previous case" and "The pattern has already been handled by a previous arm of the switch expression") may be misleading in new scenarios. We may have to modify the wording to just say that the pattern will never match the input.

```
case 1 and 2:
```

Similarly, this would be an error because a value cannot be both `1` and `2`.

```
case 1 or 2 or 3 or 1:
```

This case is possible to match, but the `or 1` at the end adds no meaning to the pattern. I suggest we should aim to produce an error whenever some conjunct or disjunct of a compound pattern does not either define a pattern variable or affect the set of matched values.

```
case < 2: break;
case 0 or 1 or 2 or 3 or 4 or 5: break;
```

Here, `0 or 1 or` adds nothing to the second case, as those values would have been handled by the first case. This too deserves an error.

```
byte b = ...;
int x = b switch { <100 => 0, 100 => 1, 101 => 2, >101 => 3 };
```

A switch expression such as this should be considered *exhaustive* (it handles all possible input values).

In C# 8.0, a switch expression with an input of type `byte` is only considered exhaustive if it contains a final arm whose pattern matches everything (a *discard-pattern* or *var-pattern*). Even a switch expression that has an arm for every distinct `byte` value is not considered exhaustive in C# 8. In order to properly handle exhaustiveness of relational patterns, we will have to handle this case too. This will technically be a breaking change, but no user is likely to notice.

Init Only Setters

18.12.2021 • 15 minutes to read

Summary

This proposal adds the concept of init only properties and indexers to C#. These properties and indexers can be set at the point of object creation but become effectively `get` only once object creation has completed. This allows for a much more flexible immutable model in C#.

Motivation

The underlying mechanisms for building immutable data in C# haven't changed since 1.0. They remain:

1. Declaring fields as `readonly`.
2. Declaring properties that contain only a `get` accessor.

These mechanisms are effective at allowing the construction of immutable data but they do so by adding cost to the boilerplate code of types and opting such types out of features like object and collection initializers. This means developers must choose between ease of use and immutability.

A simple immutable object like `Point` requires twice as much boiler plate code to support construction as it does to declare the type. The bigger the type the bigger the cost of this boiler plate:

```
struct Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y)
    {
        this.X = x;
        this.Y = y;
    }
}
```

The `init` accessor makes immutable objects more flexible by allowing the caller to mutate the members during the act of construction. That means the object's immutable properties can participate in object initializers and thus removes the need for all constructor boilerplate in the type. The `Point` type is now simply:

```
struct Point
{
    public int X { get; init; }
    public int Y { get; init; }
}
```

The consumer can then use object initializers to create the object

```
var p = new Point() { X = 42, Y = 13 };
```

Detailed Design

init accessors

An init only property (or indexer) is declared by using the `init` accessor in place of the `set` accessor:

```
class Student
{
    public string FirstName { get; init; }
    public string LastName { get; init; }
}
```

An instance property containing an `init` accessor is considered settable in the following circumstances, except when in a local function or lambda:

- During an object initializer
- During a `with` expression initializer
- Inside an instance constructor of the containing or derived type, on `this` or `base`
- Inside the `init` accessor of any property, on `this` or `base`
- Inside attribute usages with named parameters

The times above in which the `init` accessors are settable are collectively referred to in this document as the construction phase of the object.

This means the `Student` class can be used in the following ways:

```
var s = new Student()
{
    FirstName = "Jared",
    LastName = "Parosns",
};
s.LastName = "Parsons"; // Error: LastName is not settable
```

The rules around when `init` accessors are settable extend across type hierarchies. If the member is accessible and the object is known to be in the construction phase then the member is settable. That specifically allows for the following:

```
class Base
{
    public bool Value { get; init; }
}

class Derived : Base
{
    Derived()
    {
        // Not allowed with get only properties but allowed with init
        Value = true;
    }
}

class Consumption
{
    void Example()
    {
        var d = new Derived() { Value = true; };
    }
}
```

At the point an `init` accessor is invoked, the instance is known to be in the open construction phase. Hence an

`init` accessor is allowed to take the following actions in addition to what a normal `set` accessor can do:

1. Call other `init` accessors available through `this` or `base`
2. Assign `readonly` fields declared on the same type through `this`

```
class Complex
{
    readonly int Field1;
    int Field2;
    int Prop1 { get; init ; }
    int Prop2
    {
        get => 42;
        init
        {
            Field1 = 13; // okay
            Field2 = 13; // okay
            Prop1 = 13; // okay
        }
    }
}
```

The ability to assign `readonly` fields from an `init` accessor is limited to those fields declared on the same type as the accessor. It cannot be used to assign `readonly` fields in a base type. This rule ensures that type authors remain in control over the mutability behavior of their type. Developers who do not wish to utilize `init` cannot be impacted from other types choosing to do so:

```
class Base
{
    internal readonly int Field;
    internal int Property
    {
        get => Field;
        init => Field = value; // Okay
    }

    internal int OtherProperty { get; init; }
}

class Derived : Base
{
    internal readonly int DerivedField;
    internal int DerivedProperty
    {
        get => DerivedField;
        init
        {
            DerivedField = 42;  // Okay
            Property = 0;       // Okay
            Field = 13;         // Error Field is readonly
        }
    }

    public Derived()
    {
        Property = 42;  // Okay
        Field = 13;     // Error Field is readonly
    }
}
```

When `init` is used in a virtual property then all the overrides must also be marked as `init`. Likewise it is not possible to override a simple `set` with `init`.

```
class Base
{
    public virtual int Property { get; init; }
}

class C1 : Base
{
    public override int Property { get; init; }
}

class C2 : Base
{
    // Error: Property must have init to override Base.Property
    public override int Property { get; set; }
}
```

An `interface` declaration can also participate in `init` style initialization via the following pattern:

```
interface IPerson
{
    string Name { get; init; }
}

class Init
{
    void M<T>() where T : IPerson, new()
    {
        var local = new T()
        {
            Name = "Jared"
        };
        local.Name = "Jraed"; // Error
    }
}
```

Restrictions of this feature:

- The `init` accessor can only be used on instance properties
- A property cannot contain both an `init` and `set` accessor
- All overrides of a property must have `init` if the base had `init`. This rule also applies to interface implementation.

Readonly structs

`init` accessors (both auto-implemented accessors and manually-implemented accessors) are permitted on properties of `readonly struct`s, as well as `readonly` properties. `init` accessors are not permitted to be marked `readonly` themselves, in both `readonly` and non-`readonly` `struct` types.

```
readonly struct ReadonlyStruct1
{
    public int Prop1 { get; init; } // Allowed
}

struct ReadonlyStruct2
{
    public readonly int Prop2 { get; init; } // Allowed

    public int Prop3 { get; readonly init; } // Error
}
```

Metadata encoding

Property `init` accessors will be emitted as a standard `set` accessor with the return type marked with a modreq of `IsExternalInit`. This is a new type which will have the following definition:

```
namespace System.Runtime.CompilerServices
{
    public sealed class IsExternalInit
    {
    }
}
```

The compiler will match the type by full name. There is no requirement that it appear in the core library. If there are multiple types by this name then the compiler will tie break in the following order:

1. The one defined in the project being compiled
2. The one defined in corelib

If neither of these exist then a type ambiguity error will be issued.

The design for `IsExternalInit` is futher covered in this issue

Questions

Breaking changes

One of the main pivot points in how this feature is encoded will come down to the following question:

> Is it a binary breaking change to replace `init` with `set`?

Replacing `init` with `set` and thus making a property fully writable is never a source breaking change on a non-virtual property. It simply expands the set of scenarios where the property can be written. The only behavior in question is whether or not this remains a binary breaking change.

If we want to make the change of `init` to `set` a source and binary compatible change then it will force our hand on the modreq vs. attributes decision below because it will rule out modreqs as a solution. If on the other hand this is seen as a non-interesting then this will make the modreq vs. attribute decision less impactful.

Resolution This scenario is not seen as compelling by LDM.

Modreqs vs. attributes

The emit strategy for `init` property accessors must choose between using attributes or modreqs when emitting during metadata. These have different trade offs that need to be considered.

Annotating a property set accessor with a modreq declaration means CLI compliant compilers will ignore the accessor unless it understands the modreq. That means only compilers aware of `init` will read the member. Compilers unaware of `init` will ignore the `set` accessor and hence will not accidentally treat the property as read / write.

The downside of modreq is `init` becomes a part of the binary signature of the `set` accessor. Adding or removing `init` will break binary compatbility of the application.

Using attributes to annotate the `set` accessor means that only compilers which understand the attribute will know to limit access to it. A compiler unaware of `init` will see it as a simple read / write property and allow access.

This would seemingly mean this decision is a choice between extra safety at the expense of binary compatibility. Digging in a bit the extra safety is not exactly what it seems. It will not protect against the following circumstances:

1. Reflection over `public` members
2. The use of `dynamic`
3. Compilers that don't recognize modreqs

It should also be considered that, when we complete the IL verification rules for .NET 5, `init` will be one of those rules. That means extra enforcement will be gained from simply verifying compilers emitting verifiable IL.

The primary languages for .NET (C#, F# and VB) will all be updated to recognize these `init` accessors. Hence the only realistic scenario here is when a C# 9 compiler emits `init` properties and they are seen by an older toolset such as C# 8, VB 15, etc ... C# 8. That is the trade off to consider and weigh against binary compatibility.

Note This discussion primarily applies to members only, not to fields. While `init` fields were rejected by LDM they are still interesting to consider for the modreq vs. attribute discussion. The `init` feature for fields is a relaxation of the existing restriction of `readonly`. That means if we emit the fields as `readonly` + an attribute there is no risk of older compilers mis-using the field because they would already recognize `readonly`. Hence using a modreq here doesn't add any extra protection.

Resolution The feature will use a modreq to encode the property `init` setter. The compelling factors were (in no particular order):

- Desire to discourage older compilers from violating `init` semantics
- Desire to make adding or removing `init` in a `virtual` declaration or `interface` both a source and binary breaking change.

Given there was also no significant support for removing `init` to be a binary compatible change it made the choice of using modreq straight forward.

init vs. initonly

There were three syntax forms which got significant consideration during our LDM meeting:

```
// 1. Use init
int Option1 { get; init; }
// 2. Use init set
int Option2 { get; init set; }
// 3. Use initonly
int Option3 { get; initonly; }
```

Resolution There was no syntax which was overwhelmingly favored in LDM.

One point which got significant attention was how the choice of syntax would impact our ability to do `init` members as a general feature in the future. Choosing option 1 would mean that it would be difficult to define a property which had an `init` style `get` method in the future. Eventually it was decided that if we decided to go forward with general `init` members in future, we could allow `init` to be a modifier in the property accessor list as well as a short hand for `init set`. Essentially the following two declarations would be identical.

```
int Property1 { get; init; }
int Property1 { get; init set; }
```

The decision was made to move forward with `init` as a standalone accessor in the property accessor list.

Warn on failed init

Consider the following scenario. A type declares an `init` only member which is not set in the constructor. Should the code which constructs the object get a warning if they failed to initialize the value?

At that point it is clear the field will never be set and hence has a lot of similarities with the warning around

failing to initialize `private` data. Hence a warning would seemingly have some value here?

There are significant downsides to this warning though:

1. It complicates the compatibility story of changing `readonly` to `init`.
2. It requires carrying additional metadata around to denote the members which are required to be initialized by the caller.

Further if we believe there is value here in the overall scenario of forcing object creators to be warned / error'd about specific fields then this likely makes sense as a general feature. There is no reason it should be limited to just `init` members.

Resolution There will be no warning on consumption of `init` fields and properties.

LDM wants to have a broader discussion on the idea of required fields and properties. That may cause us to come back and reconsider our position on `init` members and validation.

Allow init as a field modifier

In the same way `init` can serve as a property accessor it could also serve as a designation on fields to give them similar behaviors as `init` properties. That would allow for the field to be assigned before construction was complete by the type, derived types, or object initializers.

```
class Student
{
    public init string FirstName;
    public init string LastName;
}

var s = new Student()
{
    FirstName = "Jarde",
    LastName = "Parsons",
}

s.FirstName = "Jared"; // Error FirstName is readonly
```

In metadata these fields would be marked in the same way as `readonly` fields but with an additional attribute or modreq to indicate they are `init` style fields.

Resolution LDM agrees this proposal is sound but overall the scenario felt disjoint from properties. The decision was to proceed only with `init` properties for now. This has a suitable level of flexibility as an `init` property can mutate a `readonly` field on the declaring type of the property. This will be reconsidered if there is significant customer feedback that justifies the scenario.

Allow init as a type modifier

In the same way the `readonly` modifier can be applied to a `struct` to automatically declare all fields as `readonly`, the `init` only modifier can be declared on a `struct` or `class` to automatically mark all fields as `init`. This means the following two type declarations are equivalent:

```
struct Point
{
    public init int X;
    public init int Y;
}

// vs.

init struct Point
{
    public int X;
    public int Y;
}
```

Resolution This feature is too *cute* here and conflicts with the `readonly struct` feature on which it is based. The `readonly struct` feature is simple in that it applies `readonly` to all members: fields, methods, etc ... The `init struct` feature would only apply to properties. This actually ends up making it confusing for users.

Given that `init` is only valid on certain aspects of a type, we rejected the idea of having it as a type modifier.

Considerations

Compatibility

The `init` feature is designed to be compatible with existing `get` only properties. Specifically it is meant to be a completely additive change for a property which is `get` only today but desires more flexbile object creation semantics.

For example consider the following type:

```
class Name
{
    public string First { get; }
    public string Last { get; }

    public Name(string first, string last)
    {
        First = first;
        Last = last;
    }
}
```

Adding `init` to these properties is a non-breaking change:

```
class Name
{
    public string First { get; init; }
    public string Last { get; init; }

    public Name(string first, string last)
    {
        First = first;
        Last = last;
    }
}
```

IL verification

When .NET Core decides to re-implement IL verification, the rules will need to be adjusted to account for `init` members. This will need to be included in the rule changes for non-mutating acess to `readonly` data.

The IL verification rules will need to be broken into two parts:

1. Allowing `init` members to set a `readonly` field.
2. Determining when an `init` member can be legally called.

The first is a simple adjustment to the existing rules. The IL verifier can be taught to recognize `init` members and from there it just needs to consider a `readonly` field to be settable on `this` in such a member.

The second rule is more complicated. In the simple case of object initializers the rule is straight forward. It should be legal to call `init` members when the result of a `new` expression is still on the stack. That is until the value has been stored in a local, array element or field or passed as an argument to another method it will still be legal to call `init` members. This ensures that once the result of the `new` expression is published to a named identifier (other than `this`) then it will no longer be legal to call `init` members.

The more complicated case though is when we mix `init` members, object initializers and `await`. That can cause the newly created object to be temporarily hoisted into a state machine and hence put into a field.

```
var student = new Student()
{
    Name = await SomeMethod()
};
```

Here the result of `new Student()` will be hoised into a state machine as a field before the set of `Name` occurs. The compiler will need to mark such hoisted fields in a way that the IL verifier understands they're not user accessible and hence doesn't violate the intended semantics of `init`.

init members

The `init` modifier could be extended to apply to all instance members. This would generalize the concept of `init` during object construction and allow types to declare helper methods that could partipate in the construction process to initialize `init` fields and properties.

Such members would have all the restricions that an `init` accessor does in this design. The need is questionable though and this can be safely added in a future version of the language in a compatible manner.

Generate three accessors

One potential implementation of `init` properties is to make `init` completely separate from `set`. That means that a property can potentially have three different accessors: `get`, `set`, and `init`.

This has the potential advantage of allowing the use of modreq to enforce correctness while maintaining binary compatibility. The implementation would roughly be the following:

1. An `init` accessor is always emitted if there is a `set`. When not defined by the developer it is simply a reference to `set`.
2. The set of a property in an object initializer will always use `init` if present but fall back to `set` if it's missing.

This means that a developer can always safely delete `init` from a property.

The downside of this design is that is only useful if `init` is **always** emitted when there is a `set`. The language can't know if `init` was deleted in the past, it has to assume it was and hence the `init` must always be emitted. That would cause a significant metadata expansion and is simply not worth the cost of the compatibility here.

Target-typed `new` expressions

18.12.2021 • 3 minutes to read

Summary

Do not require type specification for constructors when the type is known.

Motivation

Allow field initialization without duplicating the type.

```
Dictionary<string, List<int>> field = new() {
    { "item1", new() { 1, 2, 3 } }
};
```

Allow omitting the type when it can be inferred from usage.

```
XmlReader.Create(reader, new() { IgnoreWhitespace = true });
```

Instantiate an object without spelling out the type.

```
private readonly static object s_syncObj = new();
```

Specification

A new syntactic form, *target_typed_new* of the *object_creation_expression* is accepted in which the *type* is optional.

```
object_creation_expression
    : 'new' type '(' argument_list? ')' object_or_collection_initializer?
    | 'new' type object_or_collection_initializer
    | target_typed_new
    ;
target_typed_new
    : 'new' '(' argument_list? ')' object_or_collection_initializer?
    ;
```

A *target_typed_new* expression does not have a type. However, there is a new *object creation conversion* that is an implicit conversion from expression, that exists from a *target_typed_new* to every type.

Given a target type `T`, the type `T0` is `T`'s underlying type if `T` is an instance of `System.Nullable`. Otherwise `T0` is `T`. The meaning of a *target_typed_new* expression that is converted to the type `T` is the same as the meaning of a corresponding *object_creation_expression* that specifies `T0` as the type.

It is a compile-time error if a *target_typed_new* is used as an operand of a unary or binary operator, or if it is used where it is not subject to an *object creation conversion*.

> **Open Issue:** should we allow delegates and tuples as the target-type?

The above rules include delegates (a reference type) and tuples (a struct type). Although both types are

constructible, if the type is inferable, an anonymous function or a tuple literal can already be used.

```
(int a, int b) t = new(1, 2); // "new" is redundant
Action a = new(() => {}); // "new" is redundant

(int a, int b) t = new(); // OK; same as (0, 0)
Action a = new(); // no constructor found
```

Miscellaneous

The following are consequences of the specification:

- `throw new()` is allowed (the target type is `System.Exception`)
- Target-typed `new` is not allowed with binary operators.
- It is disallowed when there is no type to target: unary operators, collection of a `foreach`, in a `using`, in a deconstruction, in an `await` expression, as an anonymous type property (`new { Prop = new() }`), in a `lock` statement, in a `sizeof`, in a `fixed` statement, in a member access (`new().field`), in a dynamically dispatched operation (`someDynamic.Method(new())`), in a LINQ query, as the operand of the `is` operator, as the left operand of the `??` operator, ...
- It is also disallowed as a `ref`.
- The following kinds of types are not permitted as targets of the conversion
  - **Enum types:** `new()` will work (as `new Enum()` works to give the default value), but `new(1)` will not work as enum types do not have a constructor.
  - **Interface types:** This would work the same as the corresponding creation expression for COM types.
  - **Array types:** arrays need a special syntax to provide the length.
  - **dynamic:** we don't allow `new dynamic()`, so we don't allow `new()` with `dynamic` as a target type.
  - **tuples:** These have the same meaning as an object creation using the underlying type.
  - All the other types that are not permitted in the *object_creation_expression* are excluded as well, for instance, pointer types.

Drawbacks

There were some concerns with target-typed `new` creating new categories of breaking changes, but we already have that with `null` and `default`, and that has not been a significant problem.

Alternatives

Most of complaints about types being too long to duplicate in field initialization is about *type arguments* not the type itself, we could infer only type arguments like `new Dictionary(...)` (or similar) and infer type arguments locally from arguments or the collection initializer.

Questions

- Should we forbid usages in expression trees? (no)
- How the feature interacts with `dynamic` arguments? (no special treatment)
- How IntelliSense should work with `new()`? (only when there is a single target-type)

Design meetings

- LDM-2017-10-18
- LDM-2018-05-21
- LDM-2018-06-25

- LDM-2018-08-22
- LDM-2018-10-17
- LDM-2020-03-25

Module Initializers

18.12.2021 • 2 minutes to read

Summary

Although the .NET platform has a feature that directly supports writing initialization code for the assembly (technically, the module), it is not exposed in C#. This is a rather niche scenario, but once you run into it the solutions appear to be pretty painful. There are reports of a number of customers (inside and outside Microsoft) struggling with the problem, and there are no doubt more undocumented cases.

Motivation

- Enable libraries to do eager, one-time initialization when loaded, with minimal overhead and without the user needing to explicitly call anything
- One particular pain point of current `static` constructor approaches is that the runtime must do additional checks on usage of a type with a static constructor, in order to decide whether the static constructor needs to be run or not. This adds measurable overhead.
- Enable source generators to run some global initialization logic without the user needing to explicitly call anything

Detailed design

A method can be designated as a module initializer by decorating it with a `[ModuleInitializer]` attribute.

```
using System;
namespace System.Runtime.CompilerServices
{
    [AttributeUsage(AttributeTargets.Method, AllowMultiple = false)]
    public sealed class ModuleInitializerAttribute : Attribute { }
}
```

The attribute can be used like this:

```
using System.Runtime.CompilerServices;
class C
{
    [ModuleInitializer]
    internal static void M1()
    {
        // ...
    }
}
```

Some requirements are imposed on the method targeted with this attribute:

1. The method must be `static`.
2. The method must be parameterless.
3. The method must return `void`.
4. The method must not be generic or be contained in a generic type.
5. The method must be accessible from the containing module.
    - This means the method's effective accessibility must be `internal` or `public`.

- This also means the method cannot be a local function.

When one or more valid methods with this attribute are found in a compilation, the compiler will emit a module initializer which calls each of the attributed methods. The calls will be emitted in a reserved, but deterministic order.

Drawbacks

Why should we *not* do this?

- Perhaps the existing third-party tooling for "injecting" module initializers is sufficient for users who have been asking for this feature.

Design meetings

April 8th, 2020

Extending Partial Methods

18.12.2021 • 5 minutes to read

Summary

This proposal aims to remove all restrictions around the signatures of `partial` methods in C#. The goal being to expand the set of scenarios in which these methods can work with source generators as well as being a more general declaration form for C# methods.

See also the original partial methods specification.

Motivation

C# has limited support for developers splitting methods into declarations and definitions / implementations.

```
partial class C
{
    // The declaration of C.M
    partial void M(string message);
}

partial class C
{
    // The definition of C.M
    partial void M(string message) => Console.WriteLine(message);
}
```

One behavior of `partial` methods is that when the definition is absent then the language will simply erase any calls to the `partial` method. Essentially it behaves like a call to a `[Conditional]` method where the condition was evaluated to false.

```
partial class D
{
    partial void M(string message);

    void Example()
    {
        M(GetIt()); // Call to M and GetIt erased at compile time
    }

    string GetIt() => "Hello World";
}
```

The original motivation for this feature was source generation in the form of designer generated code. Users were constantly editing the generated code because they wanted to hook some aspect of the generated code. Most notably parts of the Windows Forms startup process, after components were initialized.

Editing the generated code was error prone because any action which caused the designer to regenerate the code would cause the user edit to be erased. The `partial` method feature eased this tension because it allowed designers to emit hooks in the form of `partial` methods.

Designers could emit hooks like `partial void OnComponentInit()` and developers could define declarations for them or not define them. In either case though the generated code would compile and developers who were interested in the process could hook in as needed.

This does mean that partial methods have several restrictions:

1. Must have a `void` return type.
2. Cannot have `out` parameters.
3. Cannot have any accessibility (implicitly `private`).

These restrictions exist because the language must be able to emit code when the call site is erased. Given they can be erased `private` is the only possible accessibility because the member can't be exposed in assembly metadata. These restrictions also serve to limit the set of scenarios in which `partial` methods can be applied.

The proposal here is to remove all of the existing restrictions around `partial` methods. Essentially let them have `out`, non-void return types or any type of accessibility. Such `partial` declarations would then have the added requirement that a definition must exist. That means the language does not have to consider the impact of erasing the call sites.

This would expand the set of generator scenarios that `partial` methods could participate in and hence link in nicely with our source generators feature. For example a regex could be defined using the following pattern:

```
[RegexGenerated("(dog|cat|fish)")]
partial bool IsPetMatch(string input);
```

This gives both the developer a simple declarative way of opting into generators as well as giving generators a very easy set of declarations to look through in the source code to drive their generated output.

Compare that with the difficulty that a generator would have hooking up the following snippet of code.

```
var regex = new RegularExpression("(dog|cat|fish)");
if (regex.IsMatch(someInput))
{

}
```

Given that the compiler doesn't allow generators to modify code hooking up this pattern would be pretty much impossible for generators. They would need to resort to reflection in the `IsMatch` implementation, or asking users to change their call sites to a new method + refactor the regex to pass the string literal as an argument. It's pretty messy.

Detailed Design

The language will change to allow `partial` methods to be annotated with an explicit accessibility modifier. This means they can be labeled as `private`, `public`, etc ...

When a `partial` method has an explicit accessibility modifier though the language will require that the declaration has a matching definition even when the accessibility is `private`:

```
partial class C
{
    // Okay because no definition is required here
    partial void M1();

    // Okay because M2 has a definition
    private partial void M2();

    // Error: partial method M3 must have a definition
    private partial void M3();
}

partial class C
{
    private partial void M2() { }
}
```

Further the language will remove all restrictions on what can appear on a `partial` method which has an explicit accessibility. Such declarations can contain non-void return types, `out` parameters, `extern` modifier, etc ...
These signatures will have the full expressivity of the C# language.

```
partial class D
{
    // Okay
    internal partial bool TryParse(string s, out int i);
}

partial class D
{
    internal partial bool TryParse(string s, out int i) { }
}
```

This explicitly allows for `partial` methods to participate in `overrides` and `interface` implementations:

```
interface IStudent
{
    string GetName();
}

partial class C : IStudent
{
    public virtual partial string GetName();
}

partial class C
{
    public virtual partial string GetName() => "Jarde";
}
```

The compiler will change the error it emits when a `partial` method contains an illegal element to essentially say:

> Cannot use `ref` on a `partial` method that lacks explicit accessibility

This will help point developers in the right direction when using this feature.

Restrictions:

- `partial` declarations with explicit accessibility must have a definition
- `partial` declarations and definition signatures must match on all method and parameter modifiers. The only

aspects which can differ are parameter names and attribute lists (this is not new but rather an existing requirement of `partial` methods).

Questions

partial on all members

Given that we're expanding `partial` to be more friendly to source generators should we also expand it to work on all class members? For example should we be able to declare `partial` constructors, operators, etc ...

Resolution The idea is sound but at this point in the C# 9 schedule we're trying to avoid unnecessary feature creep. Want to solve the immediate problem of expanding the feature to work with modern source generators.

Extending `partial` to support other members will be considered for the C# 10 release. Seems likely that we will consider this extension.

Use abstract instead of partial

The crux of this proposal is essentially ensuring that a declaration has a corresponding definition / implementation. Given that should we use `abstract` since it's already a language keyword that forces the developer to think about having an implementation?

Resolution There was a healthy discussion about this but eventually it was decided against. Yes the requirements are familiar but the concepts are significantly different. Could easily lead the developer to believe they were creating virtual slots when they were not doing so.

Static anonymous functions

18.12.2021 • 2 minutes to read

Summary

Allow a 'static' modifier on lambdas and anonymous methods, which disallows capture of locals or instance state from containing scopes.

Motivation

Avoid unintentionally capturing state from the enclosing context, which can result in unexpected retention of captured objects or unexpected additional allocations.

Detailed design

A lambda or anonymous method may have a `static` modifier. The `static` modifier indicates that the lambda or anonymous method is a *static anonymous function*.

A *static anonymous function* cannot capture state from the enclosing scope. As a result, locals, parameters, and `this` from the enclosing scope are not available within a *static anonymous function*.

A *static anonymous function* cannot reference instance members from an implicit or explicit `this` or `base` reference.

A *static anonymous function* may reference `static` members from the enclosing scope.

A *static anonymous function* may reference `constant` definitions from the enclosing scope.

`nameof()` in a *static anonymous function* may reference locals, parameters, or `this` or `base` from the enclosing scope.

Accessibility rules for `private` members in the enclosing scope are the same for `static` and non-`static` anonymous functions.

No guarantee is made as to whether a *static anonymous function* definition is emitted as a `static` method in metadata. This is left up to the compiler implementation to optimize.

A non-`static` local function or anonymous function can capture state from an enclosing *static anonymous function* but cannot capture state outside the enclosing *static anonymous function*.

Removing the `static` modifier from an anonymous function in a valid program does not change the meaning of the program.

Target-Typed Conditional Expression

18.12.2021 • 3 minutes to read

Conditional Expression Conversion

For a conditional expression `c ? e1 : e2`, when

1. there is no common type for `e1` and `e2`, or
2. for which a common type exists but one of the expressions `e1` or `e2` has no implicit conversion to that type

we define a new implicit *conditional expression conversion* that permits an implicit conversion from the conditional expression to any type `T` for which there is a conversion-from-expression from `e1` to `T` and also from `e2` to `T`. It is an error if a conditional expression neither has a common type between `e1` and `e2` nor is subject to a *conditional expression conversion*.

Better Conversion from Expression

We change

> **Better conversion from expression**
> Given an implicit conversion `C1` that converts from an expression `E` to a type `T1`, and an implicit conversion `C2` that converts from an expression `E` to a type `T2`, `C1` is a ***better conversion*** than `C2` if `E` does not exactly match `T2` and at least one of the following holds:
>
> - `E` exactly matches `T1` (Exactly matching Expression)
> - `T1` is a better conversion target than `T2` (Better conversion target)

to

> **Better conversion from expression**
> Given an implicit conversion `C1` that converts from an expression `E` to a type `T1`, and an implicit conversion `C2` that converts from an expression `E` to a type `T2`, `C1` is a ***better conversion*** than `C2` if `E` does not exactly match `T2` and at least one of the following holds:
>
> - `E` exactly matches `T1` (Exactly matching Expression)
> - `C1` **is not a *conditional expression conversion* and** `C2` **is a *conditional expression conversion*.**
> - `T1` is a better conversion target than `T2` (Better conversion target) **and either** `C1` **and** `C2` **are both *conditional expression conversions* or neither is a *conditional expression conversion*.**

Cast Expression

The current C# language specification says

> A *cast_expression* of the form `(T)E`, where `T` is a *type* and `E` is a *unary_expression*, performs an explicit conversion (Explicit conversions) of the value of `E` to type `T`.

In the presence of the *conditional expression conversion* there may be more than one possible conversion from `E` to `T`. With the addition of *conditional expression conversion*, we prefer any other conversion to a *conditional expression conversion*, and use the *conditional expression conversion* only as a last resort.

Design Notes

The reason for the change to *Better conversion from expression* is to handle a case such as this:

```
M(b ? 1 : 2);

void M(short);
void M(long);
```

This approach does have two small downsides. First, it is not quite the same as the switch expression:

```
M(b ? 1 : 2); // calls M(long)
M(b switch { true => 1, false => 2 }); // calls M(short)
```

This is still a breaking change, but its scope is less likely to affect real programs:

```
M(b ? 1 : 2, 1); // calls M(long, long) without this feature; ambiguous with this feature.

M(short, short);
M(long, long);
```

This becomes ambiguous because the conversion to `long` is better for the first argument (because it does not use the *conditional expression conversion*), but the conversion to `short` is better for the second argument (because `short` is a *better conversion target* than `long`). This breaking change seems less serious because it does not silently change the behavior of an existing program.

The reason for the notes on the cast expression is to handle a case such as this:

```
_ = (short)(b ? 1 : 2);
```

This program currently uses the explicit conversion from `int` to `short`, and we want to preserve the current language meaning of this program. The change would be unobservable at runtime, but with the following program the change would be observable:

```
_ = (A)(b ? c : d);
```

where `c` is of type `C`, `d` is of type `D`, and there is an implicit user-defined conversion from `C` to `D`, and an implicit user-defined conversion from `D` to `A`, and an implicit user-defined conversion from `C` to `A`. If this code is compiled before C# 9.0, when `b` is true we convert from `c` to `D` then to `A`. If we use the *conditional expression conversion*, then when `b` is true we convert from `c` to `A` directly, which executes a different sequence of user code. Therefore we treat the *conditional expression conversion* as a last resort in a cast, to preserve existing behavior.

Covariant returns

18.12.2021 • 10 minutes to read

Summary

Support *covariant return types*. Specifically, permit the override of a method to declare a more derived return type than the method it overrides, and similarly to permit the override of a read-only property to declare a more derived type. Override declarations appearing in more derived types would be required to provide a return type at least as specific as that appearing in overrides in its base types. Callers of the method or property would statically receive the more refined return type from an invocation.

Motivation

It is a common pattern in code that different method names have to be invented to work around the language constraint that overrides must return the same type as the overridden method.

This would be useful in the factory pattern. For example, in the Roslyn code base we would have

```
class Compilation ...
{
    public virtual Compilation WithOptions(Options options)...
}
```

```
class CSharpCompilation : Compilation
{
    public override CSharpCompilation WithOptions(Options options)...
}
```

Detailed design

This is a specification for covariant return types in C#. Our intent is to permit the override of a method to return a more derived return type than the method it overrides, and similarly to permit the override of a read-only property to return a more derived return type. Callers of the method or property would statically receive the more refined return type from an invocation, and overrides appearing in more derived types would be required to provide a return type at least as specific as that appearing in overrides in its base types.

Class Method Override

The existing constraint on class override methods

> - The override method and the overridden base method have the same return type.

is modified to

> - The override method must have a return type that is convertible by an identity conversion or (if the method has a value return - not a ref return) implicit reference conversion to the return type of the overridden base method.

And the following additional requirements are appended to that list:

> - The override method must have a return type that is convertible by an identity conversion or (if the method has a value return - not a ref return) implicit reference conversion to the return type of every override of the overridden base method that is declared in a (direct or indirect) base type of the override method.
> - The override method's return type must be at least as accessible as the override method (Accessibility domains).

This constraint permits an override method in a `private` class to have a `private` return type. However it requires a `public` override method in a `public` type to have a `public` return type.

Class Property and Indexer Override

The existing constraint on class override properties

> An overriding property declaration shall specify the exact same accessibility modifiers and name as the inherited property, and there shall be an identity conversion ~~between the type of the overriding and the inherited property~~. If the inherited property has only a single accessor (i.e., if the inherited property is read-only or write-only), the overriding property shall include only that accessor. If the inherited property includes both accessors (i.e., if the inherited property is read-write), the overriding property can include either a single accessor or both accessors.

is modified to

> An overriding property declaration shall specify the exact same accessibility modifiers and name as the inherited property, and there shall be an identity conversion **or (if the inherited property is read-only and has a value return - not a ref return) implicit reference conversion from the type of the overriding property to the type of the inherited property**. If the inherited property has only a single accessor (i.e., if the inherited property is read-only or write-only), the overriding property shall include only that accessor. If the inherited property includes both accessors (i.e., if the inherited property is read-write), the overriding property can include either a single accessor or both accessors. **The overriding property's type must be at least as accessible as the overriding property (Accessibility domains).**

The remainder of the draft specification below proposes a further extension to covariant returns of interface methods to be considered later.

Interface Method, Property, and Indexer Override

Adding to the kinds of members that are permitted in an interface with the addition of the DIM feature in C# 8.0, we further add support for `override` members along with covariant returns. These follow the rules of `override` members as specified for classes, with the following differences:

The following text in classes:

> The method overridden by an override declaration is known as the ***overridden base method***. For an override method `M` declared in a class `C`, the overridden base method is determined by examining each base class of `C`, starting with the direct base class of `C` and continuing with each successive direct base class, until in a given base class type at least one accessible method is located which has the same signature as `M` after substitution of type arguments.

is given the corresponding specification for interfaces:

> The method overridden by an override declaration is known as the ***overridden base method***. For an override method `M` declared in an interface `I`, the overridden base method is determined by examining each direct or indirect base interface of `I`, collecting the set of interfaces declaring an accessible method

> which has the same signature as `M` after substitution of type arguments. If this set of interfaces has a *most derived type*, to which there is an identity or implicit reference conversion from every type in this set, and that type contains a unique such method declaration, then that is the *overridden base method*.

We similarly permit `override` properties and indexers in interfaces as specified for classes in *15.7.6 Virtual, sealed, override, and abstract accessors*.

Name Lookup

Name lookup in the presence of class `override` declarations currently modify the result of name lookup by imposing on the found member details from the most derived `override` declaration in the class hierarchy starting from the type of the identifier's qualifier (or `this` when there is no qualifier). For example, in *12.6.2.2 Corresponding parameters* we have

> For virtual methods and indexers defined in classes, the parameter list is picked from the first declaration or override of the function member found when starting with the static type of the receiver, and searching through its base classes.

to this we add

> For virtual methods and indexers defined in interfaces, the parameter list is picked from the declaration or override of the function member found in the most derived type among those types containing the declaration of override of the function member. It is a compile-time error if no unique such type exists.

For the result type of a property or indexer access, the existing text

> - If I identifies an instance property, then the result is a property access with an associated instance expression of E and an associated type that is the type of the property. If T is a class type, the associated type is picked from the first declaration or override of the property found when starting with T, and searching through its base classes.

is augmented with

> If T is an interface type, the associated type is picked from the declaration or override of the property found in the most derived of T or its direct or indirect base interfaces. It is a compile-time error if no unique such type exists.

A similar change should be made in *12.7.7.3 Indexer access*

In *12.7.6 Invocation expressions* we augment the existing text

> - Otherwise, the result is a value, with an associated type of the return type of the method or delegate. If the invocation is of an instance method, and the receiver is of a class type T, the associated type is picked from the first declaration or override of the method found when starting with T and searching through its base classes.

with

> If the invocation is of an instance method, and the receiver is of an interface type T, the associated type is picked from the declaration or override of the method found in the most derived interface from among T and its direct and indirect base interfaces. It is a compile-time error if no unique such type exists.

Implicit Interface Implementations

This section of the specification

> For purposes of interface mapping, a class member `A` matches an interface member `B` when:
>
> - `A` and `B` are methods, and the name, type, and formal parameter lists of `A` and `B` are identical.
> - `A` and `B` are properties, the name and type of `A` and `B` are identical, and `A` has the same accessors as `B` (`A` is permitted to have additional accessors if it is not an explicit interface member implementation).
> - `A` and `B` are events, and the name and type of `A` and `B` are identical.
> - `A` and `B` are indexers, the type and formal parameter lists of `A` and `B` are identical, and `A` has the same accessors as `B` (`A` is permitted to have additional accessors if it is not an explicit interface member implementation).

is modified as follows:

> For purposes of interface mapping, a class member `A` matches an interface member `B` when:
>
> - `A` and `B` are methods, and the name and formal parameter lists of `A` and `B` are identical, and the return type of `A` is convertible to the return type of `B` via an identity of implicit reference convertion to the return type of `B`.
> - `A` and `B` are properties, the name of `A` and `B` are identical, `A` has the same accessors as `B` (`A` is permitted to have additional accessors if it is not an explicit interface member implementation), and the type of `A` is convertible to the return type of `B` via an identity conversion or, if `A` is a readonly property, an implicit reference conversion.
> - `A` and `B` are events, and the name and type of `A` and `B` are identical.
> - `A` and `B` are indexers, the formal parameter lists of `A` and `B` are identical, `A` has the same accessors as `B` (`A` is permitted to have additional accessors if it is not an explicit interface member implementation), and the type of `A` is convertible to the return type of `B` via an identity conversion or, if `A` is a readonly indexer, an implicit reference conversion.

This is technically a breaking change, as the program below prints "C1.M" today, but would print "C2.M" under the proposed revision.

```
using System;

interface I1 { object M(); }
class C1 : I1 { public object M() { return "C1.M"; } }
class C2 : C1, I1 { public new string M() { return "C2.M"; } }
class Program
{
    static void Main()
    {
        I1 i = new C2();
        Console.WriteLine(i.M());
    }
}
```

Due to this breaking change, we might consider not supporting covariant return types on implicit implementations.

Constraints on Interface Implementation

We will need a rule that an explicit interface implementation must declare a return type no less derived than the return type declared in any override in its base interfaces.

API Compatibility Implications

TBD

Open Issues

The specification does not say how the caller gets the more refined return type. Presumably that would be done in a way similar to the way that callers get the most derived override's parameter specifications.

If we have the following interfaces:

```
interface I1 { I1 M(); }
interface I2 { I2 M(); }
interface I3: I1, I2 { override I3 M(); }
```

Note that in `I3`, the methods `I1.M()` and `I2.M()` have been "merged". When implementing `I3`, it is necessary to implement them both together.

Generally, we require an explicit implementation to refer to the original method. The question is, in a class

```
class C : I1, I2, I3
{
    C IN.M();
}
```

What does that mean here? What should *N* be?

I suggest that we permit implementing either `I1.M` or `I2.M` (but not both), and treat that as an implementation of both.

Drawbacks

- [ ] Every language change must pay for itself.
- [ ] We should ensure that the performance is reasonable, even in the case of deep inheritance hierarchies
- [ ] We should ensure that artifacts of the translation strategy do not affect language semantics, even when consuming new IL from old compilers.

Alternatives

We could relax the language rules slightly to allow, in source,

```
abstract class Cloneable
{
    public abstract Cloneable Clone();
}

class Digit : Cloneable
{
    public override Cloneable Clone()
    {
        return this.Clone();
    }

    public new Digit Clone() // Error: 'Digit' already defines a member called 'Clone' with the same
parameter types
    {
        return this;
    }
}
```

Unresolved questions

- [ ] How will APIs that have been compiled to use this feature work in older versions of the language?

Design meetings

- some discussion at https://github.com/dotnet/roslyn/issues/357.
- https://github.com/dotnet/csharplang/blob/master/meetings/2020/LDM-2020-01-08.md
- Offline discussion toward a decision to support overriding of class methods only in C# 9.0.

Extension `GetEnumerator` support for `foreach` loops.

18.12.2021 • 5 minutes to read

Summary

Allow foreach loops to recognize an extension method GetEnumerator method that otherwise satisfies the foreach pattern, and loop over the expression when it would otherwise be an error.

Motivation

This will bring foreach inline with how other features in C# are implemented, including async and pattern-based deconstruction.

Detailed design

The spec change is relatively straightforward. We modify `The foreach statement` section to this text:

> The compile-time processing of a foreach statement first determines the ***collection type***, ***enumerator type*** and ***element type*** of the expression. This determination proceeds as follows:
>
> - If the type `X` of *expression* is an array type then there is an implicit reference conversion from `X` to the `IEnumerable` interface (since `System.Array` implements this interface). The ***collection type*** is the `IEnumerable` interface, the ***enumerator type*** is the `IEnumerator` interface and the ***element type*** is the element type of the array type `X`.
> - If the type `X` of *expression* is `dynamic` then there is an implicit conversion from *expression* to the `IEnumerable` interface (Implicit dynamic conversions). The ***collection type*** is the `IEnumerable` interface and the ***enumerator type*** is the `IEnumerator` interface. If the `var` identifier is given as the *local_variable_type* then the ***element type*** is `dynamic`, otherwise it is `object`.
> - Otherwise, determine whether the type `X` has an appropriate `GetEnumerator` method:
>   - Perform member lookup on the type `X` with identifier `GetEnumerator` and no type arguments. If the member lookup does not produce a match, or it produces an ambiguity, or produces a match that is not a method group, check for an enumerable interface as described below. It is recommended that a warning be issued if member lookup produces anything except a method group or no match.
>   - Perform overload resolution using the resulting method group and an empty argument list. If overload resolution results in no applicable methods, results in an ambiguity, or results in a single best method but that method is either static or not public, check for an enumerable interface as described below. It is recommended that a warning be issued if overload resolution produces anything except an unambiguous public instance method or no applicable methods.
>   - If the return type `E` of the `GetEnumerator` method is not a class, struct or interface type, an error is produced and no further steps are taken.
>   - Member lookup is performed on `E` with the identifier `Current` and no type arguments. If the member lookup produces no match, the result is an error, or the result is anything except a public instance property that permits reading, an error is produced and no further steps are taken.
>   - Member lookup is performed on `E` with the identifier `MoveNext` and no type arguments. If the

member lookup produces no match, the result is an error, or the result is anything except a method group, an error is produced and no further steps are taken.
    - Overload resolution is performed on the method group with an empty argument list. If overload resolution results in no applicable methods, results in an ambiguity, or results in a single best method but that method is either static or not public, or its return type is not `bool`, an error is produced and no further steps are taken.
    - The ***collection type*** is `X`, the ***enumerator type*** is `E`, and the ***element type*** is the type of the `Current` property.
- Otherwise, check for an enumerable interface:
    - If among all the types `Ti` for which there is an implicit conversion from `X` to `IEnumerable<Ti>`, there is a unique type `T` such that `T` is not `dynamic` and for all the other `Ti` there is an implicit conversion from `IEnumerable<T>` to `IEnumerable<Ti>`, then the ***collection type*** is the interface `IEnumerable<T>`, the ***enumerator type*** is the interface `IEnumerator<T>`, and the ***element type*** is `T`.
    - Otherwise, if there is more than one such type `T`, then an error is produced and no further steps are taken.
    - Otherwise, if there is an implicit conversion from `X` to the `System.Collections.IEnumerable` interface, then the ***collection type*** is this interface, the ***enumerator type*** is the interface `System.Collections.IEnumerator`, and the ***element type*** is `object`.
- Otherwise, determine whether the type 'X' has an appropriate `GetEnumerator` extension method:
    - Perform extension method lookup on the type `X` with identifier `GetEnumerator`. If the member lookup does not produce a match, or it produces an ambiguity, or produces a match which is not a method group, an error is produced and no further steps are taken. It is recommended that a warning be issues if member lookup produces anything except a method group or no match.
    - Perform overload resolution using the resulting method group and a single argument of type `X`. If overload resolution produces no applicable methods, results in an ambiguity, or results in a single best method but that method is not accessible, an error is produced an no further steps are taken.
        - This resolution permits the first argument to be passed by ref if `X` is a struct type, and the ref kind is `in`.
    - If the return type `E` of the `GetEnumerator` method is not a class, struct or interface type, an error is produced and no further steps are taken.
    - Member lookup is performed on `E` with the identifier `Current` and no type arguments. If the member lookup produces no match, the result is an error, or the result is anything except a public instance property that permits reading, an error is produced and no further steps are taken.
    - Member lookup is performed on `E` with the identifier `MoveNext` and no type arguments. If the member lookup produces no match, the result is an error, or the result is anything except a method group, an error is produced and no further steps are taken.
    - Overload resolution is performed on the method group with an empty argument list. If overload resolution results in no applicable methods, results in an ambiguity, or results in a single best method but that method is either static or not public, or its return type is not `bool`, an error is produced and no further steps are taken.
    - The ***collection type*** is `X`, the ***enumerator type*** is `E`, and the ***element type*** is the type of the `Current` property.
- Otherwise, an error is produced and no further steps are taken.

For `await foreach`, the rules are similarly modified. The only change that is required to that spec is removing the `Extension methods do not contribute.` line from the description, as the rest of that spec is based on the

above rules with different names substituted for the pattern methods.

Drawbacks

Every change adds additional complexity to the language, and this potentially allows things that weren't designed to be `foreach`ed to be `foreach`ed, like `Range`.

Alternatives

Doing nothing.

Unresolved questions

None at this point.

Lambda discard parameters

18.12.2021 • 2 minutes to read

Summary

Allow discards (`_`) to be used as parameters of lambdas and anonymous methods. For example:

- lambdas: `(_, _) => 0`, `(int _, int _) => 0`
- anonymous methods: `delegate(int _, int _) { return 0; }`

Motivation

Unused parameters do not need to be named. The intent of discards is clear, i.e. they are unused/discarded.

Detailed design

Method parameters In the parameter list of a lambda or anonymous method with more than one parameter named `_`, such parameters are discard parameters. Note: if a single parameter is named `_` then it is a regular parameter for backwards compatibility reasons.

Discard parameters do not introduce any names to any scopes. Note this implies they do not cause any `_` (underscore) names to be hidden.

Simple names If `K` is zero and the *simple_name* appears within a *block* and if the *block*'s (or an enclosing *block*'s) local variable declaration space (Declarations) contains a local variable, parameter (with the exception of discard parameters) or constant with name `I`, then the *simple_name* refers to that local variable, parameter or constant and is classified as a variable or value.

Scopes With the exception of discard parameters, the scope of a parameter declared in a *lambda_expression* (Anonymous function expressions) is the *anonymous_function_body* of that *lambda_expression* With the exception of discard parameters, the scope of a parameter declared in an *anonymous_method_expression* (Anonymous function expressions) is the *block* of that *anonymous_method_expression*.

Related spec sections

- Corresponding parameters

Attributes on local functions

18.12.2021 • 2 minutes to read

Attributes

Local function declarations are now permitted to have attributes. Parameters and type parameters on local functions are also allowed to have attributes.

Attributes with a specified meaning when applied to a method, its parameters, or its type parameters will have the same meaning when applied to a local function, its parameters, or its type parameters, respectively.

A local function can be made conditional in the same sense as a conditional method by decorating it with a `[ConditionalAttribute]`. A conditional local function must also be `static`. All restrictions on conditional methods also apply to conditional local functions, including that the return type must be `void`.

Extern

The `extern` modifier is now permitted on local functions. This makes the local function external in the same sense as an external method.

Similarly to an external method, the *local-function-body* of an external local function must be a semicolon. A semicolon *local-function-body* is only permitted on an external local function.

An external local function must also be `static`.

Syntax

The local functions grammar is modified as follows:

```
local-function-header
    : attributes? local-function-modifiers? return-type identifier type-parameter-list?
        ( formal-parameter-list? ) type-parameter-constraints-clauses
    ;

local-function-modifiers
    : (async | unsafe | static | extern)*
    ;

local-function-body
    : block
    | arrow-expression-body
    | ';'
    ;
```

Native-sized integers

18.12.2021 • 10 minutes to read

Summary

Language support for a native-sized signed and unsigned integer types.

The motivation is for interop scenarios and for low-level libraries.

Design

The identifiers `nint` and `nuint` are new contextual keywords that represent native signed and unsigned integer types. The identifiers are only treated as keywords when name lookup does not find a viable result at that program location.

```
nint x = 3;
string y = nameof(nuint);
_ = nint.Equals(x, 3);
```

The types `nint` and `nuint` are represented by the underlying types `System.IntPtr` and `System.UIntPtr` with compiler surfacing additional conversions and operations for those types as native ints.

Constants

Constant expressions may be of type `nint` or `nuint`. There is no direct syntax for native int literals. Implicit or explicit casts of other integral constant values can be used instead: `const nint i = (nint)42;`.

`nint` constants are in the range [`int.MinValue`, `int.MaxValue`].

`nuint` constants are in the range [`uint.MinValue`, `uint.MaxValue`].

There are no `MinValue` or `MaxValue` fields on `nint` or `nuint` because, other than `nuint.MinValue`, those values cannot be emitted as constants.

Constant folding is supported for all unary operators { `+`, `-`, `~` } and binary operators { `+`, `-`, `*`, `/`, `%`, `==`, `!=`, `<`, `<=`, `>`, `>=`, `&`, `|`, `^`, `<<`, `>>` }. Constant folding operations are evaluated with `Int32` and `UInt32` operands rather than native ints for consistent behavior regardless of compiler platform. If the operation results in a constant value in 32-bits, constant folding is performed at compile-time. Otherwise the operation is executed at runtime and not considered a constant.

Conversions

There is an identity conversion between `nint` and `IntPtr`, and between `nuint` and `UIntPtr`. There is an identity conversion between compound types that differ by native ints and underlying types only: arrays, `Nullable<>`, constructed types, and tuples.

The tables below cover the conversions between special types. (The IL for each conversion includes the variants for `unchecked` and `checked` contexts if different.)

OPERAND	TARGET	CONVERSION	IL
`object`	`nint`	Unboxing	`unbox`

OPERAND	TARGET	CONVERSION	IL
`void*`	`nint`	PointerToVoid	`conv.i`
`sbyte`	`nint`	ImplicitNumeric	`conv.i`
`byte`	`nint`	ImplicitNumeric	`conv.u`
`short`	`nint`	ImplicitNumeric	`conv.i`
`ushort`	`nint`	ImplicitNumeric	`conv.u`
`int`	`nint`	ImplicitNumeric	`conv.i`
`uint`	`nint`	ExplicitNumeric	`conv.u` / `conv.ovf.u`
`long`	`nint`	ExplicitNumeric	`conv.i` / `conv.ovf.i`
`ulong`	`nint`	ExplicitNumeric	`conv.i` / `conv.ovf.i`
`char`	`nint`	ImplicitNumeric	`conv.i`
`float`	`nint`	ExplicitNumeric	`conv.i` / `conv.ovf.i`
`double`	`nint`	ExplicitNumeric	`conv.i` / `conv.ovf.i`
`decimal`	`nint`	ExplicitNumeric	`long decimal.op_Explicit(decimal) conv.i` / `... conv.ovf.i`
`IntPtr`	`nint`	Identity	
`UIntPtr`	`nint`	None	
`object`	`nuint`	Unboxing	`unbox`
`void*`	`nuint`	PointerToVoid	`conv.u`
`sbyte`	`nuint`	ExplicitNumeric	`conv.u` / `conv.ovf.u`
`byte`	`nuint`	ImplicitNumeric	`conv.u`
`short`	`nuint`	ExplicitNumeric	`conv.u` / `conv.ovf.u`
`ushort`	`nuint`	ImplicitNumeric	`conv.u`
`int`	`nuint`	ExplicitNumeric	`conv.u` / `conv.ovf.u`
`uint`	`nuint`	ImplicitNumeric	`conv.u`

OPERAND	TARGET	CONVERSION	IL
`long`	`nuint`	ExplicitNumeric	`conv.u` / `conv.ovf.u`
`ulong`	`nuint`	ExplicitNumeric	`conv.u` / `conv.ovf.u`
`char`	`nuint`	ImplicitNumeric	`conv.u`
`float`	`nuint`	ExplicitNumeric	`conv.u` / `conv.ovf.u`
`double`	`nuint`	ExplicitNumeric	`conv.u` / `conv.ovf.u`
`decimal`	`nuint`	ExplicitNumeric	`ulong decimal.op_Explicit(decimal) conv.u` / `... conv.ovf.u.un`
`IntPtr`	`nuint`	None	
`UIntPtr`	`nuint`	Identity	
Enumeration	`nint`	ExplicitEnumeration	
Enumeration	`nuint`	ExplicitEnumeration	

OPERAND	TARGET	CONVERSION	IL
`nint`	`object`	Boxing	`box`
`nint`	`void*`	PointerToVoid	`conv.i`
`nint`	`nuint`	ExplicitNumeric	`conv.u` / `conv.ovf.u`
`nint`	`sbyte`	ExplicitNumeric	`conv.i1` / `conv.ovf.i1`
`nint`	`byte`	ExplicitNumeric	`conv.u1` / `conv.ovf.u1`
`nint`	`short`	ExplicitNumeric	`conv.i2` / `conv.ovf.i2`
`nint`	`ushort`	ExplicitNumeric	`conv.u2` / `conv.ovf.u2`
`nint`	`int`	ExplicitNumeric	`conv.i4` / `conv.ovf.i4`
`nint`	`uint`	ExplicitNumeric	`conv.u4` / `conv.ovf.u4`
`nint`	`long`	ImplicitNumeric	`conv.i8` / `conv.ovf.i8`
`nint`	`ulong`	ExplicitNumeric	`conv.i8` / `conv.ovf.i8`
`nint`	`char`	ExplicitNumeric	`conv.u2` / `conv.ovf.u2`

OPERAND	TARGET	CONVERSION	IL
`nint`	`float`	ImplicitNumeric	`conv.r4`
`nint`	`double`	ImplicitNumeric	`conv.r8`
`nint`	`decimal`	ImplicitNumeric	`conv.i8 decimal decimal.op_Implicit(long)`
`nint`	`IntPtr`	Identity	
`nint`	`UIntPtr`	None	
`nint`	Enumeration	ExplicitEnumeration	
`nuint`	`object`	Boxing	`box`
`nuint`	`void*`	PointerToVoid	`conv.u`
`nuint`	`nint`	ExplicitNumeric	`conv.i` / `conv.ovf.i`
`nuint`	`sbyte`	ExplicitNumeric	`conv.i1` / `conv.ovf.i1`
`nuint`	`byte`	ExplicitNumeric	`conv.u1` / `conv.ovf.u1`
`nuint`	`short`	ExplicitNumeric	`conv.i2` / `conv.ovf.i2`
`nuint`	`ushort`	ExplicitNumeric	`conv.u2` / `conv.ovf.u2`
`nuint`	`int`	ExplicitNumeric	`conv.i4` / `conv.ovf.i4`
`nuint`	`uint`	ExplicitNumeric	`conv.u4` / `conv.ovf.u4`
`nuint`	`long`	ExplicitNumeric	`conv.i8` / `conv.ovf.i8`
`nuint`	`ulong`	ImplicitNumeric	`conv.u8` / `conv.ovf.u8`
`nuint`	`char`	ExplicitNumeric	`conv.u2` / `conv.ovf.u2.un`
`nuint`	`float`	ImplicitNumeric	`conv.r.un conv.r4`
`nuint`	`double`	ImplicitNumeric	`conv.r.un conv.r8`
`nuint`	`decimal`	ImplicitNumeric	`conv.u8 decimal decimal.op_Implicit(ulong)`
`nuint`	`IntPtr`	None	
`nuint`	`UIntPtr`	Identity	

OPERAND	TARGET	CONVERSION	IL
`nuint`	Enumeration	ExplicitEnumeration	

Conversion from `A` to `Nullable<B>` is:

- an implicit nullable conversion if there is an identity conversion or implicit conversion from `A` to `B`;
- an explicit nullable conversion if there is an explicit conversion from `A` to `B`;
- otherwise invalid.

Conversion from `Nullable<A>` to `B` is:

- an explicit nullable conversion if there is an identity conversion or implicit or explicit numeric conversion from `A` to `B`;
- otherwise invalid.

Conversion from `Nullable<A>` to `Nullable<B>` is:

- an identity conversion if there is an identity conversion from `A` to `B`;
- an explicit nullable conversion if there is an implicit or explicit numeric conversion from `A` to `B`;
- otherwise invalid.

Operators

The predefined operators are as follows. These operators are considered during overload resolution based on normal rules for implicit conversions *if at least one of the operands is of type* `nint` *or* `nuint`.

(The IL for each operator includes the variants for `unchecked` and `checked` contexts if different.)

UNARY	OPERATOR SIGNATURE	IL
`+`	`nint operator +(nint value)`	`nop`
`+`	`nuint operator +(nuint value)`	`nop`
`-`	`nint operator -(nint value)`	`neg`
`~`	`nint operator ~(nint value)`	`not`
`~`	`nuint operator ~(nuint value)`	`not`

BINARY	OPERATOR SIGNATURE	IL
`+`	`nint operator +(nint left, nint right)`	`add` / `add.ovf`
`+`	`nuint operator +(nuint left, nuint right)`	`add` / `add.ovf.un`
`-`	`nint operator -(nint left, nint right)`	`sub` / `sub.ovf`
`-`	`nuint operator -(nuint left, nuint right)`	`sub` / `sub.ovf.un`

BINARY	OPERATOR SIGNATURE	IL
`*`	`nint operator *(nint left, nint right)`	`mul` / `mul.ovf`
`*`	`nuint operator *(nuint left, nuint right)`	`mul` / `mul.ovf.un`
`/`	`nint operator /(nint left, nint right)`	`div`
`/`	`nuint operator /(nuint left, nuint right)`	`div.un`
`%`	`nint operator %(nint left, nint right)`	`rem`
`%`	`nuint operator %(nuint left, nuint right)`	`rem.un`
`==`	`bool operator ==(nint left, nint right)`	`beq` / `ceq`
`==`	`bool operator ==(nuint left, nuint right)`	`beq` / `ceq`
`!=`	`bool operator !=(nint left, nint right)`	`bne`
`!=`	`bool operator !=(nuint left, nuint right)`	`bne`
`<`	`bool operator <(nint left, nint right)`	`blt` / `clt`
`<`	`bool operator <(nuint left, nuint right)`	`blt.un` / `clt.un`
`<=`	`bool operator <=(nint left, nint right)`	`ble`
`<=`	`bool operator <=(nuint left, nuint right)`	`ble.un`
`>`	`bool operator >(nint left, nint right)`	`bgt` / `cgt`
`>`	`bool operator >(nuint left, nuint right)`	`bgt.un` / `cgt.un`
`>=`	`bool operator >=(nint left, nint right)`	`bge`
`>=`	`bool operator >=(nuint left, nuint right)`	`bge.un`
`&`	`nint operator &(nint left, nint right)`	`and`

BINARY	OPERATOR SIGNATURE	IL
`&`	`nuint operator &(nuint left, nuint right)`	`and`
`|`	`nint operator |(nint left, nint right)`	`or`
`|`	`nuint operator |(nuint left, nuint right)`	`or`
`^`	`nint operator ^(nint left, nint right)`	`xor`
`^`	`nuint operator ^(nuint left, nuint right)`	`xor`
`<<`	`nint operator <<(nint left, int right)`	`shl`
`<<`	`nuint operator <<(nuint left, int right)`	`shl`
`>>`	`nint operator >>(nint left, int right)`	`shr`
`>>`	`nuint operator >>(nuint left, int right)`	`shr.un`

For some binary operators, the IL operators support additional operand types (see ECMA-335 III.1.5 Operand type table). But the set of operand types supported by C# is limited for simplicity and for consistency with existing operators in the language.

Lifted versions of the operators, where the arguments and return types are `nint?` and `nuint?`, are supported.

Compound assignment operations `x op= y` where `x` or `y` are native ints follow the same rules as with other primitive types with pre-defined operators. Specifically the expression is bound as `x = (T)(x op y)` where `T` is the type of `x` and where `x` is only evaluated once.

The shift operators should mask the number of bits to shift - to 5 bits if `sizeof(nint)` is 4, and to 6 bits if `sizeof(nint)` is 8. (see shift operators in C# spec).

The C#9 compiler will report errors binding to predefined native integer operators when compiling with an earlier language version, but will allow use of predefined conversions to and from native integers.

`csc -langversion:9 -t:library A.cs`

```
public class A
{
    public static nint F;
}
```

`csc -langversion:8 -r:A.dll B.cs`

```
class B : A
{
    static void Main()
    {
        F = F + 1; // error: nint operator+ not available with -langversion:8
        F = (System.IntPtr)F + 1; // ok
    }
}
```

Dynamic

The conversions and operators are synthesized by the compiler and are not part of the underlying `IntPtr` and `UIntPtr` types. As a result those conversions and operators *are not available* from the runtime binder for `dynamic`.

```
nint x = 2;
nint y = x + x; // ok
dynamic d = x;
nint z = d + x; // RuntimeBinderException: '+' cannot be applied 'System.IntPtr' and 'System.IntPtr'
```

Type members

The only constructor for `nint` or `nuint` is the parameter-less constructor.

The following members of `System.IntPtr` and `System.UIntPtr` *are explicitly excluded* from `nint` or `nuint`:

```
// constructors
// arithmetic operators
// implicit and explicit conversions
public static readonly IntPtr Zero; // use 0 instead
public static int Size { get; }     // use sizeof() instead
public static IntPtr Add(IntPtr pointer, int offset);
public static IntPtr Subtract(IntPtr pointer, int offset);
public int ToInt32();
public long ToInt64();
public void* ToPointer();
```

The remaining members of `System.IntPtr` and `System.UIntPtr` *are implicitly included* in `nint` and `nuint`. For .NET Framework 4.7.2:

```
public override bool Equals(object obj);
public override int GetHashCode();
public override string ToString();
public string ToString(string format);
```

Interfaces implemented by `System.IntPtr` and `System.UIntPtr` *are implicitly included* in `nint` and `nuint`, with occurrences of the underlying types replaced by the corresponding native integer types. For instance if `IntPtr` implements `ISerializable, IEquatable<IntPtr>, IComparable<IntPtr>`, then `nint` implements `ISerializable, IEquatable<nint>, IComparable<nint>`.

Overriding, hiding, and implementing

`nint` and `System.IntPtr`, and `nuint` and `System.UIntPtr`, are considered equivalent for overriding, hiding, and implementing.

Overloads cannot differ by `nint` and `System.IntPtr`, and `nuint` and `System.UIntPtr`, alone. Overrides and implementations may differ by `nint` and `System.IntPtr`, or `nuint` and `System.UIntPtr`, alone. Methods hide other methods that differ by `nint` and `System.IntPtr`, or `nuint` and `System.UIntPtr`, alone.

Miscellaneous

`nint` and `nuint` expressions used as array indices are emitted without conversion.

```
static object GetItem(object[] array, nint index)
{
    return array[index]; // ok
}
```

`nint` and `nuint` can be used as an `enum` base type.

```
enum E : nint // ok
{
}
```

Reads and writes are atomic for types `nint`, `nuint`, and `enum` with base type `nint` or `nuint`.

Fields may be marked `volatile` for types `nint` and `nuint`. ECMA-334 15.5.4 does not include `enum` with base type `System.IntPtr` or `System.UIntPtr` however.

`default(nint)` and `new nint()` are equivalent to `(nint)0`.

`typeof(nint)` is `typeof(IntPtr)`.

`sizeof(nint)` is supported but requires compiling in an unsafe context (as does `sizeof(IntPtr)`). The value is not a compile-time constant. `sizeof(nint)` is implemented as `sizeof(IntPtr)` rather than `IntPtr.Size`.

Compiler diagnostics for type references involving `nint` or `nuint` report `nint` or `nuint` rather than `IntPtr` or `UIntPtr`.

Metadata

`nint` and `nuint` are represented in metadata as `System.IntPtr` and `System.UIntPtr`.

Type references that include `nint` or `nuint` are emitted with a `System.Runtime.CompilerServices.NativeIntegerAttribute` to indicate which parts of the type reference are native ints.

```
namespace System.Runtime.CompilerServices
{
    [AttributeUsage(
        AttributeTargets.Class |
        AttributeTargets.Event |
        AttributeTargets.Field |
        AttributeTargets.GenericParameter |
        AttributeTargets.Parameter |
        AttributeTargets.Property |
        AttributeTargets.ReturnValue,
        AllowMultiple = false,
        Inherited = false)]
    public sealed class NativeIntegerAttribute : Attribute
    {
        public NativeIntegerAttribute()
        {
            TransformFlags = new[] { true };
        }
        public NativeIntegerAttribute(bool[] flags)
        {
            TransformFlags = flags;
        }
        public readonly bool[] TransformFlags;
    }
}
```

The encoding of type references with `NativeIntegerAttribute` is covered in NativeIntegerAttribute.md.

Alternatives

An alternative to the "type erasure" approach above is to introduce new types: `System.NativeInt` and `System.NativeUInt`.

```
public readonly struct NativeInt
{
    public IntPtr Value;
}
```

Distinct types would allow overloading distinct from `IntPtr` and would allow distinct parsing and `ToString()`. But there would be more work for the CLR to handle these types efficiently which defeats the primary purpose of the feature - efficiency. And interop with existing native int code that uses `IntPtr` would be more difficult.

Another alternative is to add more native int support for `IntPtr` in the framework but without any specific compiler support. Any new conversions and arithmetic operations would be supported by the compiler automatically. But the language would not provide keywords, constants, or `checked` operations.

Design meetings

- https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-05-26.md
- https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-06-13.md
- https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-07-05.md#native-int-and-intptr-operators
- https://github.com/dotnet/csharplang/blob/master/meetings/2019/LDM-2019-10-23.md
- https://github.com/dotnet/csharplang/blob/master/meetings/2020/LDM-2020-03-25.md

Function Pointers

18.12.2021 • 22 minutes to read

Summary

This proposal provides language constructs that expose IL opcodes that cannot currently be accessed efficiently, or at all, in C# today: `ldftn` and `calli`. These IL opcodes can be important in high performance code and developers need an efficient way to access them.

Motivation

The motivations and background for this feature are described in the following issue (as is a potential implementation of the feature):

https://github.com/dotnet/csharplang/issues/191

This is an alternate design proposal to compiler intrinsics

Detailed Design

Function pointers

The language will allow for the declaration of function pointers using the `delegate*` syntax. The full syntax is described in detail in the next section but it is meant to resemble the syntax used by `Func` and `Action` type declarations.

```
unsafe class Example {
    void Example(Action<int> a, delegate*<int, void> f) {
        a(42);
        f(42);
    }
}
```

These types are represented using the function pointer type as outlined in ECMA-335. This means invocation of a `delegate*` will use `calli` where invocation of a `delegate` will use `callvirt` on the `Invoke` method. Syntactically though invocation is identical for both constructs.

The ECMA-335 definition of method pointers includes the calling convention as part of the type signature (section 7.1). The default calling convention will be `managed`. Unmanaged calling conventions can by specified by putting an `unmanaged` keyword afer the `delegate*` syntax, which will use the runtime platform default. Specific unmanaged conventions can then be specified in brackets to the `unmanaged` keyword by specifying any type starting with `CallConv` in the `System.Runtime.CompilerServices` namespace, leaving off the `CallConv` prefix. These types must come from the program's core library, and the set of valid combinations is platform-dependent.

```
//This method has a managed calling convention. This is the same as leaving the managed keyword off.
delegate* managed<int, int>;

// This method will be invoked using whatever the default unmanaged calling convention on the runtime
// platform is. This is platform and architecture dependent and is determined by the CLR at runtime.
delegate* unmanaged<int, int>;

// This method will be invoked using the cdecl calling convention
// Cdecl maps to System.Runtime.CompilerServices.CallConvCdecl
delegate* unmanaged[Cdecl] <int, int>;

// This method will be invoked using the stdcall calling convention, and suppresses GC transition
// Stdcall maps to System.Runtime.CompilerServices.CallConvStdcall
// SuppressGCTransition maps to System.Runtime.CompilerServices.CallConvSuppressGCTransition
delegate* unmanaged[Stdcall, SuppressGCTransition] <int, int>;
```

Conversions between `delegate*` types is done based on their signature including the calling convention.

```
unsafe class Example {
    void Conversions() {
        delegate*<int, int, int> p1 = ...;
        delegate* managed<int, int, int> p2 = ...;
        delegate* unmanaged<int, int, int> p3 = ...;

        p1 = p2; // okay p1 and p2 have compatible signatures
        Console.WriteLine(p2 == p1); // True
        p2 = p3; // error: calling conventions are incompatible
    }
}
```

A `delegate*` type is a pointer type which means it has all of the capabilities and restrictions of a standard pointer type:

- Only valid in an `unsafe` context.
- Methods which contain a `delegate*` parameter or return type can only be called from an `unsafe` context.
- Cannot be converted to `object`.
- Cannot be used as a generic argument.
- Can implicitly convert `delegate*` to `void*`.
- Can explicitly convert from `void*` to `delegate*`.

Restrictions:

- Custom attributes cannot be applied to a `delegate*` or any of its elements.
- A `delegate*` parameter cannot be marked as `params`
- A `delegate*` type has all of the restrictions of a normal pointer type.
- Pointer arithmetic cannot be performed directly on function pointer types.

Function pointer syntax

The full function pointer syntax is represented by the following grammar:

```
pointer_type
    : ...
    | funcptr_type
    ;

funcptr_type
    : 'delegate' '*' calling_convention_specifier? '<' funcptr_parameter_list funcptr_return_type '>'
    ;

calling_convention_specifier
    : 'managed'
    | 'unmanaged' ('[' unmanaged_calling_convention ']')?
    ;

unmanaged_calling_convention
    : 'Cdecl'
    | 'Stdcall'
    | 'Thiscall'
    | 'Fastcall'
    | identifier (',' identifier)*
    ;

funptr_parameter_list
    : (funcptr_parameter ',')*
    ;

funcptr_parameter
    : funcptr_parameter_modifier? type
    ;

funcptr_return_type
    : funcptr_return_modifier? return_type
    ;

funcptr_parameter_modifier
    : 'ref'
    | 'out'
    | 'in'
    ;

funcptr_return_modifier
    : 'ref'
    | 'ref readonly'
    ;
```

If no `calling_convention_specifier` is provided, the default is `managed`. The precise metadata encoding of the `calling_convention_specifier` and what `identifier`s are valid in the `unmanaged_calling_convention` is covered in Metadata Representation of Calling Conventions.

```
delegate int Func1(string s);
delegate Func1 Func2(Func1 f);

// Function pointer equivalent without calling convention
delegate*<string, int>;
delegate*<delegate*<string, int>, delegate*<string, int>>;

// Function pointer equivalent with calling convention
delegate* managed<string, int>;
delegate*<delegate* managed<string, int>, delegate*<string, int>>;
```

Function pointer conversions

In an unsafe context, the set of available implicit conversions (Implicit conversions) is extended to include the following implicit pointer conversions:

- *Existing conversions*
- From *funcptr_type* `F0` to another *funcptr_type* `F1`, provided all of the following are true:
  - `F0` and `F1` have the same number of parameters, and each parameter `D0n` in `F0` has the same `ref`, `out`, or `in` modifiers as the corresponding parameter `D1n` in `F1`.
  - For each value parameter (a parameter with no `ref`, `out`, or `in` modifier), an identity conversion, implicit reference conversion, or implicit pointer conversion exists from the parameter type in `F0` to the corresponding parameter type in `F1`.
  - For each `ref`, `out`, or `in` parameter, the parameter type in `F0` is the same as the corresponding parameter type in `F1`.
  - If the return type is by value (no `ref` or `ref readonly`), an identity, implicit reference, or implicit pointer conversion exists from the return type of `F1` to the return type of `F0`.
  - If the return type is by reference (`ref` or `ref readonly`), the return type and `ref` modifiers of `F1` are the same as the return type and `ref` modifiers of `F0`.
  - The calling convention of `F0` is the same as the calling convention of `F1`.

Allow address-of to target methods

Method groups will now be allowed as arguments to an address-of expression. The type of such an expression will be a `delegate*` which has the equivalent signature of the target method and a managed calling convention:

```
unsafe class Util {
    public static void Log() { }

    void Use() {
        delegate*<void> ptr1 = &Util.Log;

        // Error: type "delegate*<void>" not compatible with "delegate*<int>";
        delegate*<int> ptr2 = &Util.Log;
   }
}
```

In an unsafe context, a method `M` is compatible with a function pointer type `F` if all of the following are true:

- `M` and `F` have the same number of parameters, and each parameter in `M` has the same `ref`, `out`, or `in` modifiers as the corresponding parameter in `F`.
- For each value parameter (a parameter with no `ref`, `out`, or `in` modifier), an identity conversion, implicit reference conversion, or implicit pointer conversion exists from the parameter type in `M` to the corresponding parameter type in `F`.
- For each `ref`, `out`, or `in` parameter, the parameter type in `M` is the same as the corresponding parameter type in `F`.
- If the return type is by value (no `ref` or `ref readonly`), an identity, implicit reference, or implicit pointer conversion exists from the return type of `F` to the return type of `M`.
- If the return type is by reference (`ref` or `ref readonly`), the return type and `ref` modifiers of `F` are the same as the return type and `ref` modifiers of `M`.
- The calling convention of `M` is the same as the calling convention of `F`. This includes both the calling convention bit, as well as any calling convention flags specified in the unmanaged identifier.
- `M` is a static method.

In an unsafe context, an implicit conversion exists from an address-of expression whose target is a method group `E` to a compatible function pointer type `F` if `E` contains at least one method that is applicable in its normal form to an argument list constructed by use of the parameter types and modifiers of `F`, as described in the following.

- A single method `M` is selected corresponding to a method invocation of the form `E(A)` with the following modifications:
  - The arguments list `A` is a list of expressions, each classified as a variable and with the type and modifier (`ref`, `out`, or `in`) of the corresponding *funcptr_parameter_list* of `F`.
  - The candidate methods are only those methods that are applicable in their normal form, not those applicable in their expanded form.
  - The candidate methods are only those methods that are static.
- If the algorithm of overload resolution produces an error, then a compile-time error occurs. Otherwise, the algorithm produces a single best method `M` having the same number of parameters as `F` and the conversion is considered to exist.
- The selected method `M` must be compatible (as defined above) with the function pointer type `F`. Otherwise, a compile-time error occurs.
- The result of the conversion is a function pointer of type `F`.

This means developers can depend on overload resolution rules to work in conjunction with the address-of operator:

```
unsafe class Util {
    public static void Log() { }
    public static void Log(string p1) { }
    public static void Log(int i) { };

    void Use() {
        delegate*<void> a1 = &Log; // Log()
        delegate*<int, void> a2 = &Log; // Log(int i)

        // Error: ambiguous conversion from method group Log to "void*"
        void* v = &Log;
    }
```

The address-of operator will be implemented using the `ldftn` instruction.

Restrictions of this feature:

- Only applies to methods marked as `static`.
- Non-`static` local functions cannot be used in `&`. The implementation details of these methods are deliberately not specified by the language. This includes whether they are static vs. instance or exactly what signature they are emitted with.

Operators on Function Pointer Types

The section in unsafe code on operators is modified as such:

> In an unsafe context, several constructs are available for operating on all _pointer_type_s that are not _funcptr_type_s:
>
> - The `*` operator may be used to perform pointer indirection (Pointer indirection).
> - The `->` operator may be used to access a member of a struct through a pointer (Pointer member access).
> - The `[]` operator may be used to index a pointer (Pointer element access).
> - The `&` operator may be used to obtain the address of a variable (The address-of operator).
> - The `++` and `--` operators may be used to increment and decrement pointers (Pointer increment and decrement).
> - The `+` and `-` operators may be used to perform pointer arithmetic (Pointer arithmetic).
> - The `==`, `!=`, `<`, `>`, `<=`, and `=>` operators may be used to compare pointers (Pointer comparison).

- The `stackalloc` operator may be used to allocate memory from the call stack (Fixed size buffers).
- The `fixed` statement may be used to temporarily fix a variable so its address can be obtained (The fixed statement).

In an unsafe context, several constructs are available for operating on all _funcptr_type_s:

- The `&` operator may be used to obtain the address of static methods (Allow address-of to target methods)
- The `==`, `!=`, `<`, `>`, `<=`, and `=>` operators may be used to compare pointers (Pointer comparison).

Additionally, we modify all the sections in `Pointers in expressions` to forbid function pointer types, except `Pointer comparison` and `The sizeof operator`.

Better function member

The better function member specification will be changed to include the following line:

> A `delegate*` is more specific than `void*`

This means that it is possible to overload on `void*` and a `delegate*` and still sensibly use the address-of operator.

Type Inference

In unsafe code, the following changes are made to the type inference algorithms:

Input types

https://github.com/dotnet/csharplang/blob/master/spec/expressions.md#input-types

The following is added:

> If `E` is an address-of method group and `T` is a function pointer type then all the parameter types of `T` are input types of `E` with type `T`.

Output types

https://github.com/dotnet/csharplang/blob/master/spec/expressions.md#output-types

The following is added:

> If `E` is an address-of method group and `T` is a function pointer type then the return type of `T` is an output type of `E` with type `T`.

Output type inferences

https://github.com/dotnet/csharplang/blob/master/spec/expressions.md#output-type-inferences

The following bullet is added between bullets 2 and 3:

> - If `E` is an address-of method group and `T` is a function pointer type with parameter types `T1...Tk` and return type `Tb`, and overload resolution of `E` with the types `T1..Tk` yields a single method with return type `U`, then a *lower-bound inference* is made from `U` to `Tb`.

Exact inferences

https://github.com/dotnet/csharplang/blob/master/spec/expressions.md#exact-inferences

The following sub-bullet is added as a case to bullet 2:

> - `V` is a function pointer type `delegate*<V2..Vk, V1>` and `U` is a function pointer type

`delegate*<U2..Uk, U1>`, and the calling convention of `V` is identical to `U`, and the refness of `Vi` is identical to `Ui`.

Lower-bound inferences

https://github.com/dotnet/csharplang/blob/master/spec/expressions.md#lower-bound-inferences

The following case is added to bullet 3:

- `V` is a function pointer type `delegate*<V2..Vk, V1>` and there is a function pointer type `delegate*<U2..Uk, U1>` such that `U` is identical to `delegate*<U2..Uk, U1>`, and the calling convention of `V` is identical to `U`, and the refness of `Vi` is identical to `Ui`.

The first bullet of inference from `Ui` to `Vi` is modified to:

- If `U` is not a function pointer type and `Ui` is not known to be a reference type, or if `U` is a function pointer type and `Ui` is not known to be a function pointer type or a reference type, then an *exact inference* is made

Then, added after the 3rd bullet of inference from `Ui` to `Vi`:

- Otherwise, if `V` is `delegate*<V2..Vk, V1>` then inference depends on the i-th parameter of `delegate*<V2..Vk, V1>`:
  - If V1:
    - If the return is by value, then a *lower-bound inference* is made.
    - If the return is by reference, then an *exact inference* is made.
  - If V2..Vk:
    - If the parameter is by value, then an *upper-bound inference* is made.
    - If the parameter is by reference, then an *exact inference* is made.

Upper-bound inferences

https://github.com/dotnet/csharplang/blob/master/spec/expressions.md#upper-bound-inferences

The following case is added to bullet 2:

- `U` is a function pointer type `delegate*<U2..Uk, U1>` and `V` is a function pointer type which is identical to `delegate*<V2..Vk, V1>`, and the calling convention of `U` is identical to `V`, and the refness of `Ui` is identical to `Vi`.

The first bullet of inference from `Ui` to `Vi` is modified to:

- If `U` is not a function pointer type and `Ui` is not known to be a reference type, or if `U` is a function pointer type and `Ui` is not known to be a function pointer type or a reference type, then an *exact inference* is made

Then added after the 3rd bullet of inference from `Ui` to `Vi`:

- Otherwise, if `U` is `delegate*<U2..Uk, U1>` then inference depends on the i-th parameter of `delegate*<U2..Uk, U1>`:
  - If U1:
    - If the return is by value, then an *upper-bound inference* is made.
    - If the return is by reference, then an *exact inference* is made.

- If U2..Uk:
  - If the parameter is by value, then a *lower-bound inference* is made.
  - If the parameter is by reference, then an *exact inference* is made.

Metadata representation of `in`, `out`, and `ref readonly` parameters and return types

Function pointer signatures have no parameter flags location, so we must encode whether parameters and the return type are `in`, `out`, or `ref readonly` by using modreqs.

`in`

We reuse `System.Runtime.InteropServices.InAttribute`, applied as a `modreq` to the ref specifier on a parameter or return type, to mean the following:

- If applied to a parameter ref specifier, this parameter is treated as `in`.
- If applied to the return type ref specifier, the return type is treated as `ref readonly`.

`out`

We use `System.Runtime.InteropServices.OutAttribute`, applied as a `modreq` to the ref specifier on a parameter type, to mean that the parameter is an `out` parameter.

Errors

- It is an error to apply `OutAttribute` as a modreq to a return type.
- It is an error to apply both `InAttribute` and `OutAttribute` as a modreq to a parameter type.
- If either are specified via modopt, they are ignored.

Metadata Representation of Calling Conventions

Calling conventions are encoded in a method signature in metadata by a combination of the `CallKind` flag in the signature and zero or more `modopt`s at the start of the signature. ECMA-335 currently declares the following elements in the `CallKind` flag:

```
CallKind
   : default
   | unmanaged cdecl
   | unmanaged fastcall
   | unmanaged thiscall
   | unmanaged stdcall
   | varargs
   ;
```

Of these, function pointers in C# will support all but `varargs`.

In addition, the runtime (and eventually 335) will be updated to include a new `CallKind` on new platforms. This does not have a formal name currently, but this document will use `unmanaged ext` as a placeholder to stand for the new extensible calling convention format. With no `modopt`s, `unmanaged ext` is the platform default calling convention, `unmanaged` without the square brackets.

Mapping the `calling_convention_specifier` to a `CallKind`

A `calling_convention_specifier` that is omitted, or specified as `managed`, maps to the `default` `CallKind`. This is default `CallKind` of any method not attributed with `UnmanagedCallersOnly`.

C# recognizes 4 special identifiers that map to specific existing unmanaged `CallKind`s from ECMA 335. In order for this mapping to occur, these identifiers must be specified on their own, with no other identifiers, and this

requirement is encoded into the spec for `unmanaged_calling_convention`s. These identifiers are `Cdecl`, `Thiscall`, `Stdcall`, and `Fastcall`, which correspond to `unmanaged cdecl`, `unmanaged thiscall`, `unmanaged stdcall`, and `unmanaged fastcall`, respectively. If more than one `identifer` is specified, or the single `identifier` is not of the specially recognized identifiers, we perform special name lookup on the identifier with the following rules:

- We prepend the `identifier` with the string `CallConv`
- We look only at types defined in the `System.Runtime.CompilerServices` namespace.
- We look only at types defined in the core library of the application, which is the library that defines `System.Object` and has no dependencies.
- We look only at public types.

If lookup succeeds on all of the `identifier`s specified in an `unmanaged_calling_convention`, we encode the `CallKind` as `unmanaged ext`, and encode each of the resolved types in the set of `modopt`s at the beginning of the function pointer signature. As a note, these rules mean that users cannot prefix these `identifier`s with `CallConv`, as that will result in looking up `CallConvCallConvVectorCall`.

When interpreting metadata, we first look at the `CallKind`. If it is anything other than `unmanaged ext`, we ignore all `modopt`s on the return type for the purposes of determining the calling convention, and use only the `CallKind`. If the `CallKind` is `unmanaged ext`, we look at the modopts at the start of the function pointer type, taking the union of all types that meet the following requirements:

- The is defined in the core library, which is the library that references no other libraries and defines `System.Object`.
- The type is defined in the `System.Runtime.CompilerServices` namespace.
- The type starts with the prefix `CallConv`.
- The type is public.

These represent the types that must be found when performing lookup on the `identifier`s in an `unmanaged_calling_convention` when defining a function pointer type in source.

It is an error to attempt to use a function pointer with a `CallKind` of `unmanaged ext` if the target runtime does not support the feature. This will be determined by looking for the presence of the `System.Runtime.CompilerServices.RuntimeFeature.UnmanagedCallKind` constant. If this constant is present, the runtime is considered to support the feature.

`System.Runtime.InteropServices.UnmanagedCallersOnlyAttribute`

`System.Runtime.InteropServices.UnmanagedCallersOnlyAttribute` is an attribute used by the CLR to indicate that a method should be called with a specific calling convention. Because of this, we introduce the following support for working with the attribute:

- It is an error to directly call a method annotated with this attribute from C#. Users must obtain a function pointer to the method and then invoke that pointer.
- It is an error to apply the attribute to anything other than an ordinary static method or ordinary static local function. The C# compiler will mark any non-static or static non-ordinary methods imported from metadata with this attribute as unsupported by the language.
- It is an error for a method marked with the attribute to have a parameter or return type that is not an `unmanaged_type`.
- It is an error for a method marked with the attribute to have type parameters, even if those type parameters are constrained to `unmanaged`.
- It is an error for a method in a generic type to be marked with the attribute.
- It is an error to convert a method marked with the attribute to a delegate type.
- It is an error to specify any types for `UnmanagedCallersOnly.CallConvs` that do not meet the requirements for

calling convention `modopt`s in metadata.

When determining the calling convention of a method marked with a valid `UnmanagedCallersOnly` attribute, the compiler performs the following checks on the types specified in the `CallConvs` property to determine the effective `CallKind` and `modopt`s that should be used to determine the calling convention:

- If no types are specified, the `CallKind` is treated as `unmanaged ext`, with no calling convention `modopt`s at the start of the function pointer type.
- If there is one type specified, and that type is named `CallConvCdecl`, `CallConvThiscall`, `CallConvStdcall`, or `CallConvFastcall`, the `CallKind` is treated as `unmanaged cdecl`, `unmanaged thiscall`, `unmanaged stdcall`, or `unmanaged fastcall`, respectively, with no calling convention `modopt`s at the start of the function pointer type.
- If multiple types are specified or the single type is not named one of the specially called out types above, the `CallKind` is treated as `unmanaged ext`, with the union of the types specified treated as `modopt`s at the start of the function pointer type.

The compiler then looks at this effective `CallKind` and `modopt` collection and uses normal metadata rules to determine the final calling convention of the function pointer type.

Open Questions

Detecting runtime support for `unmanaged ext`

https://github.com/dotnet/runtime/issues/38135 tracks adding this flag. Depending on the feedback from review, we will either use the property specified in the issue, or use the presence of `UnmanagedCallersOnlyAttribute` as the flag that determines whether the runtimes supports `unmanaged ext`.

Considerations

Allow instance methods

The proposal could be extended to support instance methods by taking advantage of the `EXPLICITTHIS` CLI calling convention (named `instance` in C# code). This form of CLI function pointers puts the `this` parameter as an explicit first parameter of the function pointer syntax.

```
unsafe class Instance {
    void Use() {
        delegate* instance<Instance, string> f = &ToString;
        f(this);
    }
}
```

This is sound but adds some complication to the proposal. Particularly because function pointers which differed by the calling convention `instance` and `managed` would be incompatible even though both cases are used to invoke managed methods with the same C# signature. Also in every case considered where this would be valuable to have there was a simple work around: use a `static` local function.

```
unsafe class Instance {
    void Use() {
        static string toString(Instance i) => i.ToString();
        delegate*<Instance, string> f = &toString;
        f(this);
    }
}
```

Don't require unsafe at declaration

Instead of requiring `unsafe` at every use of a `delegate*`, only require it at the point where a method group is converted to a `delegate*`. This is where the core safety issues come into play (knowing that the containing assembly cannot be unloaded while the value is alive). Requiring `unsafe` on the other locations can be seen as excessive.

This is how the design was originally intended. But the resulting language rules felt very awkward. It's impossible to hide the fact that this is a pointer value and it kept peeking through even without the `unsafe` keyword. For example the conversion to `object` can't be allowed, it can't be a member of a `class`, etc ... The C# design is to require `unsafe` for all pointer uses and hence this design follows that.

Developers will still be capable of presenting a *safe* wrapper on top of `delegate*` values the same way that they do for normal pointer types today. Consider:

```
unsafe struct Action {
    delegate*<void> _ptr;

    Action(delegate*<void> ptr) => _ptr = ptr;
    public void Invoke() => _ptr();
}
```

Using delegates

Instead of using a new syntax element, `delegate*`, simply use existing `delegate` types with a `*` following the type:

```
Func<object, object, bool>* ptr = &object.ReferenceEquals;
```

Handling calling convention can be done by annotating the `delegate` types with an attribute that specifies a `CallingConvention` value. The lack of an attribute would signify the managed calling convention.

Encoding this in IL is problematic. The underlying value needs to be represented as a pointer yet it also must:

1. Have a unique type to allow for overloads with different function pointer types.
2. Be equivalent for OHI purposes across assembly boundaries.

The last point is particularly problematic. This mean that every assembly which uses `Func<int>*` must encode an equivalent type in metadata even though `Func<int>*` is defined in an assembly though don't control. Additionally any other type which is defined with the name `System.Func<T>` in an assembly that is not mscorlib must be different than the version defined in mscorlib.

One option that was explored was emitting such a pointer as `mod_req(Func<int>) void*`. This doesn't work though as a `mod_req` cannot bind to a `TypeSpec` and hence cannot target generic instantiations.

Named function pointers

The function pointer syntax can be cumbersome, particularly in complex cases like nested function pointers. Rather than have developers type out the signature every time the language could allow for named declarations of function pointers as is done with `delegate`.

```
func* void Action();

unsafe class NamedExample {
    void M(Action a) {
        a();
    }
}
```

Part of the problem here is the underlying CLI primitive doesn't have names hence this would be purely a C# invention and require a bit of metadata work to enable. That is doable but is a significant about of work. It essentially requires C# to have a companion to the type def table purely for these names.

Also when the arguments for named function pointers were examined we found they could apply equally well to a number of other scenarios. For example it would be just as convenient to declare named tuples to reduce the need to type out the full signature in all cases.

```
(int x, int y) Point;

class NamedTupleExample {
    void M(Point p) {
        Console.WriteLine(p.x);
    }
}
```

After discussion we decided to not allow named declaration of `delegate*` types. If we find there is significant need for this based on customer usage feedback then we will investigate a naming solution that works for function pointers, tuples, generics, etc ... This is likely to be similar in form to other suggestions like full `typedef` support in the language.

Future Considerations

static delegates

This refers to the proposal to allow for the declaration of `delegate` types which can only refer to `static` members. The advantage being that such `delegate` instances can be allocation free and better in performance sensitive scenarios.

If the function pointer feature is implemented the `static delegate` proposal will likely be closed out. The proposed advantage of that feature is the allocation free nature. However recent investigations have found that is not possible to achieve due to assembly unloading. There must be a strong handle from the `static delegate` to the method it refers to in order to keep the assembly from being unloaded out from under it.

To maintain every `static delegate` instance would be required to allocate a new handle which runs counter to the goals of the proposal. There were some designs where the allocation could be amortized to a single allocation per call-site but that was a bit complex and didn't seem worth the trade off.

That means developers essentially have to decide between the following trade offs:

1. Safety in the face of assembly unloading: this requires allocations and hence `delegate` is already a sufficient option.
2. No safety in face of assembly unloading: use a `delegate*`. This can be wrapped in a `struct` to allow usage outside an `unsafe` context in the rest of the code.

Suppress emitting of `localsinit` flag.

18.12.2021 • 3 minutes to read

Summary

Allow suppressing emit of `localsinit` flag via `SkipLocalsInitAttribute` attribute.

Motivation

Background

Per CLR spec local variables that do not contain references are not initialized to a particular value by the VM/JIT. Reading from such variables without initialization is type-safe, but otherwise the behavior is undefined and implementation specific. Typically uninitialized locals contain whatever values were left in the memory that is now occupied by the stack frame. That could lead to nondeterministic behavior and hard to reproduce bugs.

There are two ways to "assign" a local variable:

- by storing a value or
- by specifying `localsinit` flag which forces everything that is allocated form the local memory pool to be zero-initialized NOTE: this includes both local variables and `stackalloc` data.

Use of uninitialized data is discouraged and is not allowed in verifiable code. While it might be possible to prove that by the means of flow analysis, it is permitted for the verification algorithm to be conservative and simply require that `localsinit` is set.

Historically C# compiler emits `localsinit` flag on all methods that declare locals.

While C# employs definite-assignment analysis which is more strict than what CLR spec would require (C# also needs to consider scoping of locals), it is not strictly guaranteed that the resulting code would be formally verifiable:

- CLR and C# rules may not agree on whether passing a local as `out` argument is a `use`.
- CLR and C# rules may not agree on treatment of conditional branches when conditions are known (constant propagation).
- CLR could as well simply require `localinits`, since that is permitted.

Problem

In high-performance application the cost of forced zero-initialization could be noticeable. It is particularly noticeable when `stackalloc` is used.

In some cases JIT can elide initial zero-initialization of individual locals when such initialization is "killed" by subsequent assignments. Not all JITs do this and such optimization has limits. It does not help with `stackalloc`.

To illustrate that the problem is real - there is a known bug where a method not containing any `IL` locals would not have `localsinit` flag. The bug is already being exploited by users by putting `stackalloc` into such methods - intentionally to avoid initialization costs. That is despite the fact that absence of `IL` locals is an unstable metric and may vary depending on changes in codegen strategy. The bug should be fixed and users should get a more documented and reliable way of suppressing the flag.

Detailed design

Allow specifying `System.Runtime.CompilerServices.SkipLocalsInitAttribute` as a way to tell the compiler to not emit `localsinit` flag.

The end result of this will be that the locals may not be zero-initialized by the JIT, which is in most cases unobservable in C#.
In addition to that `stackalloc` data will not be zero-initialized. That is definitely observable, but also is the most motivating scenario.

Permitted and recognized attribute targets are: `Method`, `Property`, `Module`, `Class`, `Struct`, `Interface`, `Constructor`. However compiler will not require that attribute is defined with the listed targets nor it will care in which assembly the attribute is defined.

When attribute is specified on a container (`class`, `module`, containing method for a nested method, ...), the flag affects all methods contained within the container.

Synthesized methods "inherit" the flag from the logical container/owner.

The flag affects only codegen strategy for actual method bodies. I.E. the flag has no effect on abstract methods and is not propagated to overriding/implementing methods.

This is explicitly a ***compiler feature*** and ***not a language feature***.
Similarly to compiler command line switches the feature controls implementation details of a particular codegen strategy and does not need to be required by the C# spec.

Drawbacks

- Old/other compilers may not honor the attribute. Ignoring the attribute is compatible behavior. Only may result in a slight perf hit.
- The code without `localinits` flag may trigger verification failures. Users that ask for this feature are generally unconcerned with verifiability.
- Applying the attribute at higher levels than an individual method has nonlocal effect, which is observable when `stackalloc` is used. Yet, this is the most requested scenario.

Alternatives

- omit `localinits` flag when method is declared in `unsafe` context. That could cause silent and dangerous behavior change from deterministic to nondeterministic in a case of `stackalloc`.
- omit `localinits` flag always. Even worse than above.
- omit `localinits` flag unless `stackalloc` is used in the method body. Does not address the most requested scenario and may turn code unverifiable with no option to revert that back.

Unresolved questions

- Should the attribute be actually emitted to metadata?

Design meetings

None yet.

Unconstrained type parameter annotations

18.12.2021 • 2 minutes to read

Summary

Allow nullable annotations for type parameters that are not constrained to value types or reference types: `T?`.

```
static T? FirstOrDefault<T>(this IEnumerable<T> collection) { ... }
```

`?` annotation

In C#8, `?` annotations could only be applied to type parameters that were explicitly constrained to value types or reference types. In C#9, `?` annotations can be applied to any type parameter, regardless of constraints.

Unless a type parameter is explicitly constrained to value types, annotations can only be applied within a `#nullable enable` context.

If a type parameter `T` is substituted with a reference type, then `T?` represents a nullable instance of that reference type.

```
var s1 = new string[0].FirstOrDefault();  // string? s1
var s2 = new string?[0].FirstOrDefault(); // string? s2
```

If `T` is substituted with a value type, then `T?` represents an instance of `T`.

```
var i1 = new int[0].FirstOrDefault();  // int i1
var i2 = new int?[0].FirstOrDefault(); // int? i2
```

If `T` is substituted with an annotated type `U?`, then `T?` represents the annotated type `U?` rather than `U??`.

```
var u1 = new U[0].FirstOrDefault();  // U? u1
var u2 = new U?[0].FirstOrDefault(); // U? u2
```

If `T` is substituted with a type `U`, then `T?` represents `U?`, even within a `#nullable disable` context.

```
#nullable disable
var u3 = new U[0].FirstOrDefault();  // U? u3
```

For return values, `T?` is equivalent to `[MaybeNull]T`; for argument values, `T?` is equivalent to `[AllowNull]T`. The equivalence is important when overriding or implementing interfaces from an assembly compiled with C#8.

```
public abstract class A
{
    [return: MaybeNull] public abstract T F1<T>();
    public abstract void F2<T>([AllowNull] T t);
}

public class B : A
{
    public override T? F1<T>() where T : default { ... }       // matches A.F1<T>()
    public override void F2<T>(T? t) where T : default { ... } // matches A.F2<T>()
}
```

`default` constraint

For compatibility with existing code where overridden and explicitly implemented generic methods could not include explicit constraint clauses, `T?` in an overridden or explicitly implemented method is treated as `Nullable<T>` where `T` is a value type.

To allow annotations for type parameters constrained to reference types, C#8 allowed explicit `where T : class` and `where T : struct` constraints on the overridden or explicitly implemented method.

```
class A1
{
    public virtual void F1<T>(T? t) where T : struct { }
    public virtual void F1<T>(T? t) where T : class { }
}

class B1 : A1
{
    public override void F1<T>(T? t) /*where T : struct*/ { }
    public override void F1<T>(T? t) where T : class { }
}
```

To allow annotations for type parameters that are not constrained to reference types or value types, C#9 allows a new `where T : default` constraint.

```
class A2
{
    public virtual void F2<T>(T? t) where T : struct { }
    public virtual void F2<T>(T? t) { }
}

class B2 : A2
{
    public override void F2<T>(T? t) /*where T : struct*/ { }
    public override void F2<T>(T? t) where T : default { }
}
```

It is an error to use a `default` constraint other than on a method override or explicit implementation. It is an error to use a `default` constraint when the corresponding type parameter in the overridden or interface method is constrained to a reference type or value type.

Design meetings

- https://github.com/dotnet/csharplang/blob/master/meetings/2019/LDM-2019-11-25.md
- https://github.com/dotnet/csharplang/blob/master/meetings/2020/LDM-2020-06-17.md#t