**Глава 1. Введение в C#**

**Введение в C#**

**Язык C# и платформа .NET**

На сегодняшний момент язык программирования C# один из самых мощных, быстро развивающихся и востребованных языков в ИТ-отрасли. В настоящий момент на нем пишутся самые различные приложения: от небольших десктопных программок до крупных веб-порталов и веб-сервисов, обслуживающих ежедневно миллионы пользователей.

C# уже не молодой язык и как и вся платформа .NET уже прошел большой путь. Первая версия языка вышла вместе с релизом Microsoft Visual Studio .NET в феврале 2002 года. Текущей версией языка является версия C# 10.0, которая вышла 8 ноября 2021 года вместе с релизом .NET 6.

C# является языком с Си-подобным синтаксисом и близок в этом отношении к C++ и Java. Поэтому, если вы знакомы с одним из этих языков, то овладеть C# будет легче.

C# является объектно-ориентированным и в этом плане много перенял у Java и С++. Например, C# поддерживает полиморфизм, наследование, перегрузку операторов, статическую типизацию. Объектно-ориентированный подход позволяет решить задачи по построению крупных, но в тоже время гибких, масштабируемых и расширяемых приложений. И C# продолжает активно развиваться, и с каждой новой версией появляется все больше интересных функциональностей.

### **Роль платформы .NET**

Когда говорят C#, нередко имеют в виду технологии платформы .NET (Windows Forms, WPF, ASP.NET, Xamarin). И, наоборот, когда говорят .NET, нередко имеют в виду C#. Однако, хотя эти понятия связаны, отождествлять их неверно. Язык C# был создан специально для работы с фреймворком .NET, однако само понятие .NET несколько шире.

Как-то Билл Гейтс сказал, что платформа .NET - это лучшее, что создала компания Microsoft. Возможно, он был прав. Фреймворк .NET представляет мощную платформу для создания приложений. Можно выделить следующие ее основные черты:

* **Поддержка нескольких языков**. Основой платформы является общеязыковая среда исполнения Common Language Runtime (CLR), благодаря чему .NET поддерживает несколько языков: наряду с C# это также VB.NET, C++, F#, а также различные диалекты других языков, привязанные к .NET, например, Delphi.NET. При компиляции код на любом из этих языков компилируется в сборку на общем языке CIL (Common Intermediate Language) - своего рода ассемблер платформы .NET. Поэтому при определенных условиях мы можем сделать отдельные модули одного приложения на отдельных языках.
* **Кроссплатформенность**. .NET является переносимой платформой (с некоторыми ограничениями). Например, последняя версия платформы на данный момент - .NET 6 поддерживается на большинстве современных ОС Windows, MacOS, Linux. Используя различные технологии на платформе .NET, можно разрабатывать приложения на языке C# для самых разных платформ - Windows, MacOS, Linux, Android, iOS, Tizen.
* **Мощная библиотека классов**. .NET представляет единую для всех поддерживаемых языков библиотеку классов. И какое бы приложение мы не собирались писать на C# - текстовый редактор, чат или сложный веб-сайт - так или иначе мы задействуем библиотеку классов .NET.
* **Разнообразие технологий**. Общеязыковая среда исполнения CLR и базовая библиотека классов являются основой для целого стека технологий, которые разработчики могут задействовать при построении тех или иных приложений. Например, для работы с базами данных в этом стеке технологий предназначена технология ADO.NET и Entity Framework Core. Для построения графических приложений с богатым насыщенным интерфейсом - технология WPF и WinUI, для создания более простых графических приложений - Windows Forms. Для разработки кроссплатформенных мобильных и десктопных приложений - Xamarin/MAUI. Для создания веб-сайтов и веб-приложений - ASP.NET и т.д.

К этому стоит добавить активной развивающийся и набирающий популяность Blazor - фреймворк, который работает поверх .NET и который позволяет создавать веб-приложения как на стороне сервера, так и на стороне клиента. А в будущем будет поддерживать создание мобильных приложений и, возможно, десктоп-приложений.

* **Производительность**. Согласно ряду тестов веб-приложения на .NET 6 в ряде категорий сильно опережают веб-приложения, построенные с помощью других технологий. Приложения на .NET 6 в принципе отличаются высокой производительностью.

Также еще следует отметить такую особенность языка C# и фреймворка .NET, как автоматическая сборка мусора. А это значит, что нам в большинстве случаев не придется, в отличие от С++, заботиться об освобождении памяти. Вышеупомянутая общеязыковая среда CLR сама вызовет сборщик мусора и очистит память.

### **.NET Framework и .NET 6**

Стоит отметить, что .NET долгое время развивался премущественно как платформа для Windows под названием .NET Framework. В 2019 вышла последняя версия этой платформы - .NET Framework 4.8. Она больше не развивается

С 2014 Microsoft стал развивать альтернативную платформу - .NET Core, которая уже предназначалась для разных платформ и должна была вобрать в себя все возможности устаревшего .NET Framework и добавить новую функциональность. Затем Microsoft последовательно выпустил ряд версий этой платформы: .NET Core 1, .NET Core 2, .NET Core 3, .NET 5. И текущей версией является расматриваемая в этом руководстве платформа .NET 6. Поэтому следует различать .NET Framework, который предназначен преимущественно для Windows, и кросплатформенный .NET 6. В данном руководстве речь будет идти о C# 10 в связке с .NET 6, поскольку это актуальная платформа.

### **Управляемый и неуправляемый код**

Нередко приложение, созданное на C#, называют **управляемым кодом** (managed code). Что это значит? А это значит, что данное приложение создано на основе платформы .NET и поэтому управляется общеязыковой средой CLR, которая загружает приложение и при необходимости очищает память. Но есть также приложения, например, созданные на языке С++, которые компилируются не в общий язык CIL, как C#, VB.NET или F#, а в обычный машинный код. В этом случае .NET не управляет приложением.

В то же время платформа .NET предоставляет возможности для взаимодействия с неуправляемым кодом..

### **JIT-компиляция**

Как выше писалось, код на C# компилируется в приложения или сборки с расширениями exe или dll на языке CIL. Далее при запуске на выполнение подобного приложения происходит JIT-компиляция (Just-In-Time) в машинный код, который затем выполняется. При этом, поскольку наше приложение может быть большим и содержать кучу инструкций, в текущий момент времени будет компилироваться лишь та часть приложения, к которой непосредственно идет обращение. Если мы обратимся к другой части кода, то она будет скомпилирована из CIL в машинный код. При том уже скомпилированная часть приложения сохраняется до завершения работы программы. В итоге это повышает производительность.

По сути это все, что вкратце надо знать о платформе .NET и языке C#. А теперь создадим первое приложение.

## Начало работы. Visual Studio

Итак, создадим первое приложение на языке C#. Что для этого потребуется? Во-первых, нам нужен текстовый редактор, в котором мы могли бы напечатать код программы. Во-вторых, нам нужен компилятор, который бы скомпилировал набранный в текстовом редакторе код в приложение exe. В-третьих, нам нужен фреймворк .NET, который требуется для компиляции и выполнения программы.

Чтобы облегчить написание, а также тестирование и отладку программного кода обычно используют специальные среды разработки, в частности, Visual Studio.

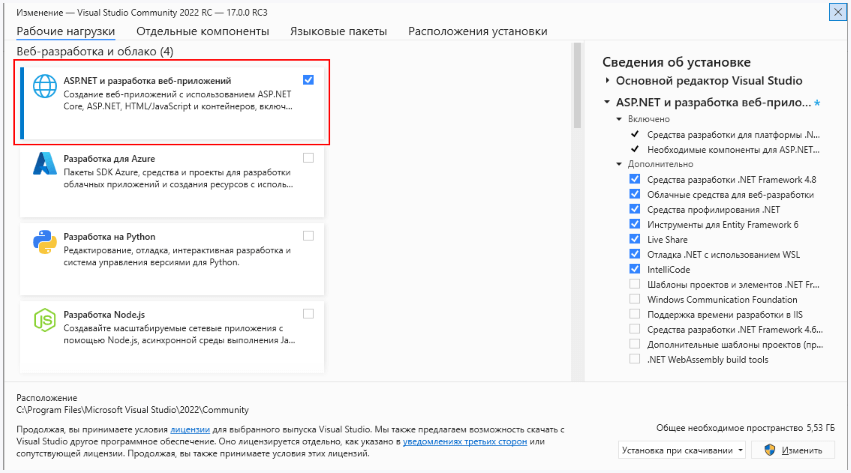
Для создания приложений на C# будем использовать бесплатную и полнофункциональную среду разработки - Visual Studio Community 2022, которую можно загрузить по следующему адресу: [Microsoft Visual Studio 2022](https://www.visualstudio.com/en-us/downloads)

## 

Стоит отметить, что Visual Studio 2019 и все предыдущие версии Visual Studio не поддерживают .NET 6 и C# 10, необходима именно версия Visual Studio 2022.

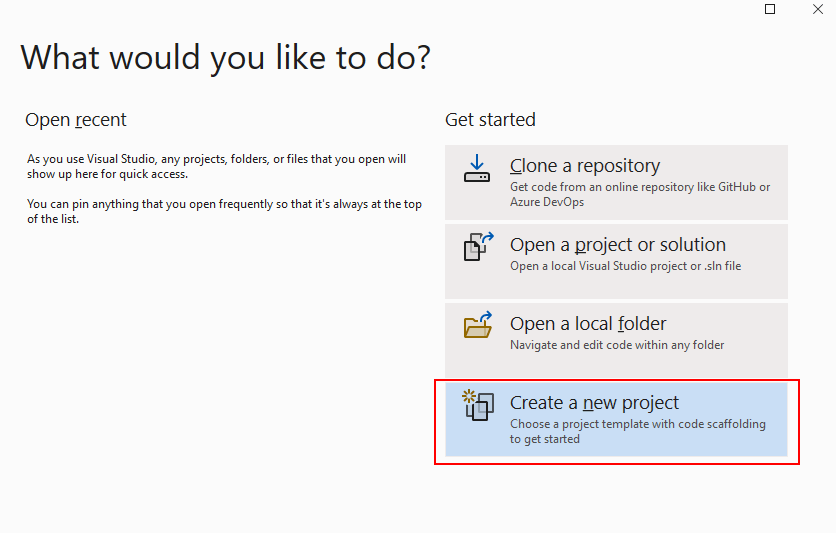
После загрузки запустим программу установщика. В открывшемся окне нам будет предложено выбрать те компоненты, которые мы хотим установить вместе Visual Studio. Стоит отметить, что Visual Studio - очень функциональная среда разработки и позволяет разрабатывать приложения с помощью множества языков и платформ. В нашем случае нам будет интересовать прежде всего C# и .NET.

Чтобы добавить в Visual Studio поддержку проектов для C# и .NET 6, в программе установки среди рабочих нагрузок можно выбрать только пункт **ASP.NET и разработка веб-приложений**. Можно выбрать и больше опций или вообще все опции, однако стоит учитывать свободный размер на жестком диске - чем больше опций будет выбрано, соответственно тем больше места на диске будет занято.

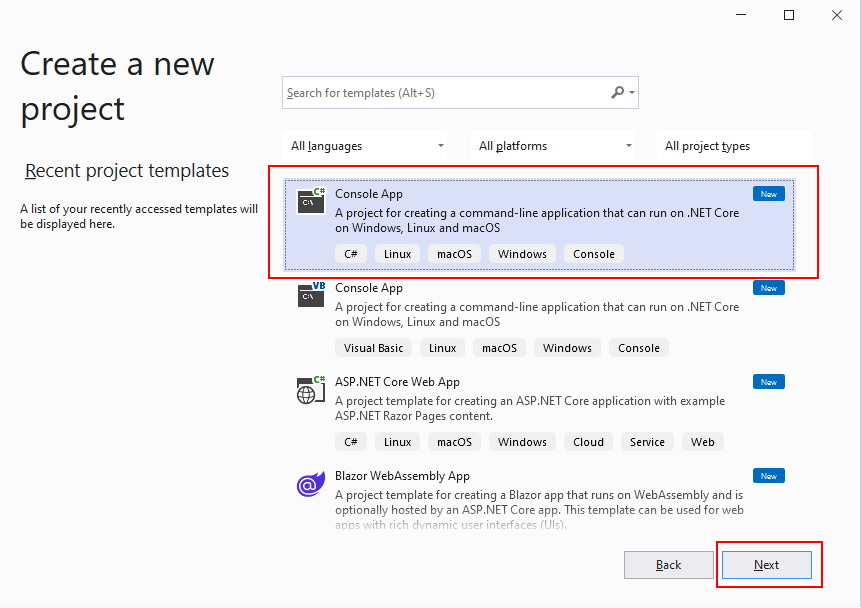


И при инсталляции Visual Studio на ваш компьютер будут установлены все необходимые инструменты для разработки программ, в том числе фреймворк .NET 6.

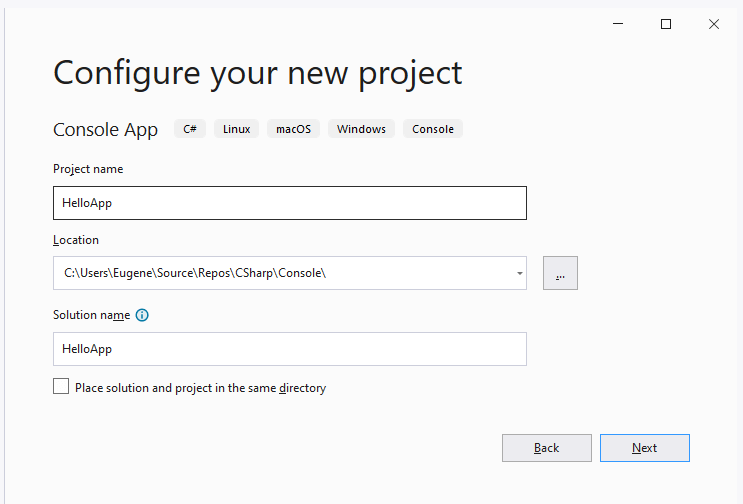
После завершения установки создадим первую программу. Она будет простенькой. Вначале откроем Visual Studio. На стартовом экране выберем **Create a new project** (Создать новый проект)



На следующем окне в качестве типа проекта выберем **Console App**, то есть мы будем создавать консольное приложение на языке C#.

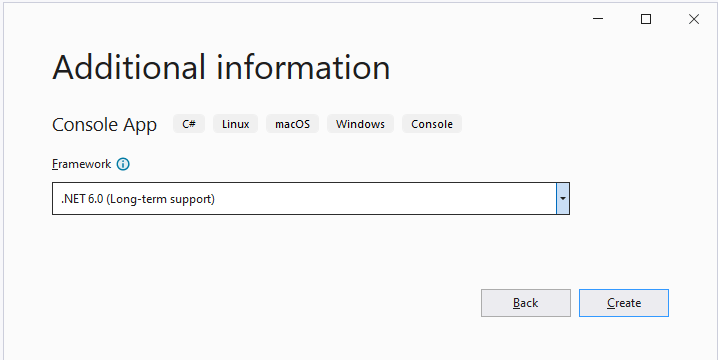


Далее на следующем этапе нам будет предложено указать имя проекта и каталог, где будет располагаться проект.

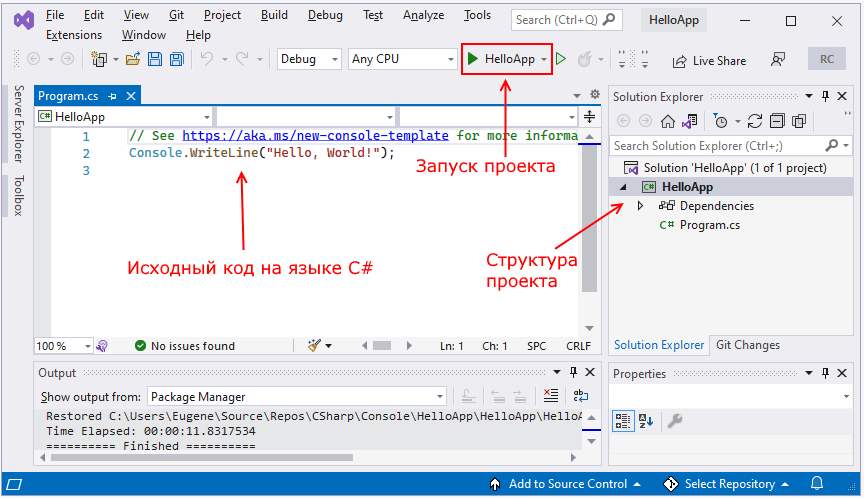


В поле **Project Name** дадим проекту какое-либо название. В моем случае это **HelloApp**.

На следующем окне Visual Studio предложит нам выбрать версию .NET, которая будет использоваться для проекта. По умолчанию здесь выбрана последняя на данный момент версия - .NET 6.0. Оставим и нажмен на кнопку Create (Создать) для создания проекта.



После этого Visual Studio создаст и откроет нам проект:



В большом поле в центре, которое по сути представляет текстовый редактор, находится сгенерированный по умолчанию код C#. Впоследствии мы изменим его на свой.

Справа находится окно Solution Explorer, в котором можно увидеть структуру нашего проекта. В данном случае у нас сгенерированная по умолчанию структура: узел **Dependencies** - это узел содержит сборки dll, которые добавлены в проект по умолчанию. Эти сборки как раз содержат классы библиотеки .NET, которые будет использовать C#. Однако не всегда все сборки нужны. Ненужные потом можно удалить, в то же время если понадобится добавить какую-нибудь нужную библиотеку, то именно в этот узел она будет добавляться.

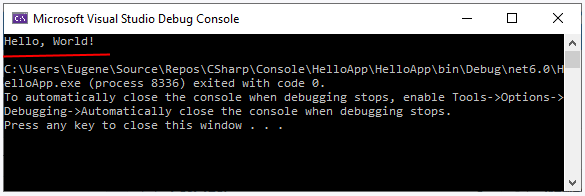
Далее идет непосредственно сам файл кода программы **Program.cs**, который по умолчанию открыт в центральном окне и который имеет всего две строки:

|  |  |
| --- | --- |
| 1  2 | // See <https://aka.ms/new-console-template> for more information  Console.WriteLine("Hello, World!"); |
|  |  |

Первая строка предваряется символами // и представляет комментарии - пояснения к коду.

Вторая строка собственно представляет собой код программы: Console.WriteLine("Hello World!");. Эта строка выводит на консоль строку "Hello World!".

Несмотря на то, что программа содержит только одну строку кода, это уже некоторая программа, которую мы можем запустить. Запустить проект мы можем с помощью клавиши F5 или с панели инструментов, нажав на зеленую стрелку. И если вы все сделали правильно, то при запуске приложения на консоль будет выведена строка "Hello World!".



Теперь изменим весь этот код на следующий:

|  |  |
| --- | --- |
| 1  2  3 | Console.Write("Введите свое имя: ");  string? name = Console.ReadLine();       // вводим имя  Console.WriteLine($"Привет {name}");    // выводим имя на консоль |

По сравнению с автоматически сгенерированным кодом я внес несколько изменений. Теперь первой строкой выводится приглашение к вводу.

|  |  |
| --- | --- |
| 1 | Console.Write("Введите свое имя: "); |

Метод **Console.Write()** выводит на консоль некоторую строка. В данном случае это строка "Введите свое имя: ".

На второй строке определяется строковая переменная name, в которую пользователь вводит информацию с консоли:

|  |  |
| --- | --- |
| 1 | string? name = Console.ReadLine(); |

Переменной name присваивается результат метода **Console.ReadLine()**, который позволяет считать с консоли введенную строку. То есть мы введем в консоли строку (точнее имя), и эта строка окажется в переменой name.

Выражение string? name указывает, что определяется переменная name типа string?, то есть переменная, которая может хранить строки. Здесь string указывает, что переменная может хранить значения типа string, то есть строки. А знак вопроса **?** указывает, что переменная также может хранить значение **null**, то есть по сути не иметь никакого значения.

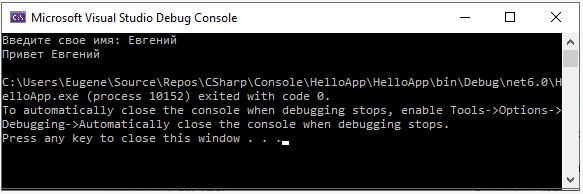
Затем введенное имя выводится на консоль:

|  |  |
| --- | --- |
| 1 | Console.WriteLine($"Привет {name}"); |

Чтобы ввести значение переменной name внутрь выводимой на консоль строки, применяются фигурные скобки {}. То есть при выводе строки на консоль выражение {name} будет заменяться на значение переменной name - введенное имя.

Однако чтобы можно было вводить таким образом значения переменных внутрь строки, перед строкой указывается знак доллара $.

Теперь протестируем проект, запустив его на выполнение, также нажав на F5 или зеленую стрелочку.



Итак, мы создали первое приложение. Вы его можете найти на жестком диске в папке проекта в каталоге **bin\Debug\net6.0**. Оно будет называться по имени проекта и иметь расширение exe. И затем этот файл можно будет запускать без Visual Studio, а также переносить его на другие компьютеры, где установлен .NET 6.

## Компиляция в командной строке с .NET CLI

Как правило, для создания программ на C# разработчики используют Visual Studio или какие-то другие IDE типа Rider от компании JetBrains. Но все подобные IDE, хотя и облегчают разработку, в то же время скрывают множество мельчайших подробностей по созданию программ. И в реальности мы даже можем обойтись без той же Visual Studio, используя только инфраструктуру .NET CLI. Рассмотрим, как использовать .NET CLI и компилировать приложение из командной строки.

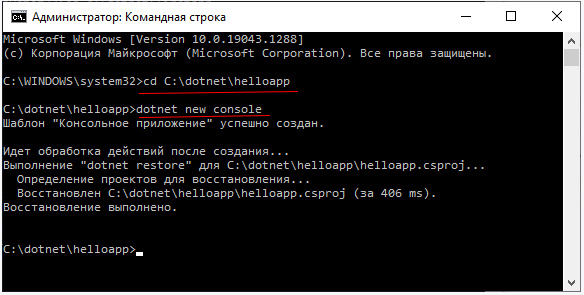
Поэтому создадим на диске C какую-нибудь новую папку для проекта. Например, пусть мы создали для проекта папку **C:\dotnet\helloapp**. Откроем командную строку и с помощью команды **cd** перейдем к этой папке.

C:\WINDOWS\system32>cd C:\dotnet\helloapp

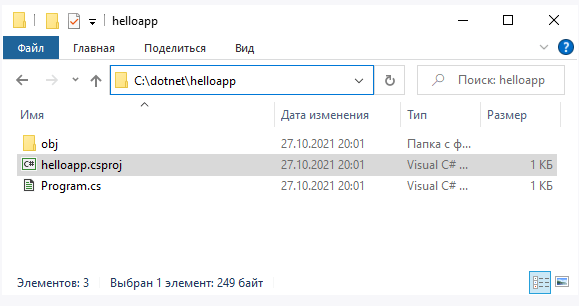
Для создания проекта в .NET CLI применяется команда **dotnet new**, которой передается название проекта. Итак, в введем в командной строке следующую команду:

C:\dotnet\helloapp>dotnet new console

В данном случае мы указываем, что создаваемый проект будет представлять консольное приложение.



После выполнения этой команды в папке helloapp будет создан проект с минимальным набором стандартных файлов и папок.



В частности, мы можем найти в папке проекта файл **helloapp.csproj**. Это главный файл проекта, который определяет его конфигурацию. Мы можем открыть его в любом текстовом редакторе, просмотреть и при необходимости изменить.

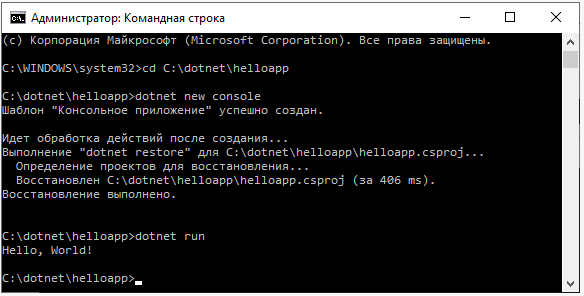
И, кроме того, по умолчанию создается главный файл программы **Program.cs** со следующим содержимым:

|  |  |
| --- | --- |
| 1  2 | // See <https://aka.ms/new-console-template> for more information  Console.WriteLine("Hello, World!"); |

Соответственно, если нам надо внести изменения в исходный код программы, то мы будем изменять содержимое именно этого файла.

В принципе этот минимальный проект уже можно запускать. Для запуска проекта введем в командной строке следующую команду:

dotnet run

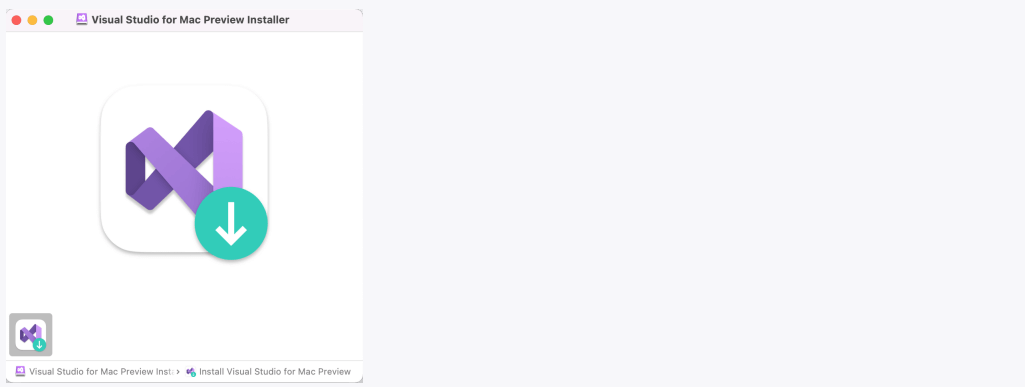


После выполнения команды в проекте в папке bin\MCD\Debug\net6.0 появится файл программы helloapp.exe, который можно вручную запустить на компьютере, где установлен .NET 6.

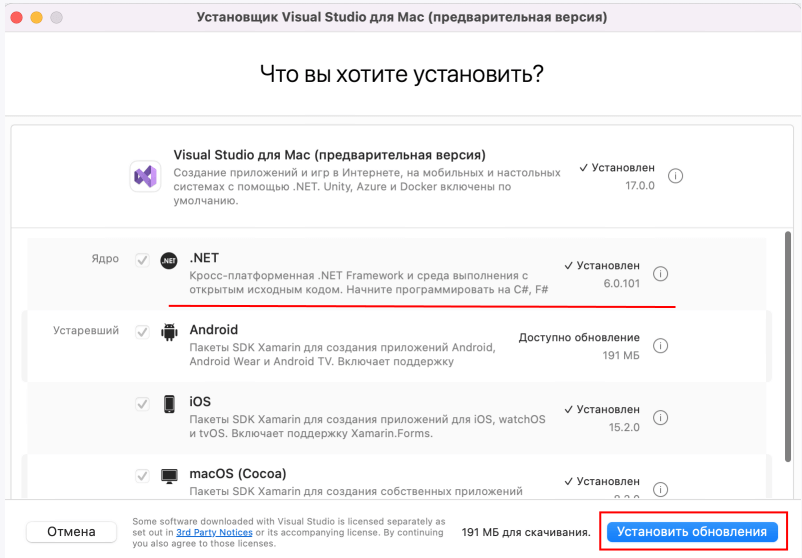
## Первая программа на MacOS

Фреймворк .NET является кроссплатформенным, соответственно создать приложения на C# можно не только на Windows, но и на других поддерживаемых операционных системах, например, на MacOS. Для работы с C# на MacOS можно использовать как .NET CLI, писать код программы в любом текстовом редакторе и компилировать в терминале. Но также для работы в MacOS можно использовать среду разработки **Visual Studio for Mac**. Для этого нам надо сначала загрузить установщик данной среды разработки со страницы <https://visualstudio.microsoft.com/ru/vs/mac/>. Для работы с .NET 6 и C# нам потребуется именно версия **Visual Studio for Mac 2022**

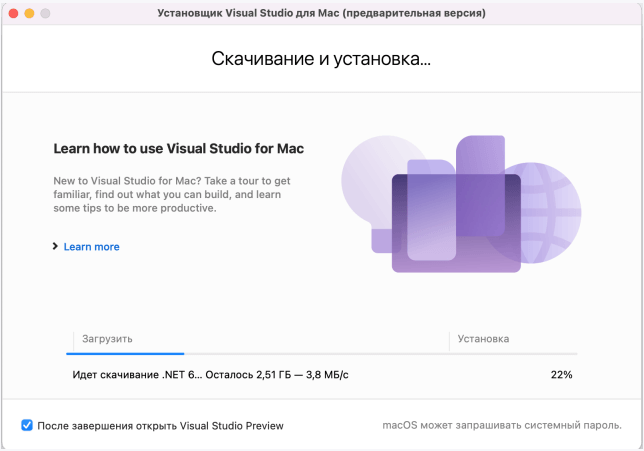
После загрузки запустим программу установки и нажмем на кнопку установки:



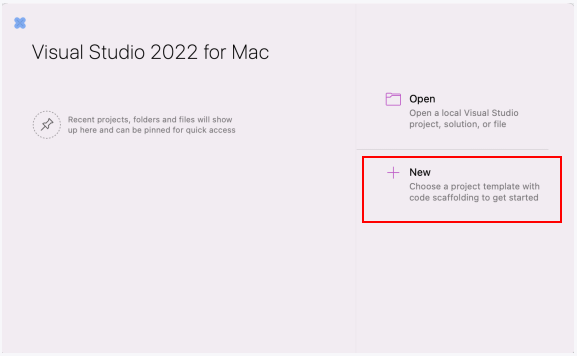
Далее нам откроется программа установки, где нам надо указать устанавливаемые компоненты. Если Visual Studio или отдельные компоненты уже были установлены, то отобразиться список компонентов, которые надо обновить:



Нажмем на кнопку установки, и начнется процесс установки:

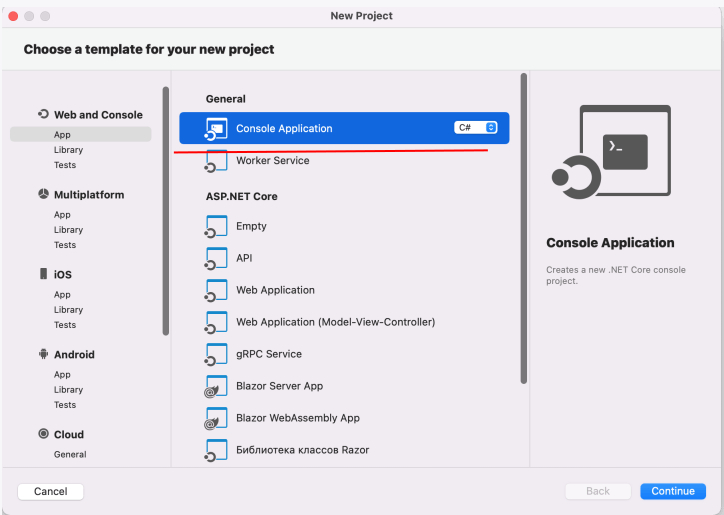


После завершения установки откроем Visual Studio for Mac:

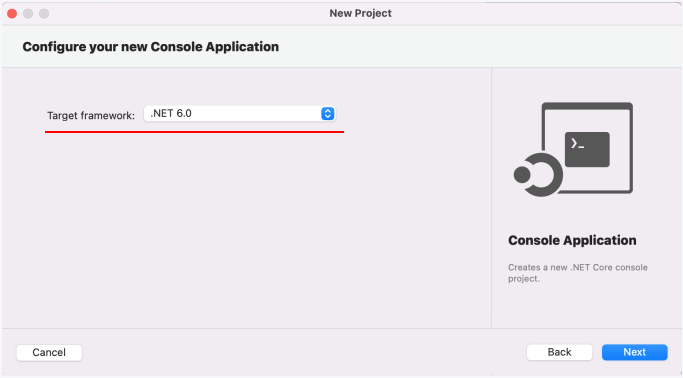


Для создания нового проекта нажмем на кнопку **New**

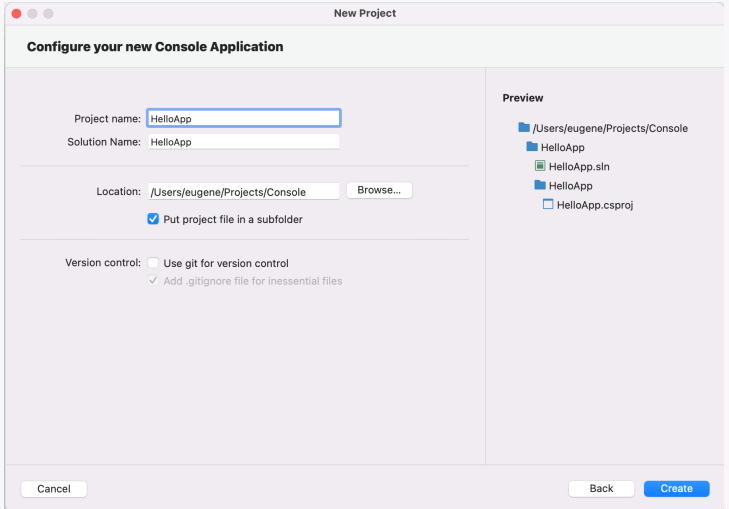
Далее нам откроется окно выбора шаблона проекта. В данном случае выберем шаблон **Console Application**



Далее в следующем окне укажем в качестве версии фреймворка .NET 6:



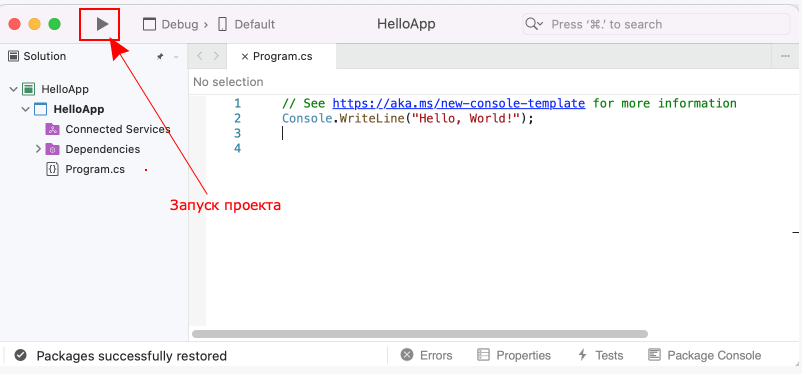
Далее в следующем окне укажем имя проекта и его расположение на жестком диске. Пусть проект называется HelloApp:



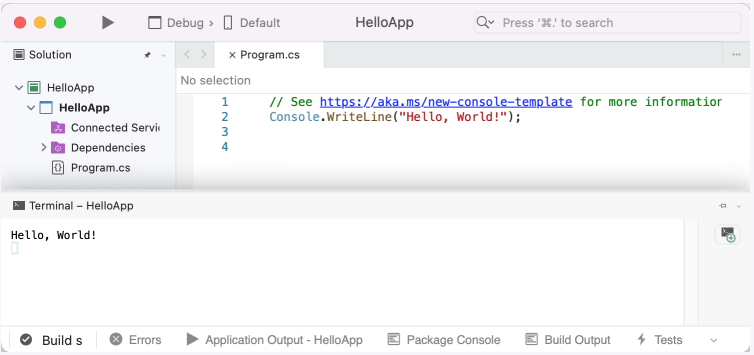
После этого Visual Studio создаст новый проект. Как и в версии для Windows, проект имеет один файл Program.cs. Он содержит простейший код, которые выводит сообщение на консоль:

|  |  |
| --- | --- |
| 1 | Console.WriteLine("Hello, World!"); |

Запустим проект, нажав на черную стрелку над структурой проекта:



И после запуска внизу в Visiual Studio for MacOS отобразится выше указанная строка:



В остальных аспектах разработка приложений на .NET и C# на MacOS будет аналогична разработке в Windows.

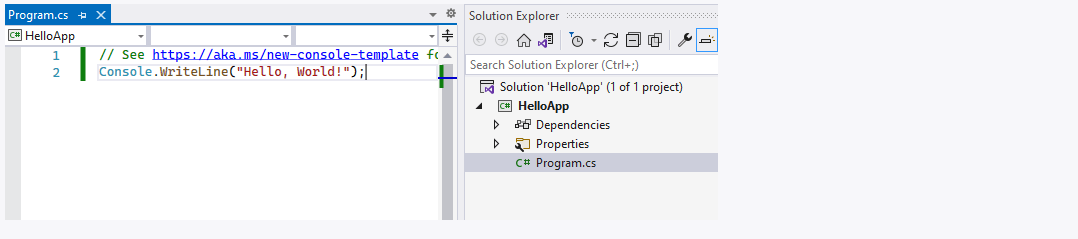
Глава 2. Основы программирования на C#

## Структура программы

### **Выполнение программы**

Весь код программы на языке C# помещается в файлы с расширением **.cs**. По умолчанию в проекте, который создается в Visual Studio (а также при использовании .NET CLI) уже есть один файл с кодом C# - файл **Program.cs** со следующим содержимым:

|  |  |
| --- | --- |
| 1  2 | // See <https://aka.ms/new-console-template> for more information  Console.WriteLine("Hello, World!"); |



Именно код файла **Program.cs** выполняется по умолчанию, если мы запустим проект на выполнение. Но при необходимости мы также можем добавлять другие файлы с кодом C#.

### **Инструкции**

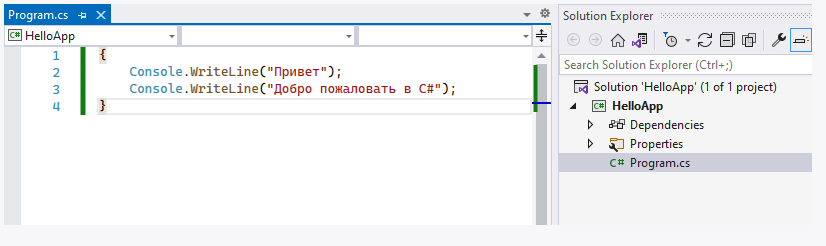
Базовым строительным блоком программы являются **инструкции** (statement). Инструкция представляет некоторое действие, например, арифметическую операцию, вызов метода, объявление переменной и присвоение ей значения. В конце каждой инструкции в C# ставится точка с запятой (;). Данный знак указывает компилятору на конец инструкции. Например, в проекте консольного приложения, который создается по умолчанию, есть такая строка:

|  |  |
| --- | --- |
| 1 | Console.WriteLine("Hello, World!"); |

Данная строка представляет вызов метода Console.WriteLine, который выводит на консоль строку. В данном случае вызов метода является инструкцией и поэтому завершается точкой с запятой.

Набор инструкций может объединяться в блок кода. Блок кода заключается в фигурные скобки, а инструкции помещаются между открывающей и закрывающей фигурными скобками. Например, изменим код файла **Program.cs** на следующий:

|  |  |
| --- | --- |
| 1  2  3  4 | {      Console.WriteLine("Привет");      Console.WriteLine("Добро пожаловать в C#");  } |



Здесь блок кода содержит две инструкции. И при выполении этого кода, консоль выведет две строки

Привет

Добро пожаловать в C#

В данном блоке кода две инструкции, которые выводят на консоль определенную строку.

Одни блоки кода могут содержать другие блоки:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | {      Console.WriteLine("Первый блок");      {          Console.WriteLine("Второй блок");      }  } |

### Регистрозависимость

C# является регистрозависимым языком. Это значит, что в зависимости от регистра символов какие-то определенные названия могут представлять разные классы, методы, переменные и т.д. Например, для вывода на консоль используется метод **WriteLine** - его имя начинается именно с большой буквы: "WriteLine". Если мы вместо "Console.WriteLine" напишем "Console.writeline", то программа не скомпилируется, так как данный метод обязательно должен называться "WriteLine", а не "writeline" или "WRITELINE" или как-то иначе.

### Комментарии

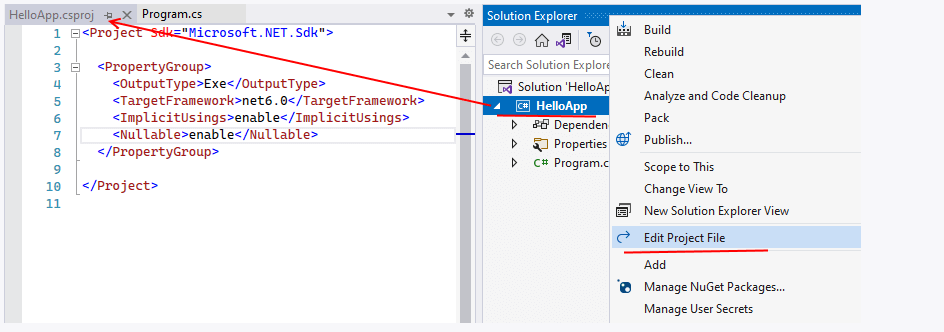
Важной частью программного кода являются комментарии. Они не являются собственно частью программы, при компиляции они игнорируются. Тем не менее комментарии делают код программы более понятным, помогая понять те или иные его части.

есть два типа комментариев: однострочный и многострочный. Однострочный комментарий размещается на одной строке после двойного слеша //. А многострочный комментарий заключается между символами /\* текст комментария \*/. Он может размещаться на нескольких строках. Например:

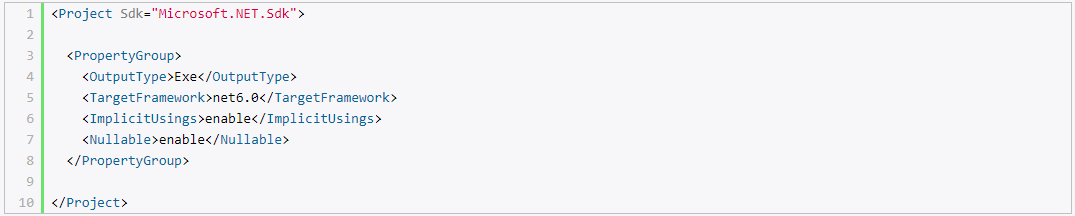
|  |  |
| --- | --- |
| 1  2  3  4  5  6 | /\*         первая программа на C#,         которая выводит приветствие на консоль  \*/  Console.WriteLine("Привет");                // Выводим строку "Привет"  Console.WriteLine("Добро пожаловать в C#"); // Выводим строку "Добро пожаловать в C#" |

### Файл проекта

В каждом проекте проекте C# есть файл, который отвечает за общую конфгурацию проекта. По умолчанию этот файл называется **Название\_проекта.csproj**. Итак, откроем данный файл. Для этого либо двойным кликом левой кнопкой мыши нажмем на название проекта, либо нажмем на название проекта правой кнопкой мыши и в появившемся меню выберем пункт **Edit Project File**



После этого Visual Studio откроет нам файл проекта, который будет выглядеть следующим образом:



Этот файл в виде кода xml определяет конфгурацию проекта и он может содержать множество элементов. Остановлюсь только на двух основных:

* **OutputType**: определяет выходной тип проекта. Это может быть выполняемое приложение в виде файла с расширением **exe**, которое запускается по нажатию. И также это может быть файл с расширением **.dll** - некоторый набор функциональностей, который используется другими проектами. По умолчанию здесь установлено значение "Exe", что значит, что мы создаем исполняемое приложение.
* **TargetFramework**: определяет применяемую для компиляции версию фреймворка .NET. В данном случае это значение "net6.0", то есть применяется .NET 6.0.

На самых ранних этапах этот файл может не понадобиться, однако впоследствии может потребоваться внести некоторые изменения в конфигурацию, и тогда может возникнуть потребность в обращении к этому файлу.

## Переменные и константы

Для хранения данных в программе применяются **переменные**. Переменная представляет именнованную область памяти, в которой хранится значение определенного типа. Переменная имеет тип, имя и значение. Тип определяет, какого рода информацию может хранить переменная.

Перед использованием любую переменную надо определить. Синтаксис определения переменной выглядит следующим образом:

|  |  |
| --- | --- |
| 1 | тип имя\_переменной; |

Вначале идет тип переменной, потом ее имя. В качестве имени переменной может выступать любое произвольное название, которое удовлетворяет следующим требованиям:

* имя может содержать любые цифры, буквы и символ подчеркивания, при этом первый символ в имени должен быть буквой или символом подчеркивания
* в имени не должно быть знаков пунктуации и пробелов
* имя не может быть ключевым словом языка C#. Таких слов не так много, и при работе в Visual Studio среда разработки подсвечивает ключевые слова синим цветом.

Хотя имя переменой может быть любым, но следует давать переменным описательные имена, которые будут говорить об их предназначении.

Например, определим простейшую переменную:

|  |  |
| --- | --- |
| 1 | string name; |

В данном случае определена переменная name, которая имеет тип **string**. то есть переменная представляет строку. Поскольку определение переменной представляет собой инструкцию, то после него ставится точка с запятой.

При этом следует учитывать, что C# является регистрозависимым языком, поэтому следующие два определения переменных будут представлять две разные переменные:

|  |  |
| --- | --- |
| 1  2 | string name;  string Name; |

После определения переменной можно присвоить некоторое значение:

|  |  |
| --- | --- |
| 1  2 | string name;  name = "Tom"; |

Так как переменная name представляет тип string, то есть строку, то мы можем присвоить ей строку в двойных кавычках. Причем переменной можно присвоить только то значение, которое соответствует ее типу.

В дальнейшем с помощью имени переменной мы сможем обращаться к той области памяти, в которой хранится ее значение.

Также мы можем сразу при определении присвоить переменной значение. Данный прием называется инициализацией:

|  |  |
| --- | --- |
| 1 | string name = "Tom"; |

Отличительной чертой переменных является то, что в программе можно многократно менять их значение. Например, создадим небольшую программу, в которой определим переменную, поменяем ее значение и выведем его на консоль:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | string name = "Tom";  // определяем переменную и инициализируем ее    Console.WriteLine(name);    // Tom    name = "Bob";       // меняем значение переменной  Console.WriteLine(name);    // Bob |

Консольный вывод программы:

Tom

Bob

### Константы

Отличительной особенностью переменных является то, что мы можем изменить их значение в процессе работы программы. Но, кроме того, в C# есть константы. **Константа** должна быть обязательно инициализирована при определении, и после определения значение константы не может быть изменено

Константы предназначены для описания таких значений, которые не должны изменяться в программе. Для определения констант используется ключевое слово **const**, которое указывается перед типом константы:

|  |  |
| --- | --- |
| 1 | const string NAME = "Tom";  // определяем константу |

Так, в данном случае определена константа NAME, которая хранит строку "Tom". Нередко для название констант используется верхний регистр, но это не более чем условность.

При использовании констант надо помнить, что объявить мы их можем только один раз и что к моменту компиляции они должны быть определены. Так, в следующем случае мы получим ошибку, так как константе не присвоено начальное значение:

|  |  |
| --- | --- |
| 1 | const string NAME;  // ! Ошибка - константа NAME не инициализирована |

Кроме того, мы ее не сможем изменим в процессе работы программы:

|  |  |
| --- | --- |
| 1  2 | const string NAME = "Tom";  // определяем константу  NAME = "Bob";   // !Ошибка - у констаты нельзя изменить значение |

Таким образом, если нам надо хранить в программе некоторые данные, но их не следует изменить, они определяются в виде констант. Если же их можно изменять, то они определяются в виде переменных.

## Литералы

Литералы представляют неизменяемые значения (иногда их еще называют константами). Литералы можно передавать переменным в качестве значения. Литералы бывают логическими, целочисленными, вещественными, символьными и строчными. И отдельный литерал представляет ключевое слово null.

### **Логические литералы**

Есть две логических константы - **true** (истина) и **false** (ложь):

|  |  |
| --- | --- |
| 1  2 | Console.WriteLine(true);  Console.WriteLine(false); |

### **Целочисленные литералы**

Целочисленные литералы представляют положительные и отрицательные целые числа, например, 1, 2, 3, 4, -7, -109. Целочисленные литералы могут быть выражены в десятичной, шестнадцатеричной и двоичной форме.

С целыми числами в десятичной форме все должно быть понятно, так как они используются в повседневной жизни:

|  |  |
| --- | --- |
| 1  2  3 | Console.WriteLine(-11);  Console.WriteLine(5);  Console.WriteLine(505); |

Числа в двоичной форме предваряются символами 0b, после которых идет набор из нулей и единиц:

|  |  |
| --- | --- |
| 1  2  3 | Console.WriteLine(0b11);        // 3  Console.WriteLine(0b1011);      // 11  Console.WriteLine(0b100001);    // 33 |

Для записи числа в шестнадцатеричной форме применяются символы 0x, после которых идет набор символов от 0 до 9 и от A до F, которые собственно представляют число:

|  |  |
| --- | --- |
| 1  2  3 | Console.WriteLine(0x0A);    // 10  Console.WriteLine(0xFF);    // 255  Console.WriteLine(0xA1);    // 161 |

### **Вещественные литералы**

Вещественные литералы представляют дробные числа. Этот тип литералов имеет две формы. Первая форма - вещественные числа с фиксированной запятой, при которой дробную часть отделяется от целой части точкой. Например:

|  |  |
| --- | --- |
| 1  2  3 | 3.14  100.001  -0.38 |

Также вещественные литералы могут определяться в экспоненциальной форме MEp, где M — мантисса, E - экспонента, которая фактически означает "\*10^" (умножить на десять в степени), а p — порядок. Например:

|  |  |
| --- | --- |
| 1  2 | Console.WriteLine(3.2e3);   // по сути равно 3.2 \* 10<sup>3</sup> = 3200  Console.WriteLine(1.2E-1);  // равно 1.2 \* 10<sup>-1</sup> = 0.12 |

### **Символьные литералы**

Символьные литералы представляют одиночные символы. Символы заключаются в одинарные кавычки.

Символьные литералы бывают нескольких видов. Прежде всего это обычные символы:

|  |  |
| --- | --- |
| 1  2  3 | '2'  'A'  'T' |

Также мы можем передать их вывести на консоль с помощью Console.WriteLine:

|  |  |
| --- | --- |
| 1  2  3 | Console.WriteLine('2');  Console.WriteLine('A');  Console.WriteLine('T'); |

Специальную группу представляют **управляющие последовательности** Управляющая последовательность представляет символ, перед которым ставится слеш. И данная последовательность интерпретируется определенным образом. Наиболее часто используемые последовательности:

* '\n' - перевод строки
* '\t' - табуляция
* '\\' - слеш

И если компилятор встретит в тексте последовательность \t, то он будет воспринимать эту последовательность не как слеш и букву t, а как табуляцию - то есть длинный отступ.

Также символы могут определяться в виде шестнадцатеричных кодов, также заключенный в одинарные кавычки.

Еще один способ определения символов представляет использования шестнадцатеричных кодов ASCII. Для этого в одинарных кавычках указываются символы '\x', после которых идет шестнадцатеричный код символа из таблицы ASCII. Коды символов из таблицы ASCII можно посмотреть [здесь](http://www.asciitable.com/).

Например, литерал '\x78' представляет символ "x":

|  |  |
| --- | --- |
| 1  2 | Console.WriteLine('\x78');    // x  Console.WriteLine('\x5A');    // Z |

И последний способ определения символьных литералов представляет применение кодов из таблицы символов [Unicode](https://unicode-table.com/ru/). Для этого в одинарных кавычках указываются символы '\u', после которых идет шестнадцатеричный код Unicode. Например, код '\u0411' представляет кириллический символ 'Б':

|  |  |
| --- | --- |
| 1  2 | Console.WriteLine('\u0420');    // Р  Console.WriteLine('\u0421');    // С |

### **Строковые литералы**

Строковые литералы представляют строки. Строки заключаются в двойные кавычки:

|  |  |
| --- | --- |
| 1  2  3 | Console.WriteLine("hello");  Console.WriteLine("фыва");  Console.WriteLine("hello word"); |

Если внутри строки необходимо вывести двойную кавычку, то такая внутренняя кавычка предваряется обратным слешем:

|  |  |
| --- | --- |
| 1 | Console.WriteLine("Компания \"Рога и копыта\""); |

Также в строках можно использовать управляющие последовательности. Например, последовательность '\n' осуществляет перевод на новую строку:

|  |  |
| --- | --- |
| 1 | Console.WriteLine("Привет \nмир"); |

При выводе на консоль слово "мир" будет перенесено на новую строку:

Привет

мир

### null

**null** представляет ссылку, которая не указывает ни на какой объект. То есть по сути отсутствие значения.

## Типы данных

Как и во многих языках программирования, в C# есть своя система типов данных, которая используется для создания переменных. Тип данных определяет внутреннее представление данных, множество значений, которые может принимать объект, а также допустимые действия, которые можно применять над объектом.

В языке C# есть следующие базовые типы данных:

* **bool:** хранит значение true или false (логические литералы). Представлен системным типом System.Boolean

|  |  |
| --- | --- |
| 1  2 | bool alive = true;  bool isDead = false; |

* **byte**: хранит целое число от 0 до 255 и занимает 1 байт. Представлен системным типом System.Byte

|  |  |
| --- | --- |
| 1  2 | byte bit1 = 1;  byte bit2 = 102; |

* **sbyte**: хранит целое число от -128 до 127 и занимает 1 байт. Представлен системным типом System.SByte

|  |  |
| --- | --- |
| 1  2 | sbyte bit1 = -101;  sbyte bit2 = 102; |

* **short**: хранит целое число от -32768 до 32767 и занимает 2 байта. Представлен системным типом System.Int16

|  |  |
| --- | --- |
| 1  2 | short n1 = 1;  short n2 = 102; |

* **ushort**: хранит целое число от 0 до 65535 и занимает 2 байта. Представлен системным типом System.UInt16

|  |  |
| --- | --- |
| 1  2 | ushort n1 = 1;  ushort n2 = 102; |

* **int**: хранит целое число от -2147483648 до 2147483647 и занимает 4 байта. Представлен системным типом System.Int32. Все целочисленные литералы по умолчанию представляют значения типа int:

|  |  |
| --- | --- |
| 1  2  3 | int a = 10;  int b = 0b101;  // бинарная форма b =5  int c = 0xFF;   // шестнадцатеричная форма c = 255 |

* **uint**: хранит целое число от 0 до 4294967295 и занимает 4 байта. Представлен системным типом System.UInt32

|  |  |
| --- | --- |
| 1  2  3 | uint a = 10;  uint b = 0b101;  uint c = 0xFF; |

* **long**: хранит целое число от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 и занимает 8 байт. Представлен системным типом System.Int64

|  |  |
| --- | --- |
| 1  2  3 | long a = -10;  long b = 0b101;  long c = 0xFF; |

* **ulong**: хранит целое число от 0 до 18 446 744 073 709 551 615 и занимает 8 байт. Представлен системным типом System.UInt64

|  |  |
| --- | --- |
| 1  2  3 | ulong a = 10;  ulong b = 0b101;  ulong c = 0xFF; |

* **float**: хранит число с плавающей точкой от -3.4\*1038 до 3.4\*1038 и занимает 4 байта. Представлен системным типом System.Single
* **double**: хранит число с плавающей точкой от ±5.0\*10-324 до ±1.7\*10308 и занимает 8 байта. Представлен системным типом System.Double
* **decimal**: хранит десятичное дробное число. Если употребляется без десятичной запятой, имеет значение от ±1.0\*10-28 до ±7.9228\*1028, может хранить 28 знаков после запятой и занимает 16 байт. Представлен системным типом System.Decimal
* **char**: хранит одиночный символ в кодировке Unicode и занимает 2 байта. Представлен системным типом System.Char. Этому типу соответствуют символьные литералы:

|  |  |
| --- | --- |
| 1  2  3 | char a = 'A';  char b = '\x5A';  char c = '\u0420'; |

* **string**: хранит набор символов Unicode. Представлен системным типом System.String. Этому типу соответствуют строковые литералы.

|  |  |
| --- | --- |
| 1  2 | string hello = "Hello";  string word = "world"; |

* **object**: может хранить значение любого типа данных и занимает 4 байта на 32-разрядной платформе и 8 байт на 64-разрядной платформе. Представлен системным типом System.Object, который является базовым для всех других типов и классов .NET.

|  |  |
| --- | --- |
| 1  2  3 | object a = 22;  object b = 3.14;  object c = "hello code"; |

Например, определим несколько переменных разных типов и выведем их значения на консоль:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | string name = "Tom";  int age = 33;  bool isEmployed = false;  double weight = 78.65;    Console.WriteLine($"Имя: {name}");  Console.WriteLine($"Возраст: {age}");  Console.WriteLine($"Вес: {weight}");  Console.WriteLine($"Работает: {isEmployed}"); |

Для вывода данных на консоль здесь применяется интерполяция: перед строкой ставится знак $ и после этого мы можем вводить в строку в фигурных скобках значения переменных. Консольный вывод программы:

Имя: Tom

Возраст: 33

Вес: 78,65

Работает: False

### Использование суффиксов

При присвоении значений надо иметь в виду следующую тонкость: все вещественные литералы (дробные числа) рассматриваются как значения типа **double**. И чтобы указать, что дробное число представляет тип **float** или тип **decimal**, необходимо к литералу добавлять суффикс: F/f - для float и M/m - для decimal.

|  |  |
| --- | --- |
| 1  2  3  4  5 | float a = 3.14F;  float b = 30.6f;    decimal c = 1005.8M;  decimal d = 334.8m; |

Подобным образом все целочисленные литералы рассматриваются как значения типа **int**. Чтобы явным образом указать, что целочисленный литерал представляет значение типа uint, надо использовать суффикс **U/u**, для типа **long** - суффикс **L/l**, а для типа **ulong** - суффикс **UL/ul**:

|  |  |
| --- | --- |
| 1  2  3 | uint a = 10U;  long b = 20L;  ulong c = 30UL; |

### Использование системных типов

Выше при перечислении всех базовых типов данных для каждого упоминался системный тип. Потому что название встроенного типа по сути представляет собой сокращенное обозначение системного типа. Например, следующие переменные будут эквивалентны по типу:

|  |  |
| --- | --- |
| 1  2 | int a = 4;  System.Int32 b = 4; |

### Неявная типизация

Ранее мы явным образом указывали тип переменных, например, int x;. И компилятор при запуске уже знал, что x хранит целочисленное значение.

Однако мы можем использовать и модель неявной типизации:

|  |  |
| --- | --- |
| 1  2 | var hello = "Hell to World";  var c = 20; |

Для неявной типизации вместо названия типа данных используется ключевое слово var. Затем уже при компиляции компилятор сам выводит тип данных исходя из присвоенного значения. Так как по умолчанию все целочисленные значения рассматриваются как значения типа int, то поэтому в итоге переменная c будет иметь тип int. Аналогично переменной hello присваивается строка, поэтому эта переменная будет иметь тип string

Эти переменные подобны обычным, однако они имеют некоторые ограничения.

Во-первых, мы не можем сначала объявить неявно типизируемую переменную, а затем инициализировать:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | // этот код работает  int a;  a = 20;    // этот код не работает  var c;  c= 20; |

Во-вторых, мы не можем указать в качестве значения неявно типизируемой переменной null:

|  |  |
| --- | --- |
| 1  2 | // этот код не работает  var c=null; |

Так как значение null, то компилятор не сможет вывести тип данных.

## Консольный ввод-вывод

### Консольный вывод

Для вывода информации на консоль мы уже использовали встроенный метод **Console.WriteLine**. То есть, если мы хотим вывести некоторую информацию на консоль, то нам надо передать ее в метод Console.WriteLine:

|  |  |
| --- | --- |
| 1  2  3  4  5 | string hello = "Привет мир";  Console.WriteLine(hello);  Console.WriteLine("Добро пожаловать в C#!");  Console.WriteLine("Пока мир...");  Console.WriteLine(24.5); |

Консольный вывод:

Привет мир!

Добро пожаловать в C#!

Пока мир...

24,5

Нередко возникает необходимость вывести на консоль в одной строке значения сразу нескольких переменных. В этом случае мы можем использовать прием, который называется интерполяцией:

|  |  |
| --- | --- |
| 1  2  3  4 | string name = "Tom";  int age = 34;  double height = 1.7;  Console.WriteLine($"Имя: {name}  Возраст: {age}  Рост: {height}м"); |

Для встраивания отдельных значений в выводимую на консоль строку используются фигурные скобки, в которые заключается встраиваемое значение. Это можем значение переменной ({name}) или более сложное выражение (например, операция сложения {4 + 7}). А перед всей строкой ставится знак доллара $.

При выводе на консоль вместо помещенных в фигурные скобки выражений будут выводиться их значения:

Имя: Tom Возраст: 34 Рост: 1,7м

Есть другой способ вывода на консоль сразу нескольких значений:

|  |  |
| --- | --- |
| 1  2  3  4 | string name = "Tom";  int age = 34;  double height = 1.7;  Console.WriteLine("Имя: {0}  Возраст: {2}  Рост: {1}м", name, height, age); |

Здесь мы видим, что строка в Console.WriteLine содержит некие числа в фигурных скобках: {0}, {1}, {2}. Это плейсхолдеры, вместо которых при выводе строки на консоль будут подставляться некоторые значения. Подставляемые значения указываются после строки через запятую.

При этом важен порядок подобных плейсхолдеров. Например, в данном случае после строки первой указана переменная name, потом height и потом age. Поэтому значение переменной name будет всавляться вместо первого плейсхолдера - {0} (нумерация начинается с нуля), height - вместо {1}, а age - вместо {2}. Поэтому в итоге пи выводе на консоль строка

|  |  |
| --- | --- |
| 1 | "Имя: {0}  Возраст: {2}  Рост: {1}м" |

будет заменена на

|  |  |
| --- | --- |
| 1 | "Имя: Tom  Возраст: 34  Рост: 1,7м" |

#### **Console.Write**

Кроме Console.WriteLine() можно также использовать метод **Console.Write()**, он работает точно так же за тем исключением, что не добавляет переход на следующую строку, то есть последующий консольный вывод будет выводиться на той же строке.

|  |  |
| --- | --- |
| 1  2  3  4 | string name = "Tom";  int age = 34;  double height = 1.7;  Console.Write($"Имя: {name}  Возраст: {age}  Рост: {height}м"); |

### Консольный ввод

Кроме вывода информации на консоль мы можем получать информацию с консоли. Для этого предназначен метод **Console.ReadLine()**. Он позволяет получить введенную строку.

|  |  |
| --- | --- |
| 1  2  3 | Console.Write("Введите свое имя: ");  string? name = Console.ReadLine();  Console.WriteLine($"Привет {name}"); |

В данном случае все, что вводит пользователь, с помощью метода Console.ReadLine() передается в переменную name.

Пример работы программы:

Введите свое имя: Том

Привет Том

Особенностью метода Console.ReadLine() является то, что он может считать информацию с консоли только в виде строки. Кроме того, возможная ситуация, когда для метода Console.ReadLine не окажется доступных для считывания строк, то есть когда ему нечего считывать, он возвращаает значение **null**, то есть, грубо говоря, фактически отсутствие значения. И чтобы отразить эту ситуацию мы определяем переменную name, в которую получаем ввод с консоли, как переменную типа **string?**. Здесь string указывает, что переменная может хранить значения типа string, то есть строки. А знак вопроса **?** указывает, что переменная также может хранить значение **null**, то есть по сути не иметь никакого значения. Далее мы более подробно разберем null и как с ним работать.

Однако, может возникнуть вопрос, как нам быть, если, допустим, мы хотим ввести возраст в переменную типа int или другую информацию в переменные типа double или decimal? По умолчанию платформа .NET предоставляет ряд методов, которые позволяют преобразовать различные значения к типам int, double и т.д. Некоторые из этих методов:

* **Convert.ToInt32()** (преобразует к типу int)
* **Convert.ToDouble()** (преобразует к типу double)
* **Convert.ToDecimal()** (преобразует к типу decimal)

Пример ввода значений:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | Console.Write("Введите имя: ");  string? name = Console.ReadLine();    Console.Write("Введите возраст: ");  int age = Convert.ToInt32(Console.ReadLine());    Console.Write("Введите рост: ");  double height = Convert.ToDouble(Console.ReadLine());    Console.Write("Введите размер зарплаты: ");  decimal salary = Convert.ToDecimal(Console.ReadLine());    Console.WriteLine($"Имя: {name}  Возраст: {age}  Рост: {height}м  Зарплата: {salary}$"); |

При вводе важно учитывать текущую операционную систему. В одних культурах разделителем между целой и дробной частью является точка (США, Великобритания...), в других - запятая (Россия, Германия...). Например, если текущая ОС - русскоязычная, значит, надо вводить дробные числа с разделителем запятой. Если локализация англоязычная, значит, разделителем целой и дробной части при вводе будет точка.

Пример работы программы:

Введите имя: Том

Введите возраст: 25

Введите рост: 1,75

Введите размер зарплаты: 300,67

Имя: Том Возраст: 25 Рост: 1,75м Зарплата: 300,67$

## Арифметические операции языка C#

В C# используется большинство операций, которые применяются и в других языках программирования. Операции представляют определенные действия над операндами - участниками операции. В качестве операнда может выступать переменной или какое-либо значение (например, число). Операции бывают унарными (выполняются над одним операндом), бинарными - над двумя операндами и тернарными - выполняются над тремя операндами. Рассмотрим все виды операций.

Бинарные арифметические операции:

* **+**

Операция сложения двух чисел:

|  |  |
| --- | --- |
| 1  2 | int x = 10;  int z = x + 12; // 22 |

* **-**

Операция вычитания двух чисел:

|  |  |
| --- | --- |
| 1  2 | int x = 10;  int z = x - 6; // 4 |

* **\***

Операция умножения двух чисел:

|  |  |
| --- | --- |
| 1  2 | int x = 10;  int z = x \* 5; // 50 |

* **/**

операция деления двух чисел:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | int x = 10;  int z = x / 5; // 2    double a = 10;  double b = 3;  double c = a / b; // 3.33333333 |

При делении стоит учитывать, что если оба операнда представляют целые числа, то результат также будет округляться до целого числа:

|  |  |
| --- | --- |
| 1 | double z = 10 /  4; //результат равен 2 |

Несмотря на то, что результат операции в итоге помещается в переменную типа double, которая позволяет сохранить дробную часть, но в самой операции участвуют два литерала, которые по умолчанию рассматриваются как объекты int, то есть целые числа, и результат то же будет целочисленный.

Для выхода из этой ситуации необходимо определять литералы или переменные, участвующие в операции, именно как типы double или float:

|  |  |
| --- | --- |
| 1 | double z = 10.0 /  4.0; //результат равен 2.5 |

* **%**

Операция получение остатка от целочисленного деления двух чисел:

|  |  |
| --- | --- |
| 1  2 | double x = 10.0;  double z = x % 4.0; //результат равен 2 |

Также есть ряд унарных операций, в которых принимает участие один операнд:

* **++**

Операция инкремента

Инкремент бывает префиксным: ++x - сначала значение переменной x увеличивается на 1, а потом ее значение возвращается в качестве результата операции.

И также существует постфиксный инкремент: x++ - сначала значение переменной x возвращается в качестве результата операции, а затем к нему прибавляется 1.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | int x1 = 5;  int z1 = ++x1; // z1=6; x1=6  Console.WriteLine($"{x1} - {z1}");    int x2 = 5;  int z2 = x2++; // z2=5; x2=6  Console.WriteLine($"{x2} - {z2}"); |

* **--**

Операция декремента или уменьшения значения на единицу. Также существует префиксная форма декремента (--x) и постфиксная (x--).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | int x1 = 5;  int z1 = --x1; // z1=4; x1=4  Console.WriteLine($"{x1} - {z1}");    int x2 = 5;  int z2 = x2--; // z2=5; x2=4  Console.WriteLine($"{x2} - {z2}"); |

При выполнении сразу нескольких арифметических операций следует учитывать порядок их выполнения. Приоритет операций от наивысшего к низшему:

1. Инкремент, декремент
2. Умножение, деление, получение остатка
3. Сложение, вычитание

Для изменения порядка следования операций применяются скобки.

Рассмотрим набор операций:

|  |  |
| --- | --- |
| 1  2  3  4  5 | int a = 3;  int b = 5;  int c = 40;  int d = c---b\*a;    // a=3  b=5  c=39  d=25  Console.WriteLine($"a={a}  b={b}  c={c}  d={d}"); |

Здесь мы имеем дело с тремя операциями: декремент, вычитание и умножение. Сначала выполняется декремент переменной c, затем умножение b\*a, и в конце вычитание. То есть фактически набор операций выглядел так:

|  |  |
| --- | --- |
| 1 | int d = (c--)-(b\*a); |

Но с помощью скобок мы могли бы изменить порядок операций, например, следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5 | int a = 3;  int b = 5;  int c = 40;  int d = (c-(--b))\*a;    // a=3  b=4  c=40  d=108  Console.WriteLine($"a={a}  b={b}  c={c}  d={d}"); |

### Ассоциативность операторов

Как выше было отмечено, операции умножения и деления имеют один и тот же приоритет, но какой тогда результат будет в выражении:

|  |  |
| --- | --- |
| 1 | int x = 10 / 5 \* 2; |

Стоит нам трактовать это выражение как (10 / 5) \* 2 или как 10 / (5 \* 2)? Ведь в зависимости от трактовки мы получим разные результаты.

Когда операции имеют один и тот же приоритет, порядок вычисления определяется ассоциативностью операторов. В зависимости от ассоциативности есть два типа операторов:

* Левоассоциативные операторы, которые выполняются слева направо
* Правоассоциативные операторы, которые выполняются справа налево

Все арифметические операторы являются левоассоциативными, то есть выполняются слева направо. Поэтому выражение 10 / 5 \* 2 необходимо трактовать как (10 / 5) \* 2, то есть результатом будет 4.

## Поразрядные операции

Особый класс операций представляют поразрядные операции. Они выполняются над отдельными разрядами числа. В этом плане числа рассматриваются в двоичном представлении, например, 2 в двоичном представлении 10 и имеет два разряда, число 7 - 111 и имеет три разряда.

### Логические операции

* **&**(логическое умножение)

Умножение производится поразрядно, и если у обоих операндов значения разрядов равно 1, то операция возвращает 1, иначе возвращается число 0. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | int x1 = 2; //010   int y1 = 5;//101   Console.WriteLine(x1&y1); // выведет 0    int x2 = 4; //100  int y2 = 5; //101  Console.WriteLine(x2 & y2); // выведет 4 |

В первом случае у нас два числа 2 и 5. 2 в двоичном виде представляет число 010, а 5 - 101. Поразрядно умножим числа (0\*1, 1\*0, 0\*1) и в итоге получим 000.

Во втором случае у нас вместо двойки число 4, у которого в первом разряде 1, так же как и у числа 5, поэтому в итоге получим (1\*1, 0\*0, 0 \*1) = 100, то есть число 4 в десятичном формате.

* **|** (логическое сложение)

Похоже на логическое умножение, операция также производится по двоичным разрядам, но теперь возвращается единица, если хотя бы у одного числа в данном разряде имеется единица. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | int x1 = 2; //010  int y1 = 5;//101  Console.WriteLine(x1|y1); // выведет 7 - 111  int x2 = 4; //100  int y2 = 5;//101  Console.WriteLine(x2 | y2); // выведет 5 - 101 |

* **^** (логическое исключающее ИЛИ)

Также эту операцию называют XOR, нередко ее применяют для простого шифрования:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | int x = 45; // Значение, которое надо зашифровать - в двоичной форме 101101  int key = 102; //Пусть это будет ключ - в двоичной форме 1100110    int encrypt = x ^ key; //Результатом будет число 1001011 или 75  Console.WriteLine($"Зашифрованное число: {encrypt}") ;    int decrypt = encrypt ^ key; // Результатом будет исходное число 45  Console.WriteLine($"Расшифрованное число: {decrypt}"); |

Здесь опять же производятся поразрядные операции. Если у нас значения текущего разряда у обоих чисел разные, то возвращается 1, иначе возвращается 0. Таким образом, мы получаем из 9^5 в качестве результата число 12. И чтобы расшифровать число, мы применяем ту же операцию к результату.

* **~** (логическое отрицание или инверсия)

Еще одна поразрядная операция, которая инвертирует все разряды: если значение разряда равно 1, то оно становится равным нулю, и наоборот.

|  |  |
| --- | --- |
| 1  2 | int x = 12;                 // 00001100  Console.WriteLine(~x);      // 11110011   или -13 |

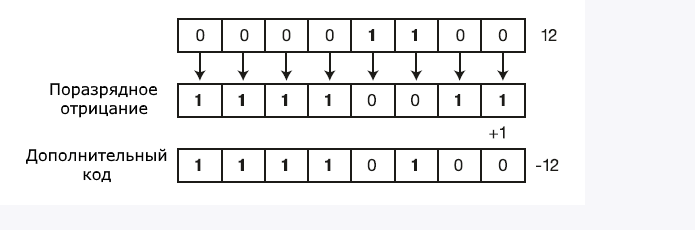
### Представление отрицательных чисел

Для записи чисел со знаком в C# применяется **дополнительный код** (two’s complement), при котором старший разряд является знаковым. Если его значение равно 0, то число положительное, и его двоичное представление не отличается от представления беззнакового числа. Например, 0000 0001 в десятичной системе 1.

Если старший разряд равен 1, то мы имеем дело с отрицательным числом. Например, 1111 1111 в десятичной системе представляет -1. Соответственно, 1111 0011 представляет -13.

Чтобы получить из положительного числа отрицательное, его нужно инвертировать и прибавить единицу:

|  |  |
| --- | --- |
| 1  2  3  4 | int x = 12;  int y = ~x;  y += 1;  Console.WriteLine(y);   // -12 |



### **Операции сдвига**

Операции сдвига также производятся над разрядами чисел. Сдвиг может происходить вправо и влево.

* x<<y - сдвигает число x влево на y разрядов. Например, 4<<1 сдвигает число 4 (которое в двоичном представлении 100) на один разряд влево, то есть в итоге получается 1000 или число 8 в десятичном представлении.
* x>>y - сдвигает число x вправо на y разрядов. Например, 16>>1 сдвигает число 16 (которое в двоичном представлении 10000) на один разряд вправо, то есть в итоге получается 1000 или число 8 в десятичном представлении.

Таким образом, если исходное число, которое надо сдвинуть в ту или другую строну, делится на два, то фактически получается умножение или деление на два. Поэтому подобную операцию можно использовать вместо непосредственного умножения или деления на два. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | int a = 16; // в двоичной форме 10000  int b = 2; // в двоичной форме  int c = a << b; // Сдвиг числа 10000 влево на 2 разряда, равно 1000000 или 64 в десятичной системе    Console.WriteLine($"Зашифрованное число: {c}") ;    // 64    int d = a >> b; // Сдвиг числа 10000 вправо на 2 разряда, равно 100 или 4 в десятичной системе  Console.WriteLine($"Зашифрованное число: {d}");     // 4 |

При этом числа, которые участвую в операциях, необязательно должны быть кратны 2:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | int a = 22; // в двоичной форме 10110  int b = 2; // в двоичной форме  int c = a << b; // Сдвиг числа 10110 влево на 2 разряда, равно 1011000 или 88 в десятичной системе    Console.WriteLine($"Зашифрованное число: {c}") ;    // 88    int d = a >> b; // Сдвиг числа 10110 вправо на 2 разряда, равно 101 или 5 в десятичной системе  Console.WriteLine($"Зашифрованное число: {d}");     // 5 |

## Операции присваивания

Операции присвоения устанавливают значение. В операциях присвоения участвуют два операнда, причем левый операнд может представлять только модифицируемое именованное выражение, например, переменную

Как и во многих других языках программирования, в C# имеется базовая операция присваивания **=**, которая присвоивает значение правого операнда левому операнду:

|  |  |
| --- | --- |
| 1 | int number = 23; |

Здесь переменной number присваивается число 23. Переменная number представляет левый операнд, которому присваивается значение правого операнда, то есть числа 23.

Также можно выполнять множественно присвоение сразу нескольких переменным одновременно:

|  |  |
| --- | --- |
| 1  2 | int a, b, c;  a = b = c = 34; |

Стоит отметить, что операции присвоения имеют низкий приоритет. И вначале будет вычисляться значение правого операнда и только потом будет идти присвоение этого значения левому операнду. Например:

|  |  |
| --- | --- |
| 1  2 | int a, b, c;  a = b = c = 34 \* 2 / 4; // 17 |

Сначала будет вычисляться выражение 34 \* 2 / 4, затем полученное значение будет присвоено переменным.

Кроме базовой операции присвоения в C# есть еще ряд операций:

* **+=**: присваивание после сложения. Присваивает левому операнду сумму левого и правого операндов: выражение **A += B** равнозначно выражению **A = A + B**
* **-=**: присваивание после вычитания. Присваивает левому операнду разность левого и правого операндов: **A -= B** эквивалентно **A = A - B**
* **\*=**: присваивание после умножения. Присваивает левому операнду произведение левого и правого операндов: **A \*= B** эквивалентно **A = A \* B**
* **/=**: присваивание после деления. Присваивает левому операнду частное левого и правого операндов: **A /= B** эквивалентно **A = A / B**
* **%=**: присваивание после деления по модулю. Присваивает левому операнду остаток от целочисленного деления левого операнда на правый: **A %= B** эквивалентно **A = A % B**
* **&=**: присваивание после поразрядной конъюнкции. Присваивает левому операнду результат поразрядной конъюнкции его битового представления с битовым представлением правого операнда: **A &= B** эквивалентно **A = A & B**
* **|=**: присваивание после поразрядной дизъюнкции. Присваивает левому операнду результат поразрядной дизъюнкции его битового представления с битовым представлением правого операнда: **A |= B** эквивалентно **A = A | B**
* **^=**: присваивание после операции исключающего ИЛИ. Присваивает левому операнду результат операции исключающего ИЛИ его битового представления с битовым представлением правого операнда: **A ^= B** эквивалентно **A = A ^ B**
* **<<=**: присваивание после сдвига разрядов влево. Присваивает левому операнду результат сдвига его битового представления влево на определенное количество разрядов, равное значению правого операнда: **A <<= B** эквивалентно **A = A << B**
* **>>=**: присваивание после сдвига разрядов вправо. Присваивает левому операнду результат сдвига его битового представления вправо на определенное количество разрядов, равное значению правого операнда: **A >>= B** эквивалентно **A = A >> B**

Применение операций присвоения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | int a = 10;  a += 10;        // 20  a -= 4;         // 16  a \*= 2;         // 32  a /= 8;         // 4  a <<= 4;      // 64  a >>= 2;      // 16 |

Операции присвоения являются правоассоциативными, то есть выполняются справа налево. Например:

|  |  |
| --- | --- |
| 1  2  3 | int a = 8;  int b = 6;  int c = a += b -= 5;    // 9 |

В данном случае выполнение выражения будет идти следующим образом:

1. b -= 5 (6-5=1)
2. a += (b-=5) (8+1 = 9)
3. c = (a += (b-=5)) (c = 9)

## Преобразования базовых типов данных

При рассмотрении типов данных указывалось, какие значения может иметь тот или иной тип и сколько байт памяти он может занимать. В прошлой теме были расмотрены арифметические операции. Теперь применим операцию сложения к данным разных типов:

|  |  |
| --- | --- |
| 1  2 | byte a = 4;  int b = a + 70; |

Результатом операции вполне справедливо является число 74, как и ожидается.

Но теперь попробуем применить сложение к двум объектам типа **byte**:

|  |  |
| --- | --- |
| 1  2 | byte a = 4;  byte b = a + 70;  // ошибка |

Здесь поменялся только тип переменной, которая получает результат сложения - с int на byte. Однако при попытке скомпилировать программу мы получим ошибку на этапе компиляции. И если мы работаем в Visual Studio, среда подчеркнет вторую строку красной волнистой линией, указывая, что в ней ошибка.

При операциях мы должны учитывать диапазон значений, которые может хранить тот или иной тип. Но в данном случае число 74, которое мы ожидаем получить, вполне укладывается в диапазон значений типа byte, тем не менее мы получаем ошибку.

Дело в том, что операция сложения (да и вычитания) возвращает значение типа int, если в операции участвуют целочисленные типы данных с разрядностью меньше или равно int (то есть типы byte, short, int). Поэтому результатом операции a + 70 будет объект, который имеет длину в памяти 4 байта. Затем этот объект мы пытаемся присвоить переменной b, которая имеет тип byte и в памяти занимает 1 байт.

И чтобы выйти из этой ситуации, необходимо применить операцию преобразования типов. **Операция преобразования** типов предполагает указание в скобках того типа, к которому надо преобразовать значение:

|  |  |
| --- | --- |
| 1 | (тип\_данных\_в\_который\_надо\_преобразовать)значение\_для\_преобразования; |

Так, изменим предыдущий пример, применив операцию преобразования типов:

|  |  |
| --- | --- |
| 1  2 | byte a = 4;  byte b = (byte)(a + 70); |

### Сужающие и расширяющие преобразования

Преобразования могут сужающие (narrowing) и расширяющие (widening). Расширяющие преобразования расширяют размер объекта в памяти. Например:

|  |  |
| --- | --- |
| 1  2 | byte a = 4;             // 0000100  ushort b = a;   // 000000000000100 |

В данном случае переменной типа ushort присваивается значение типа byte. Тип byte занимает 1 байт (8 бит), и значение переменной a в двоичном виде можно представить как:

|  |  |
| --- | --- |
| 1 | 00000100 |

Значение типа ushort занимает 2 байта (16 бит). И при присвоении переменной b значение переменной a расширяется до 2 байт

|  |  |
| --- | --- |
| 1 | 0000000000000100 |

То есть значение, которое занимает 8 бит, **расширяется** до 16 бит.

Сужающие преобразования, наоборот, сужают значение до типа меньшей разядности. Во втором листинге статьи мы как раз имели дело с сужающими преобразованиями:

|  |  |
| --- | --- |
| 1  2 | ushort a = 4;  byte b = (byte) a; |

Здесь переменной b, которая занимает 8 бит, присваивается значение ushort, которое занимает 16 бит. То есть из 0000000000000100 получаем 00000100. Таким образом, значение сужается с 16 бит (2 байт) до 8 бит (1 байт).

### **Явные и неявные преобразования**

#### **Неявные преобразования**

В случае с расширяющими преобразованиями компилятор за нас выполнял все преобразования данных, то есть преобразования были неявными (**implicit conversion**). Такие преобразования не вызывают каких-то затруднений. Тем не менее стоит сказать пару слов об общей механике подобных преобразований.

Если производится преобразование от безнакового типа меньшей разрядности к безнаковому типу большой разрядности, то добавляются дополнительные биты, которые имеют значени 0. Это называется **дополнение нулями** или zero extension.

|  |  |
| --- | --- |
| 1  2 | byte a = 4;             // 0000100  ushort b = a;   // 000000000000100 |

Если производится преобразование к знаковому типу, то битовое представление дополняется нулями, если число положительное, и единицами, если число отрицательное. Последний разряд числа содержит знаковый бит - 0 для положительных и 1 для отрицательных чисел. При расширении в добавленные разряды компируется знаковый бит.

Рассмотрим преобразование положительного числа:

|  |  |
| --- | --- |
| 1  2 | sbyte a = 4;            // 0000100  short b = a;    // 000000000000100 |

Преобразование отрицательного числа:

|  |  |
| --- | --- |
| 1  2 | sbyte a = -4;           // 1111100  short b = a;    // 111111111111100 |



#### **Явные преобразования**

При явных преобразованиях (**explicit conversion**) мы сами должны применить операцию преобразования (операция ()). Суть операции преобразования типов состоит в том, что перед значением указывается в скобках тип, к которому надо привести данное значение:

|  |  |
| --- | --- |
| 1  2  3 | int a = 4;  int b = 6;  byte c = (byte)(a+b); |

Расширяющие преобразования от типа с меньшей разрядностью к типу с большей разрядностью компилятор проводит неявно. Это могут быть следующие цепочки преобразований:

**byte** -> **short** -> **int** -> **long** -> **decimal**

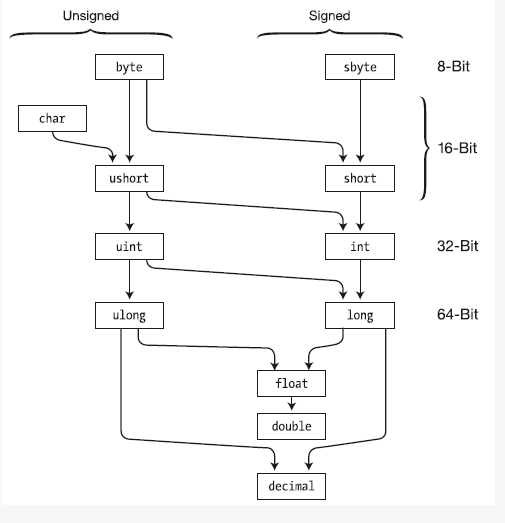
**int** -> **double**

**short** -> **float** -> **double**

**char** -> **int**

Все безопасные автоматические преобразования можно описать следующей таблицей:

|  |  |
| --- | --- |
| **Тип** | **В какие типы безопасно преобразуется** |
| byte | short, ushort, int, uint, long, ulong, float, double, decimal |
| sbyte | short, int, long, float, double, decimal |
| short | int, long, float, double, decimal |
| ushort | int, uint, long, ulong, float, double, decimal |
| int | long, float, double, decimal |
| uint | long, ulong, float, double, decimal |
| long | float, double, decimal |
| ulong | float, double, decimal |
| float | double |
| char | ushort, int, uint, long, ulong, float, double, decimal |



В остальных случаях следует использовать явные преобразования типов.

Также следует отметить, что несмотря на то, что и double, и decimal могут хранить дробные данные, а decimal имеет большую разрядность, чем double, но все равно значение double нужно явно приводить к типу decimal:

|  |  |
| --- | --- |
| 1  2 | double a = 4.0;  decimal b = (decimal)a; |

### **Потеря данных и ключевое слово checked**

Рассмотрим другую ситуацию, что будет, например, в следующем случае:

|  |  |
| --- | --- |
| 1  2  3  4 | int a = 33;  int b = 600;  byte c = (byte)(a+b);  Console.WriteLine(c);   // 121 |

Результатом будет число 121, так число 633 не попадает в допустимый диапазон для типа byte, и старшие биты будут усекаться. В итоге получится число 121. Поэтому при преобразованиях надо это учитывать. И мы в данном случае можем либо взять такие числа a и b, которые в сумме дадут число не больше 255, либо мы можем выбрать вместо byte другой тип данных, например, int.

Однако ситуации разные могут быть. Мы можем точно не знать, какие значения будут иметь числа a и b. И чтобы избежать подобных ситуаций, в c# имеется ключевое слово checked:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | try  {      int a = 33;      int b = 600;      byte c = checked((byte)(a + b));      Console.WriteLine(c);  }  catch (OverflowException ex)  {      Console.WriteLine(ex.Message);  } |

При использовании ключевого слова checked приложение выбрасывает исключение о переполнении. Поэтому для его обработки в данном случае используется конструкция try...catch. Подробнее данную конструкцию и обработку исключений мы рассмотрим позже, а пока надо знать, что в блок try мы включаем действия, в которых может потенциально возникнуть ошибка, а в блоке catch обрабатываем ошибку.

## Условные выражения

Отдельный набор операций представляет условные выражения. Такие операции возвращают логическое значение, то есть значение типа **bool**: **true**, если выражение истинно, и **false**, если выражение ложно. К подобным операциям относятся операции сравнения и логические операции.

### Операции сравнения

В операциях сравнения сравниваются два операнда и возвращается значение типа **bool** - **true**, если выражение верно, и **false**, если выражение неверно.

* **==**

Сравнивает два операнда на равенство. Если они равны, то операция возвращает **true**, если не равны, то возвращается **false**:

|  |  |
| --- | --- |
| 1  2  3 | int a = 10;  int b = 4;  bool c = a == b; // false |

* **!=**

Сравнивает два операнда и возвращает true, если операнды не равны, и false, если они равны.

|  |  |
| --- | --- |
| 1  2  3  4 | int a = 10;  int b = 4;  bool c = a != b;    // true  bool d = a!=10;     // false |

* **<**

Операция "меньше чем". Возвращает true, если первый операнд меньше второго, и false, если первый операнд больше второго:

|  |  |
| --- | --- |
| 1  2  3 | int a = 10;  int b = 4;  bool c = a < b; // false |

* **>**

Операция "больше чем". Сравнивает два операнда и возвращает true, если первый операнд больше второго, иначе возвращает false:

|  |  |
| --- | --- |
| 1  2  3  4 | int a = 10;  int b = 4;  bool c = a > b;     // true  bool d = a > 25;    // false |

* **<=**

Операция "меньше или равно". Сравнивает два операнда и возвращает true, если первый операнд меньше или равен второму. Иначе возвращает false.

|  |  |
| --- | --- |
| 1  2  3  4 | int a = 10;  int b = 4;  bool c = a <= b;     // false  bool d = a <= 25;    // true |

* **>=**

Операция "больше или равно". Сравнивает два операнда и возвращает true, если первый операнд больше или равен второму, иначе возвращается false:

|  |  |
| --- | --- |
| 1  2  3  4 | int a = 10;  int b = 4;  bool c = a >= b;     // true  bool d = a >= 25;    // false |

Операции <, > <=, >= имеют больший приоритет, чем == и !=.

### Логические операции

Также в C# определены логические операторы, которые также возвращают значение типа **bool**. В качестве операндов они принимают значения типа **bool**. Как правило, применяются к отношениям и объединяют несколько операций сравнения.

* **|**

Операция логического сложения или логическое ИЛИ. Возвращает true, если хотя бы один из операндов возвращает true.

|  |  |
| --- | --- |
| 1  2 | bool x1 = (5 > 6) | (4 < 6); // 5 > 6 - false, 4 < 6 - true, поэтому возвращается true  bool x2 = (5 > 6) | (4 > 6); // 5 > 6 - false, 4 > 6 - false, поэтому возвращается false |

* **&**

Операция логического умножения или логическое И. Возвращает true, если оба операнда одновременно равны true.

|  |  |
| --- | --- |
| 1  2 | bool x1 = (5 > 6) & (4 < 6); // 5 > 6 - false, 4 < 6 - true, поэтому возвращается false  bool x2 = (5 < 6) & (4 < 6); // 5 < 6 - true, 4 < 6 - true, поэтому возвращается true |

* **||**

Операция логического сложения. Возвращает true, если хотя бы один из операндов возвращает true.

|  |  |
| --- | --- |
| 1  2 | bool x1 = (5 > 6) || (4 < 6); // 5 > 6 - false, 4 < 6 - true, поэтому возвращается true  bool x2 = (5 > 6) || (4 > 6); // 5 > 6 - false, 4 > 6 - false, поэтому возвращается false |

* **&&**

Операция логического умножения. Возвращает true, если оба операнда одновременно равны true.

|  |  |
| --- | --- |
| 1  2 | bool x1 = (5 > 6) && (4 < 6); // 5 > 6 - false, 4 < 6 - true, поэтому возвращается false  bool x2 = (5 < 6) && (4 < 6); // 5 < 6 - true, 4 < 6 - true, поэтому возвращается true |

* **!**

Операция логического отрицания. Производится над одним операндом и возвращает true, если операнд равен false. Если операнд равен true, то операция возвращает false:

|  |  |
| --- | --- |
| 1  2 | bool a = true;  bool b = !a;    // false |

* **^**

Операция исключающего ИЛИ. Возвращает true, если либо первый, либо второй операнд (но не одновременно) равны true, иначе возвращает false

|  |  |
| --- | --- |
| 1  2 | bool x5 = (5 > 6) ^ (4 < 6); // 5 > 6 - false, 4 < 6 - true, поэтому возвращается true  bool x6 = (50 > 6) ^ (4 / 2 < 3); // 50 > 6 - true, 4/2 < 3 - true, поэтому возвращается false |

Здесь у нас две пары операций | и || (а также & и &&) выполняют похожие действия, однако же они не равнозначны.

В выражении z=x|y; будут вычисляться оба значения - x и y.

В выражении же z=x||y; сначала будет вычисляться значение x, и если оно равно true, то вычисление значения y уже смысла не имеет, так как у нас в любом случае уже z будет равно true. Значение y будет вычисляться только в том случае, если x равно false

То же самое касается пары операций &/&&. В выражении z=x&y; будут вычисляться оба значения - x и y.

В выражении же z=x&&y; сначала будет вычисляться значение x, и если оно равно false, то вычисление значения y уже смысла не имеет, так как у нас в любом случае уже z будет равно false. Значение y будет вычисляться только в том случае, если x равно true

Поэтому операции || и && более удобны в вычислениях, так как позволяют сократить время на вычисление значения выражения, и тем самым повышают производительность. А операции | и & больше подходят для выполнения поразрядных операций над числами.

## Конструкция if..else и тернарная операция

Условные конструкции - один из базовых компонентов многих языков программирования, которые направляют работу программы по одному из путей в зависимости от определенных условий. Одной из таких конструкций в языке программирования C# является конструкция if..else

Конструкция if/else проверяет истинность некоторого условия и в зависимости от результатов проверки выполняет определенный код.

Ее простейшая форма состоит из блока **if**:

|  |  |
| --- | --- |
| 1  2  3  4 | if(условие)  {      выполняемые инструкции  } |

После ключевого слова if ставится условие. Условие должно представлять значение типа **bool**. Это может быть непосредственно значение типа bool или результат условного выражения или другого выражения, которое возвращает значение тиа bool. И если это условие истинно (равно true), то срабатывает код, который помещен далее после условия внутри фигурных скобок.

Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | int num1 = 8;  int num2 = 6;  if(num1 > num2)  {      Console.WriteLine($"Число {num1} больше числа {num2}");  } |

В данном случае у нас первое число больше второго, поэтому выражение num1 > num2 истинно и возвращает true, следовательно, управление переходит к строке Console.WriteLine("Число {num1} больше числа {num2}");

Если блок if содержит одну инструкцию, то мы можем его сократить, убрав фигурные скобки:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | int num1 = 8;  int num2 = 6;  if (num1 > num2)      Console.WriteLine($"Число {num1} больше числа {num2}");    // или так  if (num1 > num2) Console.WriteLine($"Число {num1} больше числа {num2}"); |

Также мы можем соединить сразу несколько условий, используя логические операторы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | int num1 = 8;  int num2 = 6;  if(num1 > num2 && num1==8)  {      Console.WriteLine($"Число {num1} больше числа {num2}");  } |

В данном случае блок if будет выполняться, если num1 > num2 равно true и num1==8 равно true.

#### **Выражение else**

Но что, если мы захотим, чтобы при несоблюдении условия также выполнялись какие-либо действия? В этом случае мы можем добавить блок else:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | int num1 = 8;  int num2 = 6;  if(num1 > num2)  {      Console.WriteLine($"Число {num1} больше числа {num2}");  }  else  {      Console.WriteLine($"Число {num1} меньше числа {num2}");  } |

Блок else выполняется, если условие после if ложно, то есть равно false. Если блок else содержит толко одну инструкцию, то оять же мы можем его сократить, убрав фигурные скобки:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | int num1 = 8;  int num2 = 6;  if(num1 > num2)      Console.WriteLine($"Число {num1} больше числа {num2}");  else      Console.WriteLine($"Число {num1} меньше числа {num2}"); |

#### **else if**

Но в примере выше при сравнении чисел мы можем насчитать три состояния: первое число больше второго, первое число меньше второго и числа равны. Используя конструкцию else if, мы можем обрабатывать дополнительные условия:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | int num1 = 8;  int num2 = 6;  if(num1 > num2)  {      Console.WriteLine($"Число {num1} больше числа {num2}");  }  else if (num1 < num2)  {      Console.WriteLine($"Число {num1} меньше числа {num2}");  }  else  {      Console.WriteLine("Число num1 равно числу num2");  } |

При необходимости можно добавить несколько выражений else if:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | string name = "Alex";    if (name == "Tom")      Console.WriteLine("Вас зовут Tomas");  else if (name == "Bob")      Console.WriteLine("Вас зовут Robert");  else if (name == "Mike")      Console.WriteLine("Вас зовут Michael");  else      Console.WriteLine("Неизвестное имя"); |

### **Тернарная операция**

Тернарную операция также позволяет проверить некоторое условие и в зависимости от его истинности выполнить некоторые действия. Она имеет следующий синтаксис:

|  |  |
| --- | --- |
| 1 | [первый операнд - условие] ? [второй операнд] : [третий операнд] |

Здесь сразу три операнда. В зависимости от условия тернарная операция возвращает второй или третий операнд: если условие равно true, то возвращается второй операнд; если условие равно false, то третий. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5 | int x=3;  int y=2;    int z = x < y ? (x+y) : (x-y);  Console.WriteLine(z);   // 1 |

Здесь первый операнд (то есть условие) представляет выражение x < y. Если оно равно true, то возвращается второй операнд - (x+y), то есть результат операции сложения. Если условие равно false, то возвращается третий операнд - (x-y).

Результат тернарной операции (то есть второй или третий операнд в зависимости от условия) присваивается переменной z.

## Циклы

Циклы являются управляющими конструкциями, позволяя в зависимости от определенных условий выполнять некоторое действие множество раз. В C# имеются следующие виды циклов:

* for
* foreach
* while
* do...while

### **Цикл for**

Цикл for имеет следующее формальное определение:

|  |  |
| --- | --- |
| 1  2  3  4 | for ([действия\_до\_выполнения\_цикла]; [условие]; [действия\_после\_выполнения])  {      // действия  } |

Объявление цикла **for** состоит из трех частей. Первая часть объявления цикла - некоторые действия, которые выполняются один раз до выполнения цикла. Обычно здесь определяются переменные, которые будут использоваться в цикле.

Вторая часть - условие, при котором будет выполняться цикл. Пока условие равно true, будет выполняться цикл.

И третья часть - некоторые действия, которые выполняются после завершения блока цикла. Эти действия выполняются каждый раз при завершении блока цикла.

После объявления цикла в фигурных скобках помещаются сами действия цикла.

Рассмотрим стандартный цикл for:

|  |  |
| --- | --- |
| 1  2  3  4 | for (int i = 1; i < 4; i++)  {      Console.WriteLine(i);  } |

Здесь первая часть объявления цикла - int i = 1 - создает и инициализирует переменную i.

Вторая часть - условие i < 4. То есть пока переменная i меньше 4, будет выполняться цикл.

И третья часть - действия, выполняемые после завершения действий из блока цикла - увеличение переменной i на единицу.

Весь процесс цикла можно представить следующим образом:

1. Определяется переменная int i = 1
2. Проверяется условие i < 4. Оно истинно (так как 1 меньше 4), поэтому выполняется блок цикла, а именно инструкция Console.WriteLine(i), которая выводит на консоль значение переменной i
3. Блок цикла закончил выполнение, поэтому выполняется треться часть объявления цикла - i++. После этого переменная i будет равна 2.
4. Снова проверяется условие i < 4. Оно истинно (так как 2 меньше 4), поэтому опять выполняется блок цикла - Console.WriteLine(i)
5. Блок цикла закончил выполнение, поэтому снова выполняется выражение i++. После этого переменная i будет равна 3.
6. Снова проверяется условие i < 4. Оно истинно (так как 3 меньше 4), поэтому опять выполняется блок цикла - Console.WriteLine(i)
7. Блок цикла закончил выполнение, поэтому снова выполняется выражение i++. После этого переменная i будет равна 4.
8. Снова проверяется условие i < 4. Теперь оно возвражает false, так как значение переменной i НЕ меньше 4, поэтому цикл завершает выполнение. Далее уже выполняется остальная часть программы, которая идет после цикла

В итоге блок цикла сработает 3 раза, пока значение i не станет равным 4. И каждый раз это значение будет увеличиваться на 1. Однократное выполнение блока цикла называется **итерацией**. Таким образом, здесь цикл выполнит три итерации. Результат работы программы:

1

2

3

Если блок цикла for содержит одну инструкцию, то мы можем его сократить, убрав фигурные свобки:

|  |  |
| --- | --- |
| 1  2  3  4  5 | for (int i = 1; i < 4; i++)      Console.WriteLine(i);    // или так  for (int i = 1; i < 4; i++) Console.WriteLine(i); |

При этом необязательно именно в первой части цикла объявлять переменную, а в третий части изменять ее значение - это могут быть любые действия. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | var i = 1;    for (Console.WriteLine("Начало выполнения цикла"); i < 4; Console.WriteLine($"i = {i}"))  {      i++;  } |

Здесь опять же цикл срабатывает, пока переменная i меньше 4, только приращение переменной i происходит в блоке цикла. Консольный вывод данной программы:

Начало выполнения цикла

i = 2

i = 3

i = 4

Нам необязательно указывать все условия при объявлении цикла. Например, мы можем написать так:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | int i = 1;  for (; ;)  {      Console.WriteLine($"i = {i}");      i++;  } |

Формально определение цикла осталось тем же, только теперь блоки в определении у нас пустые: for (; ;). У нас нет инициализированной переменной, нет условия, поэтому цикл будет работать вечно - бесконечный цикл.

Мы также можем опустить ряд блоков:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | int i = 1;  for (; i<4;)  {      Console.WriteLine($"i = {i}");      i++;  } |

Этот пример по сути эквивалентен первому примеру: у нас также есть переменная-счетчик, только определена она вне цикла. У нас есть условие выполнения цикла. И есть приращение переменной уже в самом блоке for.

Также стоит отметить, что можно определять несколько переменных в объявлении цикла:

|  |  |
| --- | --- |
| 1  2 | for (int i = 1, j = 1; i < 10; i++, j++)      Console.WriteLine($"{i \* j}"); |

Здесь в первой части объявления цикла определяются две переменных: i и j. Цикл выполняется, пока i не будет равна 10. После каждой итерации переменые i и j увеличиваются на единицу. Консольный вывод программы:

1

4

9

16

25

36

49

64

81

### **Цикл do..while**

В цикле do сначала выполняется код цикла, а потом происходит проверка условия в инструкции while. И пока это условие истинно, цикл повторяется.

|  |  |
| --- | --- |
| 1  2  3  4  5 | do  {      действия цикла  }  while (условие) |

Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | int i = 6;  do  {      Console.WriteLine(i);      i--;  }  while (i > 0); |

Здесь код цикла сработает 6 раз, пока i не станет равным нулю. Но важно отметить, что цикл do гарантирует хотя бы единократное выполнение действий, даже если условие в инструкции while не будет истинно. То есть мы можем написать:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | int i = -1;  do  {      Console.WriteLine(i);      i--;  }  while (i > 0); |

Хотя у нас переменная i меньше 0, цикл все равно один раз выполнится.

### **Цикл while**

В отличие от цикла do цикл **while** сразу проверяет истинность некоторого условия, и если условие истинно, то код цикла выполняется:

|  |  |
| --- | --- |
| 1  2  3  4 | while (условие)  {      действия цикла  } |

Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | int i = 6;  while (i > 0)  {      Console.WriteLine(i);      i--;  } |

### **Цикл foreach**

Цикл foreach предназначен для перебора набора или коллекции элементов. Его общее определение:

|  |  |
| --- | --- |
| 1  2  3  4 | foreach(тип\_данных переменная in коллекция)  {      // действия цикла  } |

После оператора foreach в скобках сначала идет определение переменной. Затем ключевое слово **in** и далее коллекция, элементы которой надо перебрать.

При выполнении цикл последовательно перебирает элементы коллекции и помещает их в переменную, и таким образом в блоке цикла мы можем выполнить с ними некоторые действия.

Например, возьмем строку. Строка по сути - это коллекция символов. И .NET позволяет перебрать все элементы строки - ее символы с помощью цикла **foreach**.

|  |  |
| --- | --- |
| 1  2  3  4 | foreach(char c in "Tom")  {      Console.WriteLine(c);  } |

Здесь цикл foreach пробегается по всем символам строки "Tom" и каждый символ помещает в символьную переменную c. В блоке цикла значение переменной c выводится на консоль. Поскольку в строке "Tom" три символа, то цикл выполнится три раза. Консольный вывод программы:

T

o

m

Стоит отметить, что определяемая в объявлении цикла должна по типу соответствовать типу элементов перебираемой коллекции. Так, элементы строки - значения типа char - символы. Поэтому переменная c имеет тип char. Однако в реальности не всегда бывает очевидно, какой тип представляют элементы коллекции. В этом случае мы можем определить переменную с помощью оператора **var**:

|  |  |
| --- | --- |
| 1  2  3  4 | foreach(var c in "Tom")  {      Console.WriteLine(c);  } |

В дальнейшем мы подробнее рассмотрим, что представляют собой коллекции в .NET и какие именно коллекции можно перебирать с помощью цикла foreach.

### **Операторы continue и break**

Иногда возникает ситуация, когда требуется выйти из цикла, не дожидаясь его завершения. В этом случае мы можем воспользоваться оператором **break**.

Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | for (int i = 0; i < 9; i++)  {      if (i == 5)          break;      Console.WriteLine(i);  } |

Хотя в условии цикла сказано, что цикл будет выполняться, пока счетчик i не достигнет значения 9, в реальности цикл сработает 5 раз. Так как при достижении счетчиком i значения 5, сработает оператор break, и цикл завершится.

0

1

2

3

4

Теперь поставим себе другую задачу. А что если мы хотим, чтобы при проверке цикл не завершался, а просто пропускал текущую итерацию. Для этого мы можем воспользоваться оператором **continue**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | for (int i = 0; i < 9; i++)  {      if (i == 5)          continue;      Console.WriteLine(i);  } |

В этом случае цикл, когда дойдет до числа 5, которое не удовлетворяет условию проверки, просто пропустит это число и перейдет к следующей итерации:

0

1

2

3

4

6

7

8

Стоит отметить, что операторы break и continue можно применять в любом типе циклов.

### Вложенные циклы

Одни циклы могут быть вложенными в другие. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | for (int i = 1; i < 10; i++)  {      for (int j = 1; j < 10; j++)      {          Console.Write($"{i \* j} \t");      }      Console.WriteLine();  } |

В данном случае цикл for (int i = 1; i < 10; i++) выполняется 9 раз, то есть имеет 9 итераций. Но в рамках каждой итерации выполняется девять раз вложенный цикл for (int j = 1; j < 10; j++). В итоге данная программа выведет таблицу умножения.

## Массивы

Массив представляет набор однотипных данных. Объявление массива похоже на объявление переменной за тем исключением, что после указания типа ставятся квадратные скобки:

|  |  |
| --- | --- |
| 1 | тип\_переменной[] название\_массива; |

Например, определим массив целых чисел:

|  |  |
| --- | --- |
| 1 | int[] numbers; |

После определения переменной массива мы можем присвоить ей определенное значение:

|  |  |
| --- | --- |
| 1 | int[] nums = new int[4]; |

Здесь вначале мы объявили массив nums, который будет хранить данные типа int. Далее используя операцию new, мы выделили память для 4 элементов массива: new int[4]. Число 4 еще называется **длиной массива**. При таком определении все элементы получают значение по умолчанию, которое предусмотренно для их типа. Для типа int значение по умолчанию - 0.

Также мы сразу можем указать значения для этих элементов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | int[] nums2 = new int[4] { 1, 2, 3, 5 };    int[] nums3 = new int[] { 1, 2, 3, 5 };    int[] nums4 = new[] { 1, 2, 3, 5 };    int[] nums5 = { 1, 2, 3, 5 }; |

Все перечисленные выше способы будут равноценны.

Подобным образом можно определять массивы и других типов, например, массив значений типа string:

|  |  |
| --- | --- |
| 1 | string[] people = { "Tom", "Sam", "Bob" }; |

### **Индексы и получение элементов массива**

Для обращения к элементам массива используются **индексы**. Индекс представляет номер элемента в массиве, при этом нумерация начинается с нуля, поэтому индекс первого элемента будет равен 0, индекс четвертого элемента - 3.

Используя индексы, мы можем получить элементы массива:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | int[] numbers = { 1, 2, 3, 5 };    // получение элемента массива  Console.WriteLine(numbers[3]);  // 5    // получение элемента массива в переменную  var n = numbers[1];     // 2  Console.WriteLine(n);  // 2 |

Также мы можем изменить элемент массива по индексу:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | int[] numbers = { 1, 2, 3, 5 };    // изменим второй элемент массива  numbers[1] = 505;    Console.WriteLine(numbers[1]);  // 505 |

И так как у нас массив определен только для 4 элементов, то мы не можем обратиться, например, к шестому элементу. Если мы так попытаемся сделать, то мы получим ошибку во время выполнения:

|  |  |
| --- | --- |
| 1  2  3 | int[] numbers = { 1, 2, 3, 5 };    Console.WriteLine(numbers[6]);  // ! Ошибка - в массиве только 4 элемента |

### **Свойство Length и длина массива**

каждый массив имеет свойство **Length**, которое хранит длину массива. Например, получим длину выше созданного массива numbers:

|  |  |
| --- | --- |
| 1  2  3 | int[] numbers = { 1, 2, 3, 5 };    Console.WriteLine(numbers.Length);  // 4 |

Для получения длины массива после названия массива через точку указывается свойство Length: numbers.Length.

### Получение элементов с конца массива

Благодаря наличию свойства Length, мы можем вычислить индекс последнего элемента массива - это длина массива - 1. Например, если длина массива - 4 (то есть массив имеет 4 элемента), то индекс последнего элемента будет равен 3. И, используя свойство Length, мы можем легко получить элементы с конца массива:

|  |  |
| --- | --- |
| 1  2  3  4  5 | int[] numbers = { 1, 2, 3, 5};    Console.WriteLine(numbers[numbers.Length - 1]);  // 5 - первый с конца или последний элемент  Console.WriteLine(numbers[numbers.Length - 2]);  // 3 - второй с конца или предпоследний элемент  Console.WriteLine(numbers[numbers.Length - 3]);  // 2 - третий элемент с конца |

Однако при подобном подходе выражения типа numbers.Length - 1, смысл которых состоит в том, чтобы получить какой-то определенный элемент с конца массива, утяжеляют код. И, начиная, с версии C# 8.0 в язык был добавлен специальный оператор **^**, с помощью которого можно задать индекс относительно конца коллекции.

Перепишем предыдущий пример, применяя оператор **^**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | int[] numbers = { 1, 2, 3, 5};    Console.WriteLine(numbers[^1]);  // 5 - первый с конца или последний элемент  Console.WriteLine(numbers[^2]);  // 3 - второй с конца или предпоследний элемент  Console.WriteLine(numbers[^3]);  // 2 - третий элемент с конца |

### **Перебор массивов**

Для перебора массивов мы можем использовать различные типы циклов. Например, цикл **foreach**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | int[] numbers = { 1, 2, 3, 4, 5 };  foreach (int i in numbers)  {      Console.WriteLine(i);  } |

Здесь в качестве контейнера выступает массив данных типа int. Поэтому мы объявляем переменную с типом int

Подобные действия мы можем сделать и с помощью цикл for:

|  |  |
| --- | --- |
| 1  2  3  4  5 | int[] numbers = { 1, 2, 3, 4, 5 };  for (int i = 0; i < numbers.Length; i++)  {      Console.WriteLine(numbers[i]);  } |

В то же время цикл **for** более гибкий по сравнению с **foreach**. Если **foreach** последовательно извлекает элементы контейнера и только для чтения, то в цикле for мы можем перескакивать на несколько элементов вперед в зависимости от приращения счетчика, а также можем изменять элементы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | int[] numbers = { 1, 2, 3, 4, 5 };  for (int i = 0; i < numbers.Length; i++)  {      numbers[i] = numbers[i] \* 2;      Console.WriteLine(numbers[i]);  } |

Также можно использовать и другие виды циклов, например, while:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **int**[] numbers = { 1, 2, 3, 4, 5 };  **int** i = 0;  **while**(i < numbers.Length)  {      Console.WriteLine(numbers[i]);      i++;  } |

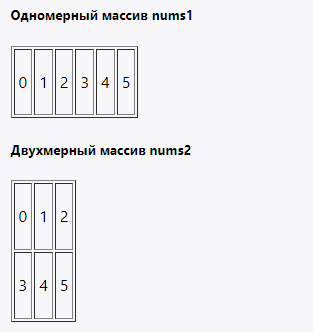
### **Многомерные массивы**

Массивы характеризуются таким понятием как **ранг** или количество измерений. Выше мы рассматривали массивы, которые имеют одно измерение (то есть их ранг равен 1) - такие массивы можно представлять в виде ряда (строки или столбца) элемента. Но массивы также бывают многомерными. У таких массивов количество измерений (то есть ранг) больше 1.

Массивы которые имеют два измерения (ранг равен 2) называют двухмерными. Например, создадим одномерный и двухмерный массивы, которые имеют одинаковые элементы:

|  |  |
| --- | --- |
| 1  2  3 | int[] nums1 = new int[] { 0, 1, 2, 3, 4, 5 };    int[,] nums2 = { { 0, 1, 2 }, { 3, 4, 5 } }; |

Визуально оба массива можно представить следующим образом:



Поскольку массив nums2 двухмерный, он представляет собой простую таблицу. Все возможные способы определения двухмерных массивов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | int[,] nums1;  int[,] nums2 = new int[2, 3];  int[,] nums3 = new int[2, 3] { { 0, 1, 2 }, { 3, 4, 5 } };  int[,] nums4 = new int[,] { { 0, 1, 2 }, { 3, 4, 5 } };  int[,] nums5 = new [,]{ { 0, 1, 2 }, { 3, 4, 5 } };  int[,] nums6 = { { 0, 1, 2 }, { 3, 4, 5 } }; |

Массивы могут иметь и большее количество измерений. Объявление трехмерного массива могло бы выглядеть так:

|  |  |
| --- | --- |
| 1 | int[,,] nums3 = new int[2, 3, 4]; |

Соответственно могут быть и четырехмерные массивы и массивы с большим количеством измерений. Но на практике обычно используются одномерные и двухмерные массивы.

Определенную сложность может представлять перебор многомерного массива. Прежде всего надо учитывать, что длина такого массива - это совокупное количество элементов.

|  |  |
| --- | --- |
| 1  2  3 | int[,] numbers = { { 1, 2, 3 }, { 4, 5, 6 }};  foreach (int i in numbers)      Console.Write($"{i} "); |

В данном случае длина массива numbers равна 6. И цикл foreach выводит все элементы массива в строку:

1 2 3 4 5 6

Но что если мы хотим отдельно пробежаться по каждой строке в таблице? В этом случае надо получить количество элементов в размерности. В частности, у каждого массива есть метод **GetUpperBound(номер\_размерности)**, который возвращает индекс последнего элемента в определенной размерности. И если мы говорим непосредственно о двухмерном массиве, то первая размерность (с индексом 0) по сути это и есть таблица. И с помощью выражения

|  |  |
| --- | --- |
| 1 | numbers.GetUpperBound(0) + 1 |

можно получить количество строк таблицы, представленной двухмерным массивом. А через

|  |  |
| --- | --- |
| 1 | numbers.Length / количество\_строк |

можно получить количество элементов в каждой строке:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | int[,] numbers = { { 1, 2, 3 }, { 4, 5, 6 }};    int rows = numbers.GetUpperBound(0) + 1;    // количество строк  int columns = numbers.Length / rows;        // количество столбцов  // или так  // int columns = numbers.GetUpperBound(1) + 1;    for (int i = 0; i < rows; i++)  {      for (int j = 0; j < columns; j++)      {          Console.Write($"{numbers[i, j]} \t");      }      Console.WriteLine();  } |

1 2 3

4 5 6

### Массив массивов

От многомерных массивов надо отличать **массив массивов** или так называемый "зубчатый массив":

|  |  |
| --- | --- |
| 1  2  3  4 | int[][] nums = new int[3][];  nums[0] = new int[2] { 1, 2 };          // выделяем память для первого подмассива  nums[1] = new int[3] { 1, 2, 3 };       // выделяем память для второго подмассива  nums[2] = new int[5] { 1, 2, 3, 4, 5 }; // выделяем память для третьего подмассива |

Здесь две группы квадратных скобок указывают, что это **массив массивов**, то есть такой массив, который в свою очередь содержит в себе другие массивы. Причем длина массива указывается только в первых квадратных скобках, все последующие квадратные скобки должны быть пусты: new int[3][]. В данном случае у нас массив nums содержит три массива. Причем размерность каждого из этих массивов может не совпадать.

Альтернативное определение массива массивов:

|  |  |
| --- | --- |
| 1  2  3  4  5 | int[][] numbers = {      new int[] { 1, 2 },      new int[] { 1, 2, 3 },      new int[] { 1, 2, 3, 4, 5 }  }; |

##### Зубчатый массив nums

|  |  |
| --- | --- |
| 1 | 2 |
| 1 | 2 | 3 |
| 1 | 2 | 3 | 4 | 5 |

Используя вложенные циклы, можно перебирать зубчатые массивы. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | int[][] numbers = new int[3][];  numbers[0] = new int[] { 1, 2 };  numbers[1] = new int[] { 1, 2, 3 };  numbers[2] = new int[] { 1, 2, 3, 4, 5 };  foreach(int[] row in numbers)  {      foreach(int number in row)      {          Console.Write($"{number} \t");      }      Console.WriteLine();  }    // перебор с помощью цикла for  for (int i = 0; i<numbers.Length;i++)  {      for (int j =0; j<numbers[i].Length; j++)      {          Console.Write($"{numbers[i][j]} \t");      }      Console.WriteLine();  } |

### Основные понятия массивов

Суммируем основные понятия массивов:

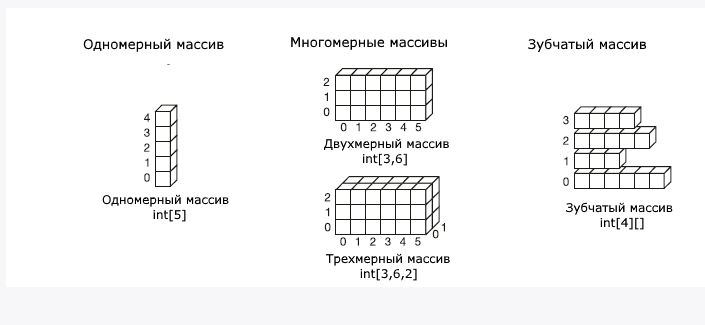
* **Ранг** (rank): количество измерений массива
* **Длина измерения** (dimension length): длина отдельного измерения массива
* **Длина массива** (array length): количество всех элементов массива

Например, возьмем массив

|  |  |
| --- | --- |
| 1 | int[,] numbers = new int[3, 4]; |

Массив numbers двухмерный, то есть он имеет два измерения, поэтому его ранг равен 2. Длина первого измерения - 3, длина второго измерения - 4. Длина массива (то есть общее количество элементов) - 12.

Примеры массивов:



**C# и .NET | Задачи с массивами**

Познакомившись с циклами, переменными, условными конструкциями и массивами, рассмотрим несколько задач для работы с массивами.

### Количество положительных чисел

Найдем количество положительных чисел в массиве:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | int[] numbers = { -4, -3, -2, -1, 0, 1, 2, 3, 4 };  int result = 0;  foreach(int number in numbers)  {      if(number > 0)      {          result++;      }  }  Console.WriteLine($"Число элементов больше нуля: {result}"); |

Здесь создаем вспомогательную переменную result, которая будет содержать количество положительных чисел. В цикле прохожим по массиву и, если его элемент больше нуля, добавляем к переменной result единицу.

### Инверсия массива

Вторая задача - инверсия массива, то есть переворот его в обратном порядке:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | int[] numbers = { -4, -3, -2, -1,0, 1, 2, 3, 4 };    int n = numbers.Length; // длина массива  int k = n / 2;          // середина массива  int temp;               // вспомогательный элемент для обмена значениями  for(int i=0; i < k; i++)  {      temp = numbers[i];      numbers[i] = numbers[n - i - 1];      numbers[n - i - 1] = temp;  }  foreach(int i in numbers)  {      Console.Write($"{i} \t");  } |

Поскольку нам надо изменять элементы массива, то для этого используется цикл for. Алгоритм решения задачи подразумевает перебор элементов до середины массива, которая в программе представлена переменной k, и обмен значений элемента, который имеет индекс i, и элемента с индексом n-i-1.

### Программа сортировки массива

Теперь возьмем задачу посложнее - простейшую сортировку массива:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | int[] nums = { 54, 7, -41, 2, 4, 2, 89, 33, -5, 12 };    // сортировка  int temp;  for (int i = 0; i < nums.Length - 1; i++)  {      for (int j = i + 1; j < nums.Length; j++)      {          if (nums[i] > nums[j])          {              temp = nums[i];              nums[i] = nums[j];              nums[j] = temp;          }      }  }    // вывод  Console.WriteLine("Вывод отсортированного массива");  for (int i = 0; i < nums.Length; i++)  {      Console.WriteLine(nums[i]);  } |

Для сортировки массива выполняем проходы по массиву и сравниваем элементы. Поскольку нам надо последовательно сравнивать каждый элемент массива с каждым (за исключением сравния с самим собой), то здесь применятся вложенный цикл.

Во внешнем цикле мы берем элемент, который будем сравнивать:

|  |  |
| --- | --- |
| 1 | for (int i = 0; i < nums.Length - 1; i++) |

Далее запускаем вложенный цикл, который начинается, со следующего элемента, и из которого извлекаем элементы, с которыми будем сравнивать тот элемент, которые берется из массива во внешнем цикле:

|  |  |
| --- | --- |
| 1 | for (int j = i + 1; j < nums.Length; j++) |

Если элемент с меньшим индексом больше элемента с большим индексом, то меняем элементы местами.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | if (nums[i] > nums[j])  {      temp = nums[i];      nums[i] = nums[j];      nums[j] = temp;  } |

В конце выводим все элементы.

## Методы

Если переменные хранят некоторые значения, то методы содержат собой набор инструкций, которые выполняют определенные действия. По сути метод - это именованный блок кода, который выполняет некоторые действия.

Общее определение методов выглядит следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4 | [модификаторы] тип\_возвращаемого\_значения название\_метода ([параметры])  {      // тело метода  } |

Модификаторы и параметры необязательны.

Ранее мы уже использовали как минимум один метод - **Console.WriteLine()**, который выводит информацию на консоль. Теперь рассмотрим, как мы можем создавать свои методы.

### **Определение метода**

Определим один метод:

|  |  |
| --- | --- |
| 1  2  3  4 | void SayHello()  {      Console.WriteLine("Hello");  } |

Здесь определен метод SayHello, который выводит некоторое сообшение. К названиям методов предъявляются в принципе те же требования, что и к названиям переменных. Однако, как правило, названия методов начинаются с большой буквы.

Перед названием метода идет возвращаемый тип данных. Здесь это тип **void**, который указыает, что фактически ничего не возвращает, он просто производит некоторые действия.

После названия метода в скобках идет перечисление параметров. Но в данном случае скобки пустые, что означает, что метод не принимает никаких параметров.

После списка параметров в круглых скобках идет блок кода, который представляет набор выполняемых методом инструкций. В данном случае блок метода SayHello содержит только одну инструкцию, которая выводит строку на консоль:

|  |  |
| --- | --- |
| 1 | Console.WriteLine("Hello"); |

Но если мы запустим данный проект, то мы не увидим никакой строки, которую должен выводить метод SayHello. Потому что после определения метод еще надо вызвать, чтобы он выполнил свою работу.

### **Вызов методов**

Чтобы использовать метод SayHello, нам надо его вызвать. Для вызова метода указывается его имя, после которого в скобках идут значения для его параметров (если метод принимает параметры).

|  |  |
| --- | --- |
| 1 | название\_метода (значения\_для\_параметров\_метода); |

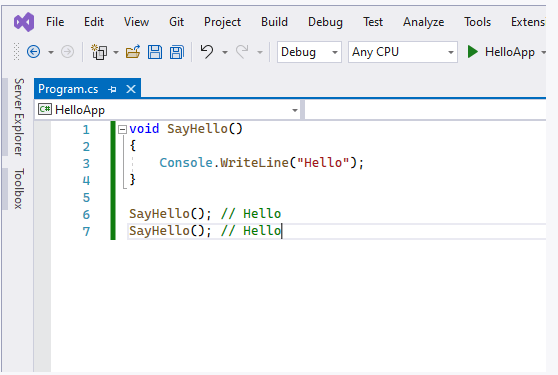
Например, вызов метода SayHello будет выглядеть следующим образом:

|  |  |
| --- | --- |
| 1 | SayHello(); |

Поскольку метод не принимает никаких параметров, то после названия метода идут пустые скобки.

Объединим определение и вызов метода:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | void SayHello()  {      Console.WriteLine("Hello");  }    SayHello(); // Hello  SayHello(); // Hello |



Консольный вывод программы:

Hello

Hello

Преимуществом методов является то, что их можно повторно и многократно вызывать в различных частях программы. Например, в примере выше два раза вызывается метод SayHello.

При этом в данном случае нет разницы, сначала определяется метод, а потом вызывается или наоборот. Например, мы могли бы написать и так:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | SayHello(); // Hello  SayHello(); // Hello    void SayHello()  {      Console.WriteLine("Hello");  } |

Определим и вызовем еще несколько методов:

|  |  |
| --- | --- |
| 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 | void SayHelloRu()  {      Console.WriteLine("Привет");  }  void SayHelloEn()  {      Console.WriteLine("Hello");  }  void SayHelloFr()  {      Console.WriteLine("Salut");  }      string language = "en";    switch (language)  {      case "en":          SayHelloEn();          break;      case "ru":          SayHelloRu();          break;      case "fr":          SayHelloFr();          break;  } |

Здесь определены три метода SayHelloRu(), SayHelloEn() и SayHelloFr(), которые также имеют тип void, не принимают никаких параметров и также выводит некоторую строку на консоль. Условно говоря, они выводят приветствие на определенном языке.

В конструкции switch проверяется значение переменной language, которая условно хранит код языка, и в зависимости от ее значения вызывается определенный метод. Так, в данном случае на консоль будет выведено

Hello

### **Сокращенная запись методов**

Если метод в качестве тела определяет только одну инструкцию, то мы можем сократить определение метода. Например, допустим у нас есть метод:

|  |  |
| --- | --- |
| 1  2  3  4 | void SayHello()  {      Console.WriteLine("Hello");  } |

Мы можем его сократить следующим образом:

|  |  |
| --- | --- |
| 1 | void SayHello() => Console.WriteLine("Hello"); |

То есть после списка параметров ставится оператор **=>**, после которого идет выполняемая инструкция.

## Параметры методов

В прошлой теме был определен метод SayHello, который выводит на консоль некоторое сообщение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | void SayHello()  {      Console.WriteLine("Hello");  }    SayHello(); // Hello |

Но минусом подобного метода является то, что он выводит одно и то же сообщение. И было бы неплохо, если бы мы могли бы динамически определять, какое сообщение будет выводить метод на экран, то есть передать из вне в метод это сообщение. Для этого в языке C# мы можем использовать параметры.

**Параметры** позволяют передать в метод некоторые входные данные. Параметры определяются через заятую в скобках после названия метода в виде:

|  |  |
| --- | --- |
| 1  2  3  4 | тип\_метода имя\_метода (тип\_параметра1 параметр1, тип\_параметра2 параметр2, ...)  {      // действия метода  } |

Определение параметра состоит из двух частей: сначала идет тип параметра и затем его имя.

Например, определим метод PrintMessage, который получает извне выводимое сообщение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | void PrintMessage(string message)  {      Console.WriteLine(message);  }    PrintMessage("Hello work");           // Hello work  PrintMessage("Hello METANIT.COM");    // Hello METANIT.COM  PrintMessage("Hello C#");             // Hello C# |

Здесь метод PrintMessage() принимает один параметр, который называется message и имеет тип string.

Чтобы выполнить метод, который имеет параметры, при вызове после имени метода в скобках ему передаются значения для его параметров, например:

|  |  |
| --- | --- |
| 1 | PrintMessage("Hello work"); |

Здесь параметру message передается строка "Hello work". Значения, которые передаются параметрам, еще называются **аргументами**. То есть передаваемая строка "Hello work" в данном случае является аргументом.

Иногда можно встретить такие определения как **формальные параметры** и **фактические параметры**. Формальные параметры - это собственно параметры метода (в данном случае message), а фактические параметры - значения, которые передаются формальным параметрам. То есть фактические параметры - это и есть аргументы метода.

Определим еще один метод, который складывает два числа:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | void Sum(int x, int y)  {      int result = x + y;      Console.WriteLine($"{x} + {y} = {result}");  }    Sum(10, 15);    // 10 + 15 = 25 |

Метод Sum имеет два параметра: x и y. Оба параметра представляют тип int. Поэтому при вызове данного метода нам обязательно надо передать на место этих параметров два числа. Внутри метода вычисляется сумма переданных чисел и выводится на консоль.

При вызове метода Sum значения передаются параметрам по позиции. Например, в вызове Sum(10, 15) число 10 передается параметру x, а число 15 - параметру y.

Также параметры могут использоваться в сокращеной версии метода:

|  |  |
| --- | --- |
| 1  2  3 | void Sum(int x, int y) => Console.WriteLine($"{x} + {y} = { x + y }");    Sum(10, 15);    // 10 + 15 = 25 |

Передаваемые параметру значения могут представлять значения переменных или результат работы сложных выражений, которые возвращают некоторое значение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | void Sum(int x, int y) => Console.WriteLine($"{x} + {y} = { x + y }");    int a = 10, b = 15, c = 6;  Sum(a, b);                  // 10 + 15 = 25  Sum(3, c);                  // 3 + 6 = 9  Sum(14, 4 + c);             // 14 + 10 = 24 |

Если параметрами метода передаются значения переменных, то таким переменным должно быть присвоено значение. Например, следующая программа не скомпилируется:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | void Sum(int x, int y)  {      Console.WriteLine($"{x} + {y} = { x + y }");  }    int a;  int b = 15;  Sum(a, b);  // ! Ошибка |

### Соответствие параметов и аргументов по типу данных

При передаче значений параметрам важно учитывать тип параметров: между аргументами и параметрами должно быть соответствие по типу. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | void PrintPerson(string name, int age)  {      Console.WriteLine($"Name: {name}  Age: {age}");  }    PrintPerson("Tom", 24); // Name: Tom  Age: 24 |

В данном случае первый параметр метода PrintPerson() представляет тип string, поэтому при вызове метода мы должны передать этому параметру значение типа string, то есть строку. Второй параметр представляет тип int, поэтому должны передать ему целое число, которое соответствует типу int.

|  |  |
| --- | --- |
| 1 | PrintPerson("Tom", 24); |

Также мы можем передать параметрам значения тех типов, которые автоматически могут быть преобразованы в тип параметров. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | void PrintPerson(string name, int age)  {      Console.WriteLine($"Name: {name}  Age: {age}");  }    byte b = 37;  PrintPerson("Tom", b); // Name: Tom  Age: 37 |

Здесь параметру типа int передается значение типа byte, но комилятор может автоматически преобразовать значение типа byte к тиу int. Поэтому здесь ошибки не возникнет. Какие преобразования типов могут быть выполнены автоматически, рассматривалось в одной из предыдущих тем: [Преобразования базовых типов данных](https://metanit.com/sharp/tutorial/2.2.php)

Данные других типов мы передать параметров не можем. Например, следующий вызов метода PrintPerson будет ошибочным:

|  |  |
| --- | --- |
| 1 | PrintPerson(45, "Bob"); // Ошибка! несоответствие значений типам параметров |

### **Необязательные параметры**

По умолчанию при вызове метода необходимо предоставить значения для всех его параметров. Но C# также позволяет использовать необязательные параметры. Для таких параметров нам необходимо объявить значение по умолчанию. Также следует учитывать, что после необязательных параметров все последующие параметры также должны быть необязательными:

|  |  |
| --- | --- |
| 1  2  3  4 | void PrintPerson(string name, int age = 1, string company = "Undefined")  {      Console.WriteLine($"Name: {name}  Age: {age}  Company: {company}");  } |

Здесь параметры age и company являются необязательными, так как им присвоены значения. Поэтому при вызове метода мы можем не передавать для них данные:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | void PrintPerson(string name, int age = 1, string company = "Undefined")  {      Console.WriteLine($"Name: {name}  Age: {age}  Company: {company}");  }    PrintPerson("Tom", 37, "Microsoft");  // Name: Tom  Age: 37  Company: Microsoft  PrintPerson("Tom", 37);               // Name: Tom  Age: 37  Company: Undefined  PrintPerson("Tom");                   // Name: Tom  Age: 1   Company: Undefined |

Консольный вывод программы:

Name: Tom Age: 37 Company: Microsoft

Name: Tom Age: 37 Company: Undefined

Name: Tom Age: 1 Company: Undefined

### **Именованные параметры**

В предыдущих примерах при вызове методов значения для параметров передавались в порядке объявления этих параметров в методе. То есть аргументы передавались параметрам **по позиции**. Но мы можем нарушить подобный порядок, используя именованные параметры:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | void PrintPerson(string name, int age = 1, string company = "Undefined")  {      Console.WriteLine($"Name: {name}  Age: {age}  Company: {company}");  }    PrintPerson("Tom", company:"Microsoft", age: 37);  // Name: Tom  Age: 37  Company: Microsoft  PrintPerson(age:41, name: "Bob");          // Name: Bob  Age: 41  Company: Undefined  PrintPerson(company:"Google", name:"Sam"); // Name: Sam  Age: 1   Company: Google |

Для передачи значений параметрам о имени при вызове метода указывается имя параметра и через двоеточие его значение: name:"Tom"

Консольный вывод программы:

Name: Tom Age: 37 Company: Microsoft

Name: Bob Age: 41 Company: Undefined

Name: Sam Age: 1 Company: Google

## Возвращение значения и оператор return

|  |  |
| --- | --- |
| 1  2  3  4 | string GetMessage()  {      return "Hello";  } |

Метод GetMessage имеет тип string, следовательно, он должен возвратить строку. Поэтому в теле метода используется оператор **return**, после которого указана возвращаемая строка.

При этом методы, которые в качестве возвращаемого типа имеют любой тип, кроме **void**, обязательно должны использовать оператор return для возвращения значения. Например, следующее определение метода некорректно:

|  |  |
| --- | --- |
| 1  2  3  4 | string GetMessage()  {      Console.WriteLine("Hello");  } |

Также между возвращаемым типом метода и возвращаемым значением после оператора return должно быть соответствие. Например, в следующем случае возвращаемый тип - string, но метод возвращает число (тип int), поэтому такое определение метода некорректно:

|  |  |
| --- | --- |
| 1  2  3  4 | string GetMessage()  {      return 3;   // Ошибка! Метод должен возвращать строку, а не число  } |

Результат методов, который возвращают значение, мы можем присвоить переменным или использовать иным образом в программе:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | string GetMessage()  {      return "Hello";  }    string message = GetMessage();  // получаем результат метода в переменную message  Console.WriteLine(message);     // Hello |

Метод GetMessage() возвращает значение типа string. Поэтому мы можем присвоить это значение какой-нибудь переменной типа string: string message = GetMessage();

Либо даже передать в качестве значения параметру другого метода:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | string GetMessage()  {      return "Hello";  }  void PrintMessage(string message)  {      Console.WriteLine(message);  }  PrintMessage(GetMessage()); |

В вызове PrintMessage(GetMessage()) сначада вызывается метод GetMessage() и его результат передается параметру message метода PrintMessage

После оператора return также можно указывать сложные выражения или вызовы других методов, которые возвращают определенный результат. Например, определим метод, который возвращает сумму чисел:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | int Sum(int x, int y)  {      return x + y;  }    int result = Sum(10, 15);   // 25  Console.WriteLine(result);   // 25    Console.WriteLine(Sum(5, 6));   // 11 |

Метод Sum() имеет тип int, следовательно, он должен возвратить значение типа int - целое число. Поэтому в теле метода используется оператор **return**, после которого указано возвращаемое число (в данном случае результат суммы переменных x и y).

### Сокращенная версия методов с результатом

Также мы можем сокращать методы, которые возвращают значение:

|  |  |
| --- | --- |
| 1  2  3  4 | string GetMessage()  {      return "hello";  } |

аналогичен следующему методу:

|  |  |
| --- | --- |
| 1 | string GetMessage() => "hello"; |

А метод

|  |  |
| --- | --- |
| 1  2  3  4 | int Sum(int x, int y)  {      return x + y;  } |

аналогичен следующему методу:

|  |  |
| --- | --- |
| 1 | int Sum(int x, int y) => x + y; |

### Выход из метода

Оператор return не только возвращает значение, но и производит выход из метода. Поэтому он должен определяться после остальных инструкций. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5 | string GetHello()  {      return "Hello";      Console.WriteLine("After return");  } |

С точки зрения синтаксиса данный метод корректен, однако его инструкция Console.WriteLine("After return") не имеет смысла - она никогда не выполнится, так как до ее выполнения оператор **return** возвратит значение и произведет выход из метода.

Однако мы можем использовать оператор **return** и в методах с типом void. В этом случае после оператора return не ставится никакого возвращаемого значения (ведь метод ничего не возвращает). Типичная ситуация - в зависимости от опеределенных условий произвести выход из метода:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | void PrintPerson(string name, int age)  {      if(age > 120 || age < 1)      {          Console.WriteLine("Недопустимый возраст");          return;      }      Console.WriteLine($"Имя: {name}  Возраст: {age}");  }    PrintPerson("Tom", 37);         // Имя: Tom  Возраст: 37  PrintPerson("Dunkan", 1234);    // Недопустимый возраст |

Здесь метод PrintPerson() в качестве параметров принимает имя и возраст пользователя. Однако в методе вначале мы проверяем, соответствует ли возраст некоторому диапазону (меньше 120 и больше 0). Если возраст находится вне этого диапазона, то выводим сообщение о недопустимом возрасте и с помощью оператора **return** выходим из метода. После этого метод заканчивает свою работу.

Однако если возраст корректен, то выводим информацию о пользователе на консоль. Консольный вывод:

Имя: Tom Возраст: 37

Недопустимый возраст

## Передача параметров по ссылке и значению. Выходные параметры

Существует два способа передачи параметров в метод в языке C#: **по значению** и **по ссылке**.

### Передача параметров по значению

Наиболее простой способ передачи параметров представляет передача по значению, по сути это обычный способ передачи параметров:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | void Increment(int n)  {      n++;      Console.WriteLine($"Число в методе Increment: {n}");  }    int number = 5;  Console.WriteLine($"Число до метода Increment: {number}");  Increment(number);  Console.WriteLine($"Число после метода Increment: {number}"); |

Консольный вывод:

Число до метода Increment: 5

Число в методе Increment: 6

Число после метода Increment: 5

При передаче аргументов параметрам по значению параметр метода получает не саму переменную, а ее копию и далее работает с этой копией независимо от самой переменной.

Так, выше при вызове метод Increment получает копию переменной number и увеличивает значение этой копии. Поэтому в самом методе Increment мы видим, что значение параметра n увеличилось на 1, но после выполнения метода переменная number имеет прежнее значение - 5. То есть изменяется копия, а сама переменная не изменяется.

### **Передача параметров по ссылке и модификатор ref**

При передаче параметров по ссылке перед параметрами используется модификатор **ref**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | void Increment(ref int n)  {      n++;      Console.WriteLine($"Число в методе Increment: {n}");  }    int number = 5;  Console.WriteLine($"Число до метода Increment: {number}");  Increment(ref number);  Console.WriteLine($"Число после метода Increment: {number}"); |

Консольный вывод:

Число до метода Increment: 5

Число в методе Increment: 6

Число после метода Increment: 6

При передаче значений параметрам по ссылке метод получает адрес переменной в памяти. И, таким образом, если в методе изменяется значение параметра, передаваемого по ссылке, то также изменяется и значение переменной, которая передается на его место..

Так, в метод Increment передается ссылка на саму переменную number в памяти. И если значение параметра n в Increment изменяется, то это приводит и к изменению переменной number, так как и параметр n и переменная number указывают на один и тот же адрес в памяти.

Обратите внимание, что модификатор ref указывается как перед параметром при объявлении метода, так и при вызове метода перед аргументом, который передается параметру.

### Выходные параметры. Модификатор out

Выше мы использовали входные параметры. Но параметры могут быть также выходными. Чтобы сделать параметр выходным, перед ним ставится модификатор **out**:

|  |  |
| --- | --- |
| 1  2  3  4 | void Sum(int x, int y, out int result)  {      result = x + y;  } |

Здесь результат возвращается не через оператор return, а через выходной параметр result. Использование в программе:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | void Sum(int x, int y, out int result)  {      result = x + y;  }    int number;    Sum(10, 15, out number);    Console.WriteLine(number);   // 25 |

Причем, как и в случае с **ref** ключевое слово **out** используется как при определении метода, так и при его вызове.

Также обратите внимание, что методы, использующие такие параметры, обязательно должны присваивать им определенное значение. То есть следующий код будет недопустим, так как в нем для out-параметра не указано никакого значения:

|  |  |
| --- | --- |
| 1  2  3  4 | void Sum(int x, int y, out int result)  {      Console.WriteLine(x + y);  } |

Прелесть использования подобных параметров состоит в том, что по сути мы можем вернуть из метода не одно значение, а несколько. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | void GetRectangleData(int width, int height, out int rectArea, out int rectPerimetr)  {      rectArea = width \* height;       // площадь прямоугольника - произведение ширины на высоту      rectPerimetr = (width + height) \* 2; // периметр прямоугольника - сумма длин всех сторон  }    int area;  int perimetr;    GetRectangleData(10, 20, out area, out perimetr);    Console.WriteLine($"Площадь прямоугольника: {area}");       // 200  Console.WriteLine($"Периметр прямоугольника: {perimetr}");   // 60 |

Здесь у нас есть метод GetRectangleData, который получает ширину и высоту прямоугольника (параметры width и height). А два выходных параметра мы используем для подсчета площади и периметра прямоугольника.

При этом можно определять переменные, которые передаются **out**-параметрам в непосредственно при вызове метода. То есть мы можем сократить предыдущий пример следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | void GetRectangleData(int width, int height, out int rectArea, out int rectPerimetr)  {      rectArea = width \* height;      rectPerimetr = (width + height) \* 2;  }    GetRectangleData(10, 20, out int area, out int perimetr);    Console.WriteLine($"Площадь прямоугольника: {area}");       // 200  Console.WriteLine($"Периметр прямоугольника: {perimetr}");   // 60 |

При этом, если нам неизвестен тип значений, которые будут присвоены параметрам, то мы можем для их определения использовать оператор **var**:

|  |  |
| --- | --- |
| 1  2  3  4 | GetRectangleData(10, 20, out var area, out var perimetr);    Console.WriteLine($"Площадь прямоугольника: {area}");       // 200  Console.WriteLine($"Периметр прямоугольника: {perimetr}");   // 60 |

### Входные параметры. Модификатор in

Кроме выходных параметров с модификатором out метод может использовать входные параметры с модификатором **in**. Модификатор **in** указывает, что данный параметр будет передаваться в метод по ссылке, однако внутри метода его значение параметра нельзя будет изменить. Например, возьмем следующий метод:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | void GetRectangleData(in int width, in int height, out int rectArea, out int rectPerimetr)  {      //width = 25; // нельзя изменить, так как width - входной параметр      rectArea = width \* height;      rectPerimetr = (width + height) \* 2;  }    int w = 10;  int h = 20;  GetRectangleData(w, h, out var area, out var perimetr);    Console.WriteLine($"Площадь прямоугольника: {area}");       // 200  Console.WriteLine($"Периметр прямоугольника: {perimetr}");   // 60 |

В данном случае через входные параметры width и height в метод передаются значения, но в самом методе мы не можем изменить значения этих параметров, так как они определены с модификатором **in**.

Передача по ссылке в некоторых случаях может увеличить произодительность, а использование оператора **in**, что переданные параметрам переменне ельзя будет изменить в этом методе.

## Массив параметров и ключевое слово params

Во всех предыдущих примерах мы использовали постоянное число параметров. Но, используя ключевое слово **params**, мы можем передавать неопределенное количество параметров:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | void Sum(params int[]  numbers)  {      int result = 0;      foreach (var n in numbers)      {          result += n;      }      Console.WriteLine(result);  }    int[] nums = { 1, 2, 3, 4, 5};  Sum(nums);  Sum(1, 2, 3, 4);  Sum(1, 2, 3);  Sum(); |

Сам параметр с ключевым словом **params** при определении метода должен представлять одномерный массив того типа, данные которого мы собираемся использовать. При вызове метода на место параметра с модификатором **params** мы можем передать как отдельные значения, так и массив значений, либо вообще не передавать параметры. Количество передаваемых значений в метод неопределено, однако все эти значения должны соответствовать типу параметра с **params**.

Если же нам надо передать какие- то другие параметры, то они должны указываться до параметра с ключевм словом params:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | void Sum(int initialValue, params int[]  numbers)  {      int result = initialValue;      foreach (var n in numbers)      {          result += n;      }      Console.WriteLine(result);  }    int[] nums = { 1, 2, 3, 4, 5};  Sum(10, nums);  // число 10 - передается параметру initialValue  Sum(1, 2, 3, 4);  Sum(1, 2, 3);  Sum(20); |

Здесь метод Sum имеет обязательный параметр initialValue, поэтому при вызове метода для него нужно обязательно передать значение. Поэтому первое значение при вызове метода будет передаваться этому параметру.

Однако после параметра с модификатором **params** мы НЕ можем указывать другие параметры. То есть следующее определение метода недопустимо:

|  |  |
| --- | --- |
| 1  2  3 | //Так НЕ работает  void Sum(params int[] numbers, int initialValue)  {} |

### Массив в качестве параметра

Также этот способ передачи параметров надо отличать от передачи массива в качестве параметра:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | void Sum(int[] numbers, int initialValue)  {      int result = initialValue;      foreach (var n in numbers)      {          result += n;      }      Console.WriteLine(result);  }    int[] nums = { 1, 2, 3, 4, 5};  Sum(nums, 10);    // Sum(1, 2, 3, 4);     // так нельзя - нам надо передать массив |

Так как метод Sum принимает в качестве параметра массив без ключевого слова params, то при его вызове нам обязательно надо передать в качестве первого параметра массив. Кроме того, в отличие от метода с параметром **params** после параметра-массива могут располагаться другие параметры.

## Рекурсивные функции

Отдельно остановимся на рекурсивных функциях. Рекурсивная функция представляет такую конструкцию, при которой функция вызывает саму себя.

### Рекурсивная функция факториала

Возьмем, к примеру, вычисление факториала, которое использует формулу **n! = 1 \* 2 \* … \* n**. То есть по сути для нахождения факториала числа мы перемножаем все числа до этого числа. Например, факториал числа 4 равен 24 = 1 \* 2 \* 3 \* 4, а факторил числа 5 равен 120 = 1 \* 2 \* 3 \* 4 \* 5.

Определим метод для нахождения факториала:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | int Factorial(int n)  {      if (n == 1) return 1;        return n \* Factorial(n - 1);  } |

При создании рекурсивной функции в ней обязательно должен быть некоторый **базовый вариант**, с которого начинается вычисление функции. В случае с факториалом это факториал числа 1, который равен 1. Факторилы все осальных положительных чисел будет начинаться с вычисления факториала числа 1, который равен 1.

На уровне языка программирования для возвращения базового варианта применяется оператор **return**:

|  |  |
| --- | --- |
| 1 | if (n == 1) return 1; |

То есть, если вводимое число равно 1, то возвращается 1

Другая особенность рекурсивных функций: все рекурсивные вызовы должны обращаться к подфункциям, которые в конце концов сходятся к базовому варианту:

|  |  |
| --- | --- |
| 1 | return n \* Factorial(n - 1); |

Так, при передаче в функцию числа, которое не равно 1, при дальнейших рекурсивных вызовах подфункций в них будет передаваться каждый раз число, меньшее на единицу. И в конце концов мы дойдем до ситуации, когда число будет равно 1, и будет использован базовый вариант. Это так называемый рекурсивный спуск.

Используем эту функцию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | int Factorial(int n)  {      if (n == 1) return 1;        return n \* Factorial(n - 1);  }    int factorial4 = Factorial(4);  // 24  int factorial5 = Factorial(5);  // 120  int factorial6 = Factorial(6);  // 720    Console.WriteLine($"Факториал числа 4 = {factorial4}");  Console.WriteLine($"Факториал числа 5 = {factorial5}");  Console.WriteLine($"Факториал числа 6 = {factorial6}"); |

Рассмотрим поэтапно, что будет в случае вызова Factorial(4).

1. Сначала идет проверка, равно ли число единице:

|  |  |
| --- | --- |
| 1 | if (n == 1) return 1; |

1. Но вначале n равно 4, поэтому это условие ложно, и соответственно выполняется код

|  |  |
| --- | --- |
| 1 | return n \* Factorial(n - 1); |

1. То есть фактически мы имеем:

|  |  |
| --- | --- |
| 1 | return 4 \* Factorial(3); |

1. Далее выполняется выражение:

|  |  |
| --- | --- |
| 1 | Factorial(3) |

1. Опять же n не равно 1, поэтому выполняется код

|  |  |
| --- | --- |
| 1 | return n \* Factorial(n - 1); |

1. То есть фактически:

|  |  |
| --- | --- |
| 1 | return 3 \* Factorial(2); |

1. Далее выполняется выражение:

|  |  |
| --- | --- |
| 1 | Factorial(2) |

1. Опять же n не равно 1, поэтому выполняется код

|  |  |
| --- | --- |
| 1 | return n \* Factorial(n - 1); |

1. То есть фактически:

|  |  |
| --- | --- |
| 1 | return 2 \* Factorial(1); |

1. Далее выполняется выражение:

|  |  |
| --- | --- |
| 1 | Factorial(1) |

1. Теперь n равно 1, поэтому выполняется код

|  |  |
| --- | --- |
| 1 | if (n == 1) return 1; |

1. И возвращается 1.

В итоге выражение

|  |  |
| --- | --- |
| 1 | Factorial(4) |

В реальности выливается в

|  |  |
| --- | --- |
| 1 | 4 \* 3 \* 2 \* Factorial(1) |

### Рекурсивная функция Фибоначчи

Другим распространенным показательным примером рекурсивной функции служит функция, вычисляющая числа Фибоначчи. n-й член последовательности Фибоначчи определяется по формуле: f(n)=f(n-1) + f(n-2), причем f(0)=0, а f(1)=1. То есть последовательность Фибоначчи будет выглядеть так 0 (0-й член), 1 (1-й член), 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, .... Для определения чисел этой последовательности определим следующий метод:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | int Fibonachi(int n)  {      if (n == 0 || n == 1) return n;        return Fibonachi(n - 1) + Fibonachi(n - 2);  }    int fib4 = Fibonachi(4);  int fib5 = Fibonachi(5);  int fib6 = Fibonachi(6);    Console.WriteLine($"4 число Фибоначчи = {fib4}");  Console.WriteLine($"5 число Фибоначчи = {fib5}");  Console.WriteLine($"6 число Фибоначчи = {fib6}"); |

Здесь базовый вариант выглядит следующий образом:

|  |  |
| --- | --- |
| 1 | if (n == 0 || n == 1) return n; |

То есть, если мы ищем нулевой или первый элемент последовательности, то возвращается это же число - 0 или 1. Иначе возвращается результат выражения Fibonachi(n - 1) + Fibonachi(n - 2);

### Рекурсии и циклы

Это простейшие пример рекурсивных функций, которые призваны дать понимание работы рекурсии. В то же время для обоих функций вместо рекурсий можно использовать циклические конструкции. И, как правило, альтернативы на основе циклов работают быстрее и более эффективны, чем рекурсия. Например, вычисление чисел Фибоначчи с помощью циклов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | static int Fibonachi2(int n)  {      int result = 0;      int b = 1;      int tmp;        for (int i = 0; i < n; i++)      {          tmp = result;          result = b;          b += tmp;      }        return result;  } |

В то же время в некоторых ситуациях рекурсия предоставляет элегантное решение, например, при обходе различных древовидных представлений, к примеру, дерева каталогов и файлов.

## Локальные функции

Локальные функции представляют функции, определенные внутри других методов. Локальная функция, как правило, содержит действия, которые применяются только в рамках ее метода.

Например, определим метод который сравнивают сумму чисел двух массивов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | void Compare(int[] numbers1, int[] numbers2)  {      int numbers1Sum = 0;      int numbers2Sum = 0;        foreach(int number in numbers1)          numbers1Sum += number;        foreach (int number in numbers2)          numbers2Sum += number;        if (numbers1Sum > numbers2Sum)          Console.WriteLine("сумма чисел из массива numbers1 больше");      else          Console.WriteLine("сумма чисел из массива numbers2 больше");  }    int[] numbers1 = { 1, 2, 3 };  int[] numbers2 = { 3, 4, 5, 6, 7 };    Compare(numbers1, numbers2); |

Здесь метод Compare принимает два массива и последовательно вычисляет сумму их элементов, чтобы узнать в каком массиве сумма чисел больше. Несмотря на то, что все работает, здесь есть один недостаток: здесь повторяется действия, которые вычисляют сумму массива:

|  |  |
| --- | --- |
| 1  2  3  4 | int numbers1Sum = 0;    foreach(int number in numbers1)      numbers1Sum += number; |

К тому а что, если мы захотим сравнивать сумму только положительных или четных чисел или как-то иначе изменить логику сравнения? В этом лучше вынести повторяющиеся действия в отдельный метод. Однако если эти действия нигде больше в прогамме не будут вызываться и будут использоваться только в одном методе, то целесообразно определить эти действия в виде локальной функции. Для этого изменим метод Compare следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | void Compare(int[] numbers1, int[] numbers2)  {      int numbers1Sum = Sum(numbers1);      int numbers2Sum = Sum(numbers2);        if (numbers1Sum > numbers2Sum)          Console.WriteLine("сумма чисел из массива numbers1 больше");      else          Console.WriteLine("сумма чисел из массива numbers2 больше");        int Sum(int[] numbers)      {          int result = 0;          foreach (int number in numbers)              result += number;          return result;      }  }    int[] numbers1 = { 1, 2, 3 };  int[] numbers2 = { 3, 4, 5, 6, 7 };    Compare(numbers1, numbers2); |

Здесь подсчет суммы вынесен в локальную функцию Sum, которая принимает массив и возвращает его сумму. И далее в рамках метода Compare мы сможем ее использовать для вычисления суммы массива. При этом неважно, определена локальная функция до или после использования. Но вне ее метода локальная функция не может использоваться.

### **Статические локальные функции**

Локлальные функции могут быть статическими. Такие функции определяются с помощью ключевого слова **static**. Их особенностью является то, что они не могут обращаться к переменным окружения, то есть метода, в котором статическая функция определена.

Сначала определим локальную функцию, которая имеет доступ окружению:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | int Sum(int[] numbers)  {      int limit = 0;      int result = 0;      foreach (int number in numbers)      {          if (IsPassed(number)) result += number;      }      return result;        bool IsPassed(int number)      {          return number > limit;      }  }    int[] numbers1 = { -3, -2, -1, 0, 1, 2, 3 };  int[] numbers2 = { 3, -4, 5, -6, 7 };    Console.WriteLine(Sum(numbers1));  Console.WriteLine(Sum(numbers2)); |

Здесь функция Sum вычисляет сумму чисел массива, которые соответствуют условию в локальной функции IsPassed(). Эта локальная функция проверяет, больше ли переданное число чем значение переменной limit, определенной в методе Sum. То есть локальная функция IsPassed может обращаться к данным определенным в окружающей функции Sum.

Теперь сделаем функцию IsPassed статической:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | int Sum(int[] numbers)  {      int result = 0;      int limit = 0;      foreach (int number in numbers)      {          if (IsPassed(number, limit)) result += number;      }      return result;        static bool IsPassed(int number, int lim)      {          //return number > limit; // Ошибка: метод IsPassed не имеет доступа к переменной limit          return number > lim;      }  } |

Модификатор **static** указывается перед типом локальной функции. Теперь функция IsPassed не может обращаться к переменной limit, и в этом случае нам надо либо передать это значение в виде параметра, либо определить переменную limit непосредственно в локальной функции.

## Конструкция switch

Конструкция **switch/case** оценивает некоторое выражение и сравнивает его значение с набором значений. И при совпадении значений выполняет определенный код.:

Конструкция switch имеет следующее формальное определение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | switch (выражение)  {      case значение1:          код,выполняемый если выражение имеет значение1          break;      case значение2:          код,выполняемый если выражение имеет значение1          break;      //.............      case значениеN:          код, выполняемый если выражение имеет значениеN          break;      default:          код, выполняемый если выражение не имеет ни одно из выше указанных значений          break;  } |

После ключевого слова **switch** в скобках идет сравниваемое выражение. Значение этого выражения последовательно сравнивается со значениями, помещенными после оператора **сase**. И если совпадение будет найдено, то будет выполняться определенный блок **сase**.

В конце каждого блока сase должен ставиться один из операторов перехода: **break**, **goto case**, **return** или **throw**. Как правило, используется оператор **break**. При его применении другие блоки case выполняться не будут.

Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | string name = "Tom";    switch (name)  {      case "Bob":          Console.WriteLine("Ваше имя - Bob");          break;      case "Tom":          Console.WriteLine("Ваше имя - Tom");          break;      case "Sam":          Console.WriteLine("Ваше имя - Sam");          break;  } |

В данном случае конструкция switch последовательно сравнивает значение переменной name с набором значений, которые указаны после операторов case. Поскольку здесь значение переменной name - строка "Tom", то будет выполняться блок

|  |  |
| --- | --- |
| 1  2  3 | case "Tom":      Console.WriteLine("Ваше имя - Tom");      break; |

Соответственно мы увидим на консоли

Ваше имя - Tom

Если значение переменной name не совпадает ни с каким значением после операторов case, то ни один из блоков case не выполняется. Однако если даже в этом случае нам все равно надо выполнить какие-нибудь действия, то мы можем добавить в конструкцию **switch** необязательный блок **default**. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | string name = "Alex";    switch (name)  {      case "Bob":          Console.WriteLine("Ваше имя - Bob");          break;      case "Tom":          Console.WriteLine("Ваше имя - Tom");          break;      case "Sam":          Console.WriteLine("Ваше имя - Sam");          break;      default:          Console.WriteLine("Неизвестное имя");          break;  } |

В данном случае никакое из значений после операторов case не совпадает со значением переменной name, поэтому будет выполняться блок default:

|  |  |
| --- | --- |
| 1  2  3 | default:      Console.WriteLine("Неизвестное имя");      break; |

Однако если мы хотим, чтобы, наоборот, после выполнения текущего блока case выполнялся другой блок case, то мы можем использовать вместо break оператор goto case:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | int number = 1;  switch (number)  {      case 1:          Console.WriteLine("case 1");          goto case 5; // переход к case 5      case 3:          Console.WriteLine("case 3");          break;      case 5:          Console.WriteLine("case 5");          break;      default:          Console.WriteLine("default");          break;  } |

### Возвращение значения из switch

Конструкция **switch** позволяет возвращать некоторое значение. Для возвращения значения в блоках **case** может применятся оператор **return**. Например, определим следующий метод:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | int DoOperation(int op, int a, int b)  {      switch (op)      {          case 1: return a + b;          case 2: return a - b;          case 3: return a \* b;          default: return 0;      }  } |

В метод DoOperation() передается числовой код операции и два операнда. В зависимости от кода операции над операндами выполнется определенная операция и ее результат возвращается из метода. Для примера при по умолчанию из метода возвращается 0, если код операции не равен 1, 2 или 3.

Затем мы можем вызвать этот метод:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | int DoOperation(int op, int a, int b)  {      switch (op)      {          case 1: return a + b;          case 2: return a - b;          case 3: return a \* b;          default: return 0;      }  }    int result1 = DoOperation(1, 10, 5); // 15  Console.WriteLine(result1);         // 15    int result2 = DoOperation(3, 10, 5); // 50  Console.WriteLine(result2);         // 50 |

### **Получение результата из switch**

Хотя конструкция **switch** в примере выше прекрасно работает, тем не менее мы ее можем сократить и получить результат неосредственно из конструкции **switch**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | int DoOperation(int op, int a, int b)  {      int result = op switch {          1 => a + b,          2 => a - b,          3 => a \* b,          \_ => 0      };      return result;  } |

Теперь не требуется оператор **case**, а после сравниваемого значения ставится оператор стрелка **=>**. Значение справа от стрелки выступает в качестве возвращаемоего значения. Кроме того, вместо оператора **default** используется почерк \_. В итоге результат конструкции switch будет присвиваиваться переменной result.

Естестввенно, мы можем сразу возвратить из метода результат без присвоения переменной результата конструкции switch:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | int DoOperation(int op, int a, int b)  {      return op switch      {          1 => a + b,          2 => a - b,          3 => a \* b,          \_ => 0      };  } |

Или сделать метод еще короче:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | int DoOperation(int op, int a, int b) => op switch  {      1 => a + b,      2 => a - b,      3 => a \* b,      \_ => 0  }; |

Обращаю внимание, что данное упрощение касается лишь таких конструкций **switch**, которые **возвращают** некоторые значения, как в примере выше.

Стоит отметить, что при возвращении значения из метода, метод должен в любом случае возвращать значение. Например, следующая версия метода не будет работать

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | int DoOperation(int op, int a, int b)  {      return op switch      {          1 => a + b,          2 => a - b,          3 => a \* b      };  } |

Эта версия метода возвращает значение, если код операции равен 1, 2 или 3. Но что, если будет передано значение 4 или какое-то другое? Поэтому данная версия метода даже не скомпилируется. Поэтому нам надо предусмотреть возвращение значения из метода при всех возможных вариантах. То есть, мы можем, как в примере выше, добавить в конструкцию switch блок default, в котором будет возвращаться значение при всех остальных случаях.

## Перечисления enum

Кроме примитивных типов данных в языке программирования C# есть такой тип как **enum** или **перечисление**. Перечисления представляют набор логически связанных констант.

Объявление перечисления происходит с помощью оператора **enum**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | enum название\_перечисления  {      // значения перечисления      значение1,      значение2,      .......      значениеN  } |

После оператора enum идет название перечисления. И затем в фигурных скобках через запятую перечисляются константы перечисления.

Определим простейшее перечисление:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | enum DayTime  {      Morning,      Afternoon,      Evening,      Night  } |

Здесь определено перечисление DayTime, которое имеет четыре значения: Morning, Afternoon, Evening и Night

Каждое перечисление фактически определяет новый тип данных, с помощью которых мы также, как и с помощью любого другого типа, можем определять переменные, константы, параметры методов и т.д. В качестве значения переменной, константы и параметра метода, которые представляют перечисление, должна выступать одна из констант этого перечисления, например:

|  |  |
| --- | --- |
| 1 | const DayTime dayTime = DayTime.Morning; |

Далее в программе мы можем использовать подобные переменные/константы/параметры как и любые другие:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | DayTime dayTime = DayTime.Morning;    if(dayTime == DayTime.Morning)      Console.WriteLine("Доброе утро");  else      Console.WriteLine("Привет");    enum DayTime  {      Morning,      Afternoon,      Evening,      Night  } |

### Хранение состояния

Зачастую переменная перечисления выступает в качестве хранилища состояния, в зависимости от которого производятся некоторые действия:

|  |  |
| --- | --- |
| 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 | DayTime now = DayTime.Evening;    PrintMessage(now);                   // Добрый вечер  PrintMessage(DayTime.Afternoon);    // Добрый день  PrintMessage(DayTime.Night);        // Доброй ночи    void PrintMessage(DayTime dayTime)  {      switch (dayTime)      {          case DayTime.Morning:              Console.WriteLine("Доброе утро");              break;          case DayTime.Afternoon:              Console.WriteLine("Добрый день");              break;          case DayTime.Evening:              Console.WriteLine("Добрый вечер");              break;          case DayTime.Night:              Console.WriteLine("Доброй ночи");              break;      }  }  enum DayTime  {      Morning,      Afternoon,      Evening,      Night  } |

Здесь метод PrintMessage() в качестве параметра принимает значение типа перечисления DayTime и зависимости от этого значения выводит определенное приведение.

Другой пример:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | DoOperation(10, 5, Operation.Add);          // 15  DoOperation(10, 5, Operation.Subtract);     // 5  DoOperation(10, 5, Operation.Multiply);     // 50  DoOperation(10, 5, Operation.Divide);       // 2    void DoOperation(double x, double y, Operation op)  {      double result = op switch      {          Operation.Add => x + y,          Operation.Subtract => x - y,          Operation.Multiply => x \* y,          Operation.Divide => x / y      };      Console.WriteLine(result);  }  enum Operation  {      Add,      Subtract,      Multiply,      Divide  } |

Здесь определено перечисление Operation, которое представляет арифметические операции. Каждый тип операций определен в виде одной из констант перечисления. И также определен метод DoOperation(), который в качестве параметров принимает два числа и тип операции в виде константы перечисления и в зависимости от этого типа возвращает из конструкции switch результат определенной операции.

### **Тип и значения констант перечисления**

Константы перечисления могут иметь тип. Тип указывается после названия перечисления через двоеточие:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | enum Time : byte  {      Morning,      Afternoon,      Evening,      Night  } |

Тип перечисления обязательно должен представлять целочисленный тип (byte, sbyte, short, ushort, int, uint, long, ulong). Если тип явным образом не указан, то по умолчанию используется тип **int**.

Тип влияет на значения, которые могут иметь константы. По умолчанию каждому элементу перечисления присваивается целочисленное значение, причем первый элемент будет иметь значение 0, второй - 1 и так далее. Например, возьмем выше определенное DayTime:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | DayTime now = DayTime.Morning;    Console.WriteLine((int) now);  // 0  Console.WriteLine((int) DayTime.Night);  // 3    enum DayTime  {      Morning,      Afternoon,      Evening,      Night  } |

Мы можем использовать операцию приведения, чтобы получить целочисленное значение константы перечисления:

|  |  |
| --- | --- |
| 1 | (int) DayTime.Night // 3 |

В то же время, несмотря на то, что каждая константа сопоставляется с определенным числом, мы НЕ можем присвоить ей числовое значение:

|  |  |
| --- | --- |
| 1 | DayTime now = 2;    // ! Ошибка |

Можно также явным образом указать значения элементов, либо указав значение первого элемента:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | enum DayTime  {      Morning = 3,    // каждый следующий элемент по умолчанию увеличивается на единицу      Afternoon,      // этот элемент равен 4      Evening,        // 5      Night           // 6  } |

Но можно и для всех элементов явным образом указать значения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | enum DayTime  {      Morning = 2,      Afternoon = 4,      Evening = 8,      Night = 16  } |

При этом константы перечисления могут иметь одинаковые значения, либо даже можно присваивать одной константе значение другой константы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | enum DayTime  {      Morning = 1,      Afternoon = Morning,      Evening = 2,      Night = 2  } |

**Глава 3. Классы, структуры и пространства имен**

# Классы, структуры и пространства имен

## Классы и объекты

C# является полноценным объектно-ориентированным языком. Это значит, что программу на C# можно представить в виде взаимосвязанных взаимодействующих между собой объектов.

Описанием объекта является **класс**, а объект представляет экземпляр этого класса. Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке, у которого есть имя, возраст, какие-то другие характеристики. То есть некоторый шаблон - этот шаблон можно назвать классом. Конкретное воплощение этого шаблона может отличаться, например, одни люди имеют одно имя, другие - другое имя. И реально существующий человек (фактически экземпляр данного класса) будет представлять объект этого класса.

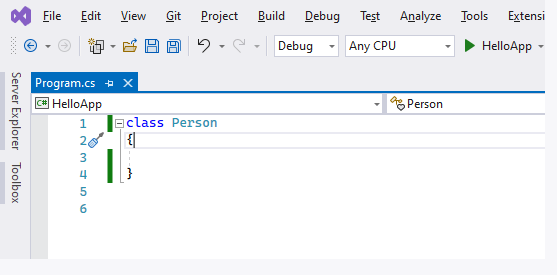
В принципе ранее уже использовались классы. Например, тип **string**, который представляет строку, фактически является классом. Или, например, класс **Console**, у которого метод WriteLine() выводит на консоль некоторую информацию. Теперь же посмотрим, как мы можем определять свои собственные классы.

По сути класс представляет новый тип, который определяется пользователем. Класс определяется с помощью ключевого слова **сlass**:

|  |  |
| --- | --- |
| 1  2  3  4 | class название\_класса  {      // содержимое класса  } |

После слова **class** идет имя класса и далее в фигурных скобках идет собственно содержимое класса. Например, определим в файле **Program.cs** класс Person, который будет представлять человека:

|  |  |
| --- | --- |
| 1  2  3  4 | class Person  {    } |

Однако такой класс не особо показателен, поэтому добавим в него некоторую функциональность.

### Поля и методы класса

Класс может хранить некоторые данные. Для хранения данных в классе применяются **поля**. По сути **поля класса** - это переменные, определенные на уровне класса.

Кроме того, класс может определять некоторое поведение или выполняемые действия. Для определения поведения в классе применяются методы.

Итак, добавим в класс Person поля и методы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | class Person  {      public string name = "Undefined";   // имя      public int age;                     // возраст        public void Print()      {          Console.WriteLine($"Имя: {name}  Возраст: {age}");      }  } |

В данном случае в классе Person определено поле name, которое хранит имя, и поле age, которое хранит возраст человека. В отличие от переменных, определенных в методах, поля класса могут иметь модификаторы, которые указываются перед полем. Так, в данном случае, чтобы все поля были доступны вне класса Person поля определены с модификатором **public**.

При определении полей мы можем присвоить им некоторые значения, как в примере выше в случае переменной name. Если поля класса не инициализированы, то они получают значения по умолчанию. Для переменных числовых типов это число 0.

Также в классе Person определен метод Print(). Методы класса имеют доступ к его поля, и в данном случае обращаемся к полям класса name и age для вывода их значения на консоль. И чтобы этот метод был виден вне класса, он также определен с модификатором **public**.

### **Создание объекта класса**

После определения класса мы можем создавать его объекты. Для создания объекта применяются **конструкторы**. По сути конструкторы представляют специальные методы, которые называются так же как и класс, и которые вызываются при создании нового объекта класса и выполняют инициализацию объекта. Общий синтаксис вызова конструктора:

|  |  |
| --- | --- |
| 1 | new конструктор\_класса(параметры\_конструктора); |

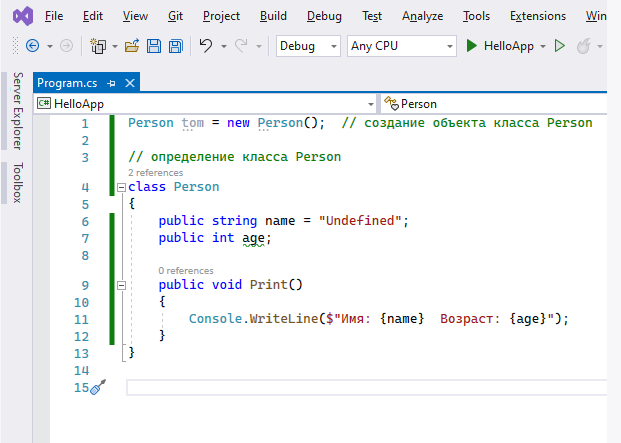
Сначала идет оператор **new**, который выделяет память для объекта, а после него идет вызов **конструктора**.

#### **Конструктор по умолчанию**

Если в классе не определено ни одного конструктора (как в случае с нашим классом Person), то для этого класса автоматически создается пустой конструктор по умолчанию, который не принимает никаких параметров.

Теперь создадим объект класса Person:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | Person tom = new Person();  // создание объекта класса Person    // определение класса Person  class Person  {      public string name = "Undefined";      public int age;        public void Print()      {          Console.WriteLine($"Имя: {name}  Возраст: {age}");      }  } |



Для создания объекта Person используется выражение new Person(). В итоге после выполнения данного выражения в памяти будет выделен участок, где будут храниться все данные объекта Person. А переменная tom получит ссылку на созданный объект, и через эту переменную мы можем использовать данный объект и обращаться к его функциональности.

### Обращение к функциональности класса

Для обращения к функциональности класса - полям, методам (а также другим элементам класса) применяется точечная нотация точки - после объекта класса ставится точка, а затем элемент класса:

|  |  |
| --- | --- |
| 1  2 | объект.поле\_класса  объект.метод\_класса(параметры\_метода) |

Например, обратимся к полям и методам объекта Person:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | Person tom = new Person();  // создание объекта класса Person    // Получаем значение полей в переменные  string personName = tom.name;  int personAge = tom.age;  Console.WriteLine($"Имя: {personName}  Возраст {personAge}");   // Имя: Undefined  Возраст: 0    // устанавливаем новые значения полей  tom.name = "Tom";  tom.age = 37;    // обращаемся к методу Print  tom.Print();    // Имя: Tom  Возраст: 37    class Person  {      public string name = "Undefined";      public int age;        public void Print()      {          Console.WriteLine($"Имя: {name}  Возраст: {age}");      }  } |

Консольный вывод данной программы:

Имя: Undefined Возраст: 0

Имя: Tom Возраст: 37

### **Константы классы**

Кроме полей класс может определять для хранения данных константы. В отличие от полей из значение устанавливается один раз непосредственно при их объявлении и впоследствии не может быть изменено. Кроме того, константы хранят некоторые данные, которые относятся не к одному объекту, а ко всему классу в целом. И для обращения к константам применяется не имя объекта, а имя класса:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | Person tom = new Person();  tom.name = "Tom";  tom.age = 37;  tom.Print();    // Person: Tom - 37    Console.WriteLine(Person.type); // Person  // Person.type = "User";    // !Ошибка: изменить константу нельзя    class Person  {      public const string type = "Person";      public string name = "Undefined";      public int age;      public void Print() => Console.WriteLine($"{type}: {name} - {age}");  } |

Здесь в классе Person определена константа type, которая хранит название класса:

|  |  |
| --- | --- |
| 1 | public const string type = "Person"; |

Название класса не зависит от объекта. Мы можем создать много объектов Person, но название класса от этого не должно измениться - оно относится ко всем объектам Person и не должно меняться. Поэтому название типа можно сохранить в виде константы.

Стоит отметить, что константе сразу при ее определении необходимо присвоить значение.

Подобно обычным полям мы можем обращаться к константам класса внутри этого класса. Например, в методе Print значение константы выводится на консоль.

Однако если мы хотим обратиться к константе вне ее класса, то для обращения необходимо использовались имя класса:

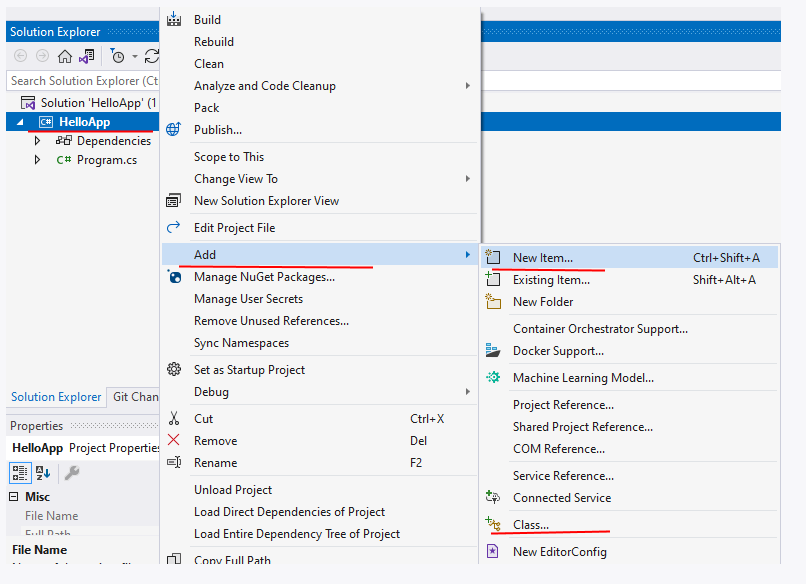
|  |  |
| --- | --- |
| 1 | Console.WriteLine(Person.type); // Person |

Таким образом, если необходимо хранить данные, которые относятся ко всему классу в целом, то можно использовать константы.

### **Добавление класса в Visual Studio**

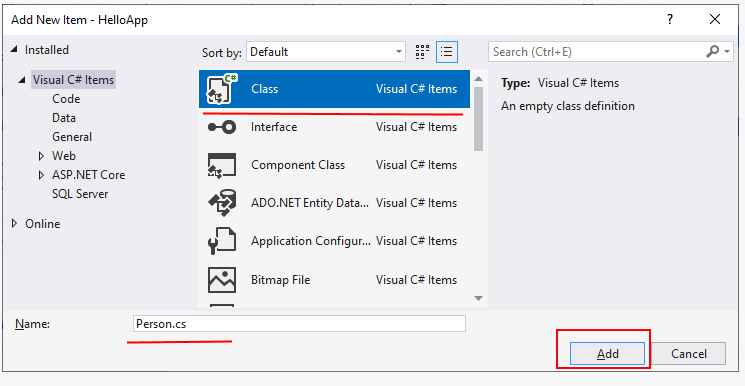
Обычно классы помещаются в отдельные файлы. Нередко для одного класса предназначен один файл. И Visual Studio предоставляет по умолчанию встроенные шаблоны для добвления класса.

Для добавления класса нажмем в Visual Studio правой кнопкой мыши на название проекта:

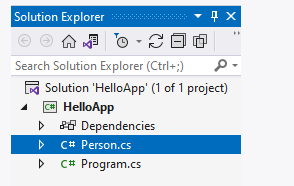


В появившемся контекстном меню выберем пункт **Add -> New Item...** (или **Add -> Class...**)

В открывшемся окне добавления нового элемента убедимся, что в центральной части с шаблонами элементов у нас выбран пункт **Class**. А внизу окна в поле **Name** введем название добавляемого класса - пусть он будет назваться **Person**:



В качестве названия класса можно вводить как Person, так и Person.cs. И после нажатия на кнопку добавления в проект будет добавлен новый класс:



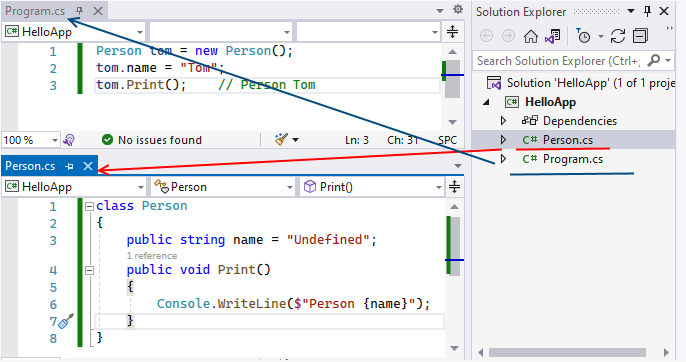
В файле **Person.cs** определим следующий код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class Person  {      public string name = "Undefined";      public void Print()      {          Console.WriteLine($"Person {name}");      }  } |

Здесь определен класс Person с одним полем name и методом Print.

В файле **Program.cs**, который представляет основной файл программы используем класс Person:

|  |  |
| --- | --- |
| 1  2  3 | Person tom = new Person();  tom.name = "Tom";  tom.Print();    // Person Tom |

**\**

Таким образом, мы можем определять классы в отдельных файлах и использовать их в программе.

## Конструкторы, инициализаторы и деконструкторы

### **Создание конструкторов**

В прошлой статье для создания объекта использовался конструктор по умолчанию. Однако мы сами можем определить свои конструкторы. Как правило, конструктор выполняет инициализацию объекта. При этом если в классе определяются свои конструкторы, то он лишается конструктора по умолчанию.

На уровне кода конструктор представляет метод, который называется по имени класса, который может иметь параметры, но для него не надо определять возвращаемый тип. Например, определим в классе Person простейший конструктор:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | Person tom = new Person();  // Создание объекта класса Person    tom.Print();    // Имя: Tom  Возраст: 37    class Person  {      public string name;      public int age;      public Person()      {          Console.WriteLine("Создание объекта Person");          name = "Tom";          age = 37;      }      public void Print()      {          Console.WriteLine($"Имя: {name}  Возраст: {age}");      }  } |

Итак, здесь определен конструктор, который выводит на консоль некоторое сообщение и инициализирует поля класса.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public Person()  {      Console.WriteLine("Создание объекта Person");      name = "Tom";      age = 37;  } |

Конструкторы могут иметь модификаторы, которые указываются перед именем конструктора. Так, в данном случае, чтобы конструктор был доступен вне класса Person, он определен с модификатором **public**.

Определив конструктор, мы можем вызвать его для создания объекта Person:

|  |  |
| --- | --- |
| 1 | Person tom = new Person();  // Создание объекта Person |

В данном случае выражение Person() как раз представляет вызов определенного в классе конструктора (это больше не автоматический конструктор по умолчанию, которого у класса теперь нет). Соответственно при его выполнении на консоли будет выводиться строка "Создание объекта Person"

Подобным образом мы можем определять и другие конструкторы в классе. Например, изменим класс Person следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | Person tom = new Person();          // вызов 1-ого конструктора без параметров  Person bob = new Person("Bob");     //вызов 2-ого конструктора с одним параметром  Person sam = new Person("Sam", 25); // вызов 3-его конструктора с двумя параметрами    tom.Print();          // Имя: Неизвестно  Возраст: 18  bob.Print();          // Имя: Bob  Возраст: 18  sam.Print();          // Имя: Sam  Возраст: 25    class Person  {      public string name;      public int age;      public Person() { name = "Неизвестно"; age = 18; }      // 1 конструктор      public Person(string n) { name = n; age = 18; }         // 2 конструктор      public Person(string n, int a) { name = n; age = a; }   // 3 конструктор        public void Print()      {          Console.WriteLine($"Имя: {name}  Возраст: {age}");      }  } |

Теперь в классе определено три конструктора, каждый из которых принимает различное количество параметров и устанавливает значения полей класса. И мы можем вызвать один из этих конструкторов для создания объекта класса.

Консольный вывод данной программы:

Имя: Неизвестно Возраст: 18

Имя: Bob Возраст: 18

Имя: Sam Возраст: 25

Стоит отметить, что начиная с версии C# 9 мы можем сократить вызов конструктора, убрав из него название типа:

|  |  |
| --- | --- |
| 1  2  3 | Person tom = new ();            // аналогично new Person();  Person bob = new ("Bob");       // аналогично new Person("Bob");  Person sam = new ("Sam", 25);   // аналогично new Person("Sam", 25); |

### Ключевое слово this

Ключевое слово **this** представляет ссылку на текущий экземпляр/объект класса. В каких ситуациях оно нам может пригодиться?

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | Person sam = new("Sam", 25);  sam.Print();          // Имя: Sam  Возраст: 25    class Person  {      public string name;      public int age;      public Person() { name = "Неизвестно"; age = 18; }      public Person(string name) { this.name = name; age = 18; }      public Person(string name, int age)      {          this.name = name;          this.age = age;      }      public void Print() => Console.WriteLine($"Имя: {name}  Возраст: {age}");  } |

В примере выше во втором и третьем конструкторе параметры называются также, как и поля класса. И чтобы разграничить параметры и поля класса, к полям класса обращение идет через ключевое слово **this**. Так, в выражении

**this.name = name;**

первая часть - this.name означает, что name - это поле текущего класса, а не название параметра name. Если бы у нас параметры и поля назывались по-разному, то использовать слово this было бы необязательно. Также через ключевое слово **this** можно обращаться к любому полю или методу.

### Цепочка вызова конструкторов

В примере выше определены три конструктора. Все три конструктора выполняют однотипные действия - устанавливают значения полей name и age. Но этих повторяющихся действий могло быть больше. И мы можем не дублировать функциональность конструкторов, а просто обращаться из одного конструктора к другому также через ключевое слово **this**, передавая нужные значения для параметров:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class Person  {      public string name;      public int age;      public Person() : this("Неизвестно")    // первый конструктор      { }      public Person(string name) : this(name, 18) // второй конструктор      { }      public Person(string name, int age)     // третий конструктор      {          this.name = name;          this.age = age;      }      public void Print() => Console.WriteLine($"Имя: {name}  Возраст: {age}");  } |

В данном случае первый конструктор вызывает второй, а второй конструктор вызывает третий. По количеству и типу параметров компилятор узнает, какой именно конструктор вызывается. Например, во втором конструкторе:

|  |  |
| --- | --- |
| 1  2 | public Person(string name) : this(name, 18)  { } |

идет обращение к третьему конструктору, которому передаются два значения. Причем в начале будет выполняться именно третий конструктор, и только потом код второго конструктора.

Стоит отметить, что в примере выше фактически все конструкторы не определяют каких-то других действий, кроме как передают третьему конструктору некоторые значения. Поэтому в реальности в данном случае проще оставить один конструктор, определив для его параметров значения по умолчанию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | Person tom = new();  Person bob = new("Bob");  Person sam = new("Sam", 25);    tom.Print();          // Имя: Неизвестно  Возраст: 18  bob.Print();          // Имя: Bob  Возраст: 18  sam.Print();          // Имя: Sam  Возраст: 25    class Person  {      public string name;      public int age;      public Person(string name = "Неизвестно", int age = 18)      {          this.name = name;          this.age = age;      }      public void Print() => Console.WriteLine($"Имя: {name}  Возраст: {age}");  } |

И если при вызове конструктора мы не передаем значение для какого-то параметра, то применяется значение по умолчанию.

### **Инициализаторы объектов**

Для инициализации объектов классов можно применять **инициализаторы**. Инициализаторы представляют передачу в фигурных скобках значений доступным полям и свойствам объекта:

|  |  |
| --- | --- |
| 1  2  3  4 | Person tom = new Person { name = "Tom", age = 31 };  // или так  // Person tom = new() { name = "Tom", age = 31 };  tom.Print();          // Имя: Tom  Возраст: 31 |

С помощью инициализатора объектов можно присваивать значения всем доступным полям и свойствам объекта в момент создания. При использовании инициализаторов следует учитывать следующие моменты:

* С помощью инициализатора мы можем установить значения только доступных из вне класса полей и свойств объекта. Например, в примере выше поля name и age имеют модификатор доступа public, поэтому они доступны из любой части программы.
* Инициализатор выполняется после конструктора, поэтому если и в конструкторе, и в инициализаторе устанавливаются значения одних и тех же полей и свойств, то значения, устанавливаемые в конструкторе, заменяются значениями из инициализатора.

Инициализаторы удобно применять, когда поле или свойство класса представляет другой класс:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | Person tom = new Person{ name = "Tom", company = { title = "Microsoft"} };  tom.Print();          // Имя: Tom  Компания: Microsoft    class Person  {      public string name;      public Company company;      public Person()      {          name = "Undefined";          company = new Company();      }      public void Print() => Console.WriteLine($"Имя: {name}  Компания: {company.title}");  }    class Company  {      public string title = "Unknown";  } |

Обратите внимание, как устанавливается поле company:

|  |  |
| --- | --- |
| 1 | company = { title = "Microsoft"} |

### **Деконструкторы**

Деконструкторы (не путать с деструкторами) позволяют выполнить декомпозицию объекта на отдельные части.

Например, пусть у нас есть следующий класс Person:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | class Person  {      string name;      int age;      public Person(string name, int age)      {          this.name = name;          this.age = age;      }        public void Deconstruct(out string personName, out int personAge)      {          personName = name;          personAge = age;      }  } |

В этом случае мы могли бы выполнить декомпозицию объекта Person так:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Person person = new Person("Tom", 33);    (string name, int age) = person;    Console.WriteLine(name);    // Tom  Console.WriteLine(age);     // 33 |

Значения переменным из деконструктора передаюся по позиции. То есть первое возвращаемое значение в виде параметра personName передается первой переменной - name, второе возващаемое значение - переменной age.

По сути деконструкторы это не более,чем синтаксический сахар. Это все равно, что если бы мы написали:

|  |  |
| --- | --- |
| 1  2  3  4 | Person person = new Person("Tom", 33);    string name; int age;  person.Deconstruct(out name, out age); |

При получении значений из декоструктора нам необходимо предоставить столько переменных, сколько деконструктор возвращает значений. Однако бывает, что не все эти значения нужны. И вместо возвращаемых значений мы можм использовать прочерк **\_**. Например, нам надо получить только возраст пользователя:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Person person = new Person("Tom", 33);    (\_, int age) = person;    Console.WriteLine(age);    // 33 |

Поскольку первое возвращаемое значение - это имя пользователя, которое не нужно, в в данном случае вместо переменной прочерк.

## Класс Program и метод Main. Программы верхнего уровня

Точкой входа в программу на языке C# является метод Main. Именно с этого метода начинается выполнение программы на C#. И программа на C# должна обязательно иметь метод Main. Однако может возникнуть вопрос, какой еще метод Main, если, например, Visual Studio 2022 по умолчанию создает проект консольного приложения со следующим кодом:

|  |  |
| --- | --- |
| 1  2 | // See <https://aka.ms/new-console-template> for more information  Console.WriteLine("Hello, World!"); |

И эта программа никаких методов Main не содержит, но при этом нормально выполняется и выводит на консоль строку "Hello, World!", как и запланировано. Это так называемая программа верхнего уровня (top-level program). А вызов Console.WriteLine("Hello, World!") представляет инструкцию вехнего уровня (top-level statement)

Однако в реальности этот код неявно помещается компилятором в метод Main, который, в свою очередь, помещается в класс Program. В действительности название класса может быть любым (как правило, это класс Program, собственно поэтому генерируемый по умолчанию файл кода называется **Program.cs**). Но метод Main является обязательной частью консольного приложения. Поэтому выше представленный код фактически эквивалентен следующей программе:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class Program  {      static void Main(string[] args)      {          // See <https://aka.ms/new-console-template> for more information          Console.WriteLine("Hello, World!");      }  } |

Определение метода **Main** обязательно начинается с модификатора static, которое указывает, что метод Main - статический. Позже мы подробнее разберем, что все это значит.

Возвращаемым типом метода **Main** обязательно является тип **void**. Кроме того, в качестве параметра он принимает массив строк - string[] args - в реальной программе это те параметры, через которые при запуске программы из консоли мы можем передать ей некоторые значения. Внутри метода располагаются действия, которые выполняет программа.

До Visual Studio 2022 все предыдущие студии создавали по умолчанию примерно такой код. Но начиная с Visual Studio 2022 нам необязательно вручную определять класс Program и в нем метод Main - компилятор генерирует их самостоятельно.

Если мы определяем какие-то переменные, константы, методы и обращаемся к ним, они помещаются в метод Main. Например, следующая программа верхнего уровня

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | string hello = "Hello METANIT.COM";    Print(hello);    void Print(string message)  {      Console.WriteLine(message);  } |

будет аналогична следующей программе:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class Program  {      static void Main(string[] args)      {          string hello = "Hello METANIT.COM";            Print(hello);            void Print(string message)          {              Console.WriteLine(message);          }      }  } |

Если определяются новые типы, например, классы, то они помещаются вне класса Program. Например, код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Person tom = new();  tom.SayHello();    class Person  {      public void SayHello() =>Console.WriteLine("Hello");  } |

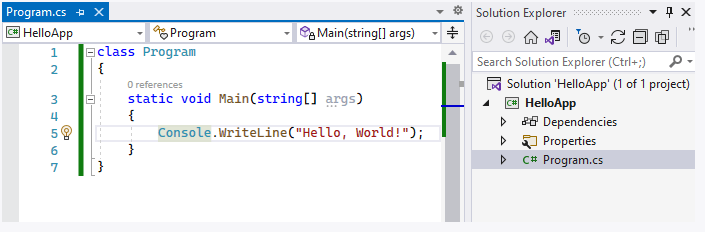
будет аналогичен следующему

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class Program  {      static void Main(string[] args)      {          Person tom = new();          tom.SayHello();      }  }  class Person  {      public void SayHello() => Console.WriteLine("Hello");  } |

Однако надо учитывать, что опредления типов (в частности классов) должны идти в конце файла после инструкций верхнего уровня. То есть:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // инструкции верхнего уровня (top-level statements)  Person tom = new();  tom.SayHello();    // определение класса идет после инструкций верхнего уровня  class Person  {      public void SayHello() => Console.WriteLine("Hello");  } |

Таким образом, мы можем продолжать писать программы верхнего уровня без явного определения метода Main. Либо мы можем явным образом определить метод Main и класс Program:

И этот код будет выполняться аналогичным образом, как если бы мы не использовали класс Program и метод Main.

## Структуры

Наряду с классами структуры представляют еще один способ создания собственных типов данных в C#. Более того многие примитивные типы, например, int, double и т.д., по сути являются структурами.

### **Определение структуры**

Для определения структуры применяется ключевое слово **struct**:

|  |  |
| --- | --- |
| 1  2  3  4 | struct имя\_структуры  {      // элементы структуры  } |

После слова **struct** идет название структуры и далее в фигурных скобках размещаются элементы структуры - поля, методы и т.д.

Например, определим структуру, которая будет называться Person и которая будет представлять человека:

|  |  |
| --- | --- |
| 1  2  3 | struct Person  {  } |

Как и классы, структуры могут хранить состояние в виде полей (переменных) и определять поведение в виде методов. Например, добавим в структуру Person пару полей и метод:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | struct Person  {      public string name;      public int age;        public void Print()      {          Console.WriteLine($"Имя: {name}  Возраст: {age}");      }  } |

В данном случае определены две переменные - name и age для хранения соответственно имени и возраста человека и метод Print для вывода информации о человеке на консоль.

И как и в случае с классами, для обращения к функциональности структуры - полям, методам и другим компонентам структуры применяется точечная нотация - после объекта структуры ставится точка, а затем указывается компонент структуры:

|  |  |
| --- | --- |
| 1  2 | объект.поле\_структуры  объект.метод\_структуры(параметры\_метода) |

### **Создание объекта структуры**

#### **Инициализация с помощью конструктора**

Для использования структуры ее необходмо инициализировать. Для инициализации создания объектов структуры, как и в случае с классами, применяется вызов конструктура с оператором **new**. Даже если в коде стуктуры не определено ни одного конструктора, тем не менее имеет как минимум один конструктор - конструктор по умолчанию, который генерируется компилятором. Этот конструктор не принимает параметров и создает объект структуры со значениями по умолчанию.

|  |  |
| --- | --- |
| 1 | new название\_структуры(); |

Например, создадим объект структуры Person с помощью конструктора по умолчанию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | Person tom = new Person();  // вызов конструктора  // или так  // Person tom = new();    tom.name = "Tom";   // изменяем значение по умолчанию в поле name    tom.Print();    // Имя: Tom  Возраст: 0    struct Person  {      public string name;      public int age;        public void Print()      {          Console.WriteLine($"Имя: {name}  Возраст: {age}");      }  } |

В данном случае создается объект tom. Для его создания вызывается конструктор по умолчанию, который устанавливает значения по умолчанию для его полей. Для числовых данных это значение 0, поэтому поле age будет иметь значение 0. Для строк это значение null, которое указывает на отсутствие значения. Но далее, если поля доступны (а в данном случае поскольку они имеют модификатор public они доступны), мы можем измениь их значения. Так, здесь полю name присваивается строка "Tom". Соответственно при выполнении метода Print() мы получим следующий консольный вывод:

Имя: Tom Возраст: 0

#### **Непосредственная иницилизация полей**

Если все поля структуры доступны (как в случае с полями структуры Person, который имеют модификатор **public**), то структуру можно инициализировать без вызова конструктора. В этом случае необходимо присвоить значения всем полям структуры перед получением значений полей и обращением к методам структуры. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | Person tom;         // не вызываем конструктор  // инициализация полей  tom.name = "Sam";  tom.age = 37;    tom.Print();    // Имя: Sam  Возраст: 37    struct Person  {      public string name;      public int age;        public void Print()      {          Console.WriteLine($"Имя: {name}  Возраст: {age}");      }  } |

### Инициализация полей по умолчанию

Стоит отметить, что начиная с версии C# 10, мы можем напрямую инициализировать поля структуры при их определении (до C# 10 это делать было нельзя):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Person tom = new Person();  tom.Print();    // Имя:Tom  Возраст: 1    struct Person  {      // инициализация полей значениями по умолчанию - доступна с C#10      public string name = "Tom";      public int age = 1;      public Person() { }      public void Print() =>Console.WriteLine($"Имя: {name}  Возраст: {age}");  } |

Однако даже в этом случае, несмотря на значения по умолчанию, необходимо явно определить и вызывать конструктор, если мы хотим использоват эти значения.

### **Конструкторы структуры**

Как и класс, структура может определять конструкторы. Однако, если в структуре определяется конструктор, то в нем обязательно надо инициализировать все поля структуры.

Например, добавим в структуру Person конструктор:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | Person tom = new();  Person bob = new("Bob");  Person sam = new("Sam", 25);    tom.Print();    // !!!! Имя:   Возраст: 0  bob.Print();    // Имя: Bob  Возраст: 1  sam.Print();    // Имя: Sam  Возраст: 25    struct Person  {      public string name;      public int age;        public Person(string name = "Tom", int age = 1)      {          this.name = name;          this.age = age;      }      public void Print() => Console.WriteLine($"Имя: {name}  Возраст: {age}");  } |

В данном случае в структуре Person определен конструктор с двумя параметрами, для которых предоставлены значения по умолчания. Однако обратите внимание на создание первого объекта структуры:

|  |  |
| --- | --- |
| 1  2 | Person tom = new(); // по прежнему используется конструктор без параметров по умолчанию  tom.Print();    // !!!! Имя:   Возраст: 0 |

Здесь по-прежнему применяется конструктор по умолчанию, тогда как при инициализации остальных двух переменных структуры применяется явно определенный конструктор.

Однако начиная с версии C# 10 мы можем определить свой конструктор без параметров:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | Person tom = new();    tom.Print();    // Имя: Tom  Возраст: 37    struct Person  {      public string name;      public int age;        public Person()      {          name = "Tom";          age = 37;      }      public void Print() => Console.WriteLine($"Имя: {name}  Возраст: {age}");  } |

Опять же при определении конструктора без параметров необходимо инициализировать все поля структуры.

В случае если нам необходимо вызывать конструкторы с различным количеством параметров, то мы можем, как и в случае с классами, вызывать их по цепочке:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | Person tom = new();  Person bob = new("Bob");  Person sam = new("Sam", 25);    tom.Print();    // Имя: Tom  Возраст: 1  bob.Print();    // Имя: Bob  Возраст: 1  sam.Print();    // Имя: Sam  Возраст: 25    struct Person  {      public string name;      public int age;        public Person() : this("Tom")      { }      public Person(string name) : this(name, 1)      { }      public Person(string name, int age)      {          this.name = name;          this.age = age;      }      public void Print() => Console.WriteLine($"Имя: {name}  Возраст: {age}");  } |

Конструкторы по прежнему должны инициализировать значения всех полей, однако поскольку при вызове любого конструктора цепочка все равно закончится на последнем конструкторе, который выполняет инициализацию, то инициализацию полей в других конструкторах можно не делать. Консольный вывод программы:

Имя: Tom Возраст: 1

Имя: Bob Возраст: 1

Имя: Sam Возраст: 25

### Инициализатор структуры

Также, как и для класса, можно использовать инициализатор для создания структуры:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | Person tom = new Person { name = "Tom", age = 22 };    tom.Print();    // Имя: Tom  Возраст: 22    struct Person  {      public string name;      public int age;      public void Print() => Console.WriteLine($"Имя: {name}  Возраст: {age}");  } |

При использовании инициализатора сначала вызывается конструктор без параметров: если мы явным образом не определили конструктор без параметров, то вызывается конструктор по умолчанию. А затем его полям присваиваются соответствующие значения.

### **Копирование структуры с помощью with**

Если нам необходимо скопировать в один объект структуры значения из другого с небольшими изменениями, то мы можем использовать оператор **with**:

|  |  |
| --- | --- |
| 1  2  3 | Person tom = new Person { name = "Tom", age = 22 };  Person bob = tom with { name = "Bob" };  bob.Print();    // Имя: Bob  Возраст: 22 |

В данном случае объект bob получает все значения объекта tom, а затем после оператора **with** в фигурных скобках указывается поля со значениями, которые мы хотим изменить.

## Типы значений и ссылочные типы

Ранее мы рассматривали следующие элементарные типы данных: int, byte, double, string, object и др. Также есть сложные типы: структуры, перечисления, классы. Все эти типы данных можно разделить на типы значений, еще называемые значимыми типами, (value types) и ссылочные типы (reference types). Важно понимать между ними различия.

Типы значений:

* Целочисленные типы (byte, sbyte, short, ushort, int, uint, long, ulong)
* Типы с плавающей запятой (float, double)
* Тип decimal
* Тип bool
* Тип char
* Перечисления enum
* Структуры (struct)

Ссылочные типы:

* Тип object
* Тип string
* Классы (class)
* Интерфейсы (interface)
* Делегаты (delegate)

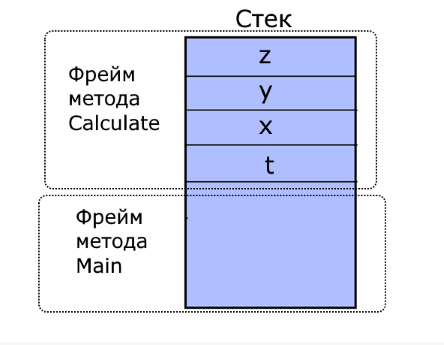
В чем же между ними различия? Для этого надо понять организацию памяти в .NET. Здесь память делится на два типа: стек и куча (heap). Параметры и переменные метода, которые представляют типы значений, размещают свое значение в стеке. Стек представляет собой структуру данных, которая растет снизу вверх: каждый новый добавляемый элемент помещается поверх предыдущего. Время жизни переменных таких типов ограничено их контекстом. Физически стек - это некоторая область памяти в адресном пространстве.

Когда программа только запускается на выполнение, в конце блока памяти, зарезервированного для стека устанавливается указатель стека. При помещении данных в стек указатель переустанавливается таким образом, что снова указывает на новое свободное место. При вызове каждого отдельного метода в стеке будет выделяться область памяти или фрейм стека, где будут храниться значения его параметров и переменных.

Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class Program  {      static void Main(string[] args)      {          Calculate(5);      }        static void Calculate(int t)      {          int x = 6;          int y = 7;          int z = y + t;      }  } |

Пи запуске такой программы в стеке будут определяться два фрейма - для метода Main (так как он вызывается при запуске программы) и для метода Calculate:



При вызове этого метода Calculate в его фрейм в стеке будут помещаться значения t, x, y и z. Они определяются в контексте данного метода. Когда метод отработает, область памяти, которая выделялась под стек, впоследствии может быть использована другими методами.

Причем если параметр или переменная метода представляет тип значений, то в стеке будет храниться непосредсвенное значение этого параметра или переменной. Например, в данном случае переменные и параметр метода Calculate представляют значимый тип - тип int, поэтому в стеке будут храниться их числовые значения.

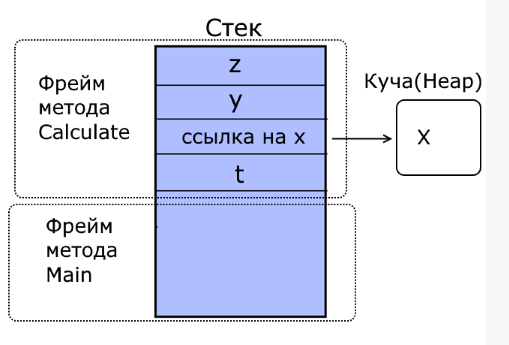
Ссылочные типы хранятся в куче или хипе, которую можно представить как неупорядоченный набор разнородных объектов. Физически это остальная часть памяти, которая доступна процессу.

При создании объекта ссылочного типа в стеке помещается ссылка на адрес в куче (хипе). Когда объект ссылочного типа перестает использоваться, в дело вступает автоматический сборщик мусора: он видит, что на объект в хипе нету больше ссылок, условно удаляет этот объект и очищает память - фактически помечает, что данный сегмент памяти может быть использован для хранения других данных.

Так, в частности, если мы изменим метод Calculate следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | static void Calculate(int t)  {      object x = 6;      int y = 7;      int z = y + t;  } |

То теперь значение переменной x будет храниться в куче, так как она представляет ссылочный тип object, а в стеке будет храниться ссылка на объект в куче.

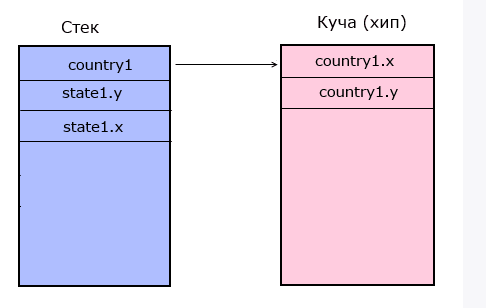


### **Составные типы**

Теперь рассмотим ситуацию, когда тип значений и ссылочный тип представляют составные типы - структуру и класс:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | State state1 = new State(); // State - структура, ее данные размещены в стеке  Country country1 = new Country(); // Country - класс, в стек помещается ссылка на адрес в хипе                                    // а в хипе располагаются все данные объекта country1  struct State  {      public int x;      public int y;  }  class Country  {      public int x;      public int y;  } |

Здесь в методе Main в стеке выделяется память для объекта state1. Далее в стеке создается ссылка для объекта country1 (Country country1), а с помощью вызова конструктора с ключевым словом new выделяется место в хипе (new Country()). Ссылка в стеке для объекта country1 будет представлять адрес на место в хипе, по которому размещен данный объект..

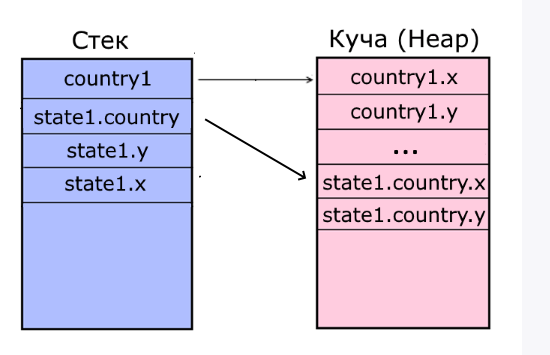


Таким образом, в стеке окажутся все поля структуры state1 и ссылка на объект country1 в хипе.

Но, допустим, в структуре State также определена переменная ссылочного типа Country. Где она будет хранить свое значение, если она определена в типе значений?

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | State state1 = new State();  Country country1 = new Country();    struct State  {      public int x;      public int y;      public Country country;      public State()      {          x = 0;          y = 0;          country = new Country();      }  }  class Country  {      public int x;      public int y;  } |

Значение переменной state1.country также будет храниться в куче, так как эта переменная представляет ссылочный тип:



### **Копирование значений**

Тип данных надо учитывать при копировании значений. При присвоении данных объекту значимого типа он получает копию данных. При присвоении данных объекту ссылочного типа он получает не копию объекта, а ссылку на этот объект в хипе. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | State state1 = new State(); // Структура State  State state2 = new State();  state2.x = 1;  state2.y = 2;  state1 = state2;  state2.x = 5; // state1.x=1 по-прежнему  Console.WriteLine(state1.x); // 1  Console.WriteLine(state2.x); // 5    Country country1 = new Country(); // Класс Country  Country country2 = new Country();  country2.x = 1;  country2.y = 4;  country1 = country2;  country2.x = 7; // теперь и country1.x = 7, так как обе ссылки и country1 и country2                  // указывают на один объект в хипе  Console.WriteLine(country1.x); // 7  Console.WriteLine(country2.x); // 7 |

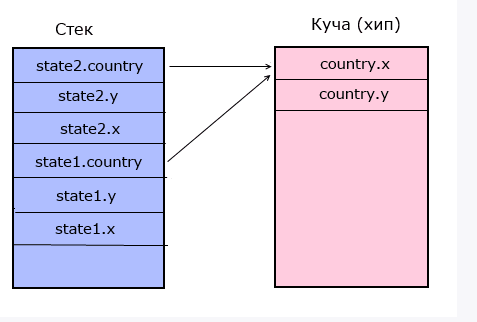
Так как state1 - структура, то при присвоении state1 = state2 она получает копию структуры state2. А объект класса country1 при присвоении country1 = country2; получает ссылку на тот же объект, на который указывает country2. Поэтому с изменением country2, так же будет меняться и country1.

### Ссылочные типы внутри типов значений

Теперь рассмотрим более изощренный пример, когда внутри структуры у нас может быть переменная ссылочного типа, например, какого-нибудь класса:

|  |  |
| --- | --- |
| 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 | State state1 = new State();  State state2 = new State();    state2.country = new Country();  state2.country.x = 5;  state1 = state2;  state2.country.x = 8; // теперь и state1.country.x=8, так как state1.country и state2.country                        // указывают на один объект в хипе  Console.WriteLine(state1.country.x); // 8  Console.WriteLine(state2.country.x); // 8    struct State  {      public int x;      public int y;      public Country country;      public State()      {          x = 0;          y = 0;          country = new Country(); // выделение памяти для объекта Country      }  }  class Country  {      public int x;      public int y;  } |

Переменные ссылочных типов в структурах также сохраняют в стеке ссылку на объект в хипе. И при присвоении двух структур state1 = state2; структура state1 также получит ссылку на объект country в хипе. Поэтому изменение state2.country повлечет за собой также изменение state1.country.



### **Объекты классов как параметры методов**

Организацию объектов в памяти следует учитывать при передаче параметров по значению и по ссылке. Если параметры методов представляют объекты классов, то использование параметров имеет некоторые особенности. Например, создадим метод, который в качестве параметра принимает объект Person:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | Person p = new Person { name = "Tom", age = 23 };  ChangePerson(p);    Console.WriteLine(p.name); // Alice  Console.WriteLine(p.age); // 23    void ChangePerson(Person person)  {      // сработает      person.name = "Alice";      // сработает только в рамках данного метода      person = new Person { name = "Bill", age = 45 };      Console.WriteLine(person.name); // Bill  }    class Person  {      public string name = "";      public int age;  } |

При передаче объекта класса по значению в метод передается копия ссылки на объект. Эта копия указывает на тот же объект, что и исходная ссылка, потому мы можем изменить отдельные поля и свойства объекта, но не можем изменить сам объект. Поэтому в примере выше сработает только строка person.name = "Alice".

А другая строка person = new Person { name = "Bill", age = 45 } создаст новый объект в памяти, и person теперь будет указывать на новый объект в памяти. Даже если после этого мы его изменим, то это никак не повлияет на ссылку p в методе Main, поскольку ссылка p все еще указывает на старый объект в памяти.

Но при передаче параметра по ссылке (с помощью ключевого слова **ref**) в метод в качестве аргумента передается сама ссылка на объект в памяти. Поэтому можно изменить как поля и свойства объекта, так и сам объект:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | Person p = new Person { name = "Tom", age = 23 };  ChangePerson(ref p);    Console.WriteLine(p.name); // Bill  Console.WriteLine(p.age); // 45    void ChangePerson(ref Person person)  {      // сработает      person.name = "Alice";      // сработает      person = new Person { name = "Bill", age = 45 };  }    class Person  {      public string name = "";      public int age;  } |

Операция new создаст новый объект в памяти, и теперь ссылка person (она же ссылка p из метода Main) будет указывать уже на новый объект в памяти.

## Область видимости (контекст) переменных и констант

Каждая переменная доступна в рамках определенного контекста или области видимость. Вне этого контекста переменная уже не существует.

Существуют различные контексты:

* Контекст класса. Переменные, определенные на уровне класса, доступны в любом методе этого класса. Их еще называют глобальными переменными или полями
* Контекст метода. Переменные, определенные на уровне метода, являются локальными и доступны только в рамках данного метода. В других методах они недоступны
* Контекст блока кода. Переменные, определенные на уровне блока кода, также являются локальными и доступны только в рамках данного блока. Вне своего блока кода они не доступны.

Например, пусть класс Program определен следующим образом:

|  |  |
| --- | --- |
| 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 | Person tom = new();  tom.PrintName();  tom.PrintSurname();    class Person                            // начало контекста класса  {      string type = "Person";             // переменная уровня класса      public void PrintName()             // начало контекста метода PrintName      {          string name = "Tom";            // переменная уровня метода            {                               // начало контекста блока кода              string shortName = "Tomas"; // переменная уровня блока кода              Console.WriteLine(type);    // в блоке доступна переменная класса              Console.WriteLine(name);    // в блоке доступна переменная окружающего метода              Console.WriteLine(shortName);// в блоке доступна переменная этого же блока          }                               // конец контекста блока кода, переменная shortName уничтожается            Console.WriteLine(type);        // в методе доступна переменная класса          Console.WriteLine(name);        // в методе доступна переменная этого же метода          //Console.WriteLine(shortName); //так нельзя, переменная c определена в блоке кода          //Console.WriteLine(surname);     //так нельзя, переменная surname определена в другом методе        }       // конец контекста метода PrintName, переменная name уничтожается        public void PrintSurname()      // начало контекста метода PrintSurname      {          string surname = "Smith";   // переменная уровня метода            Console.WriteLine(type);        // в методе доступна переменная класса          Console.WriteLine(surname);     // в методе доступна переменная этого же метода      }       // конец конекста метода PrintSurname, переменная surname уничтожается    }   // конец контекста класса, переменная type уничтожается |

Здесь определенно четыре переменных: type, name, shortName и surname. Каждая из них существует в своем контексте. Переменная type существует в контексте всего класса Person и доступна в любом месте и блоке кода в методах PrintName и PrintSurname.

Переменная name существует только в рамках метода PrintName. Также как и переменная surname существует в рамках метода PrintSurname. В методе PrintName мы не можем обратиться к переменной surname, так как она в другом контексте.

Переменная shortName существует только в блоке кода, границами которого являются открывающая и закрывающая фигурные скобки. Вне его границ переменная shortName не существует и к ней нельзя обратиться.

Нередко границы различных контекстов можно ассоциировать с открывающимися и закрывающимися фигурными скобками, как в данном случае, которые задают пределы блока кода, метода, класса.

При работе с переменными надо учитывать, что локальные переменные, определенные в методе или в блоке кода, скрывают переменные уровня класса, если их имена совпадают:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | class Person  {      string name = "Tom";             // переменная уровня класса      public void PrintName()      {          string name = "Tomas";      // переменная уровня метода скрывает переменную уровня класса            Console.WriteLine(name);    // Tomas      }  } |

При объявлении переменных также надо учитывать, что в одном контексте нельзя определить несколько переменных с одним и тем же именем.

## Пространства имен

Обычно определяемые классы и другие типы в .NET не существуют сами по себе, а заключаются в специальные контейнеры - **пространства имен**. Пространства имен позволяют организовать код программы в логические блоки, поволяют объединить и отделить от остального кода некоторую функциональность, которая связана некоторой общей идеей или которая выполняет определенную задачу.

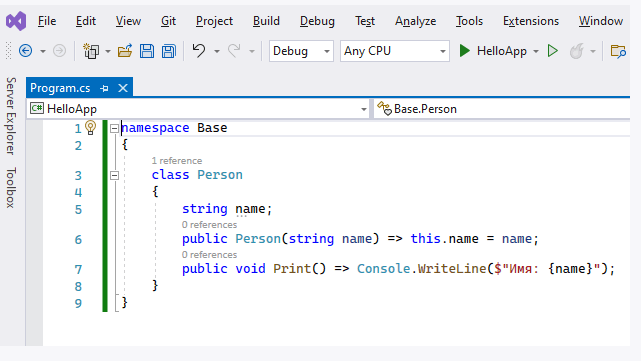
Для определения пространства имен применяется ключевое слово namespace, после которого идет название название пространства имен:

|  |  |
| --- | --- |
| 1  2  3  4 | namespace имя\_пространства\_имен  {      // содержимое пространства имен  } |

Например, определим в файле **Program.cs** пространство имен, которое будет называться **Base**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | namespace Base  {      class Person      {          string name;          public Person(string name) => this.name = name;          public void Print() => Console.WriteLine($"Имя: {name}");      }  } |

Здесь пространство имен содержит класс Person, которой имеет одну переменную - name, конструктор и метод Print.



Теперь попробуем использовать класс Person:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | Person tom = new Person("Tom"); // Ошибка - Visual Studio не видит класс Person  tom.Print();    namespace Base  {      class Person      {          string name;          public Person(string name) => this.name = name;          public void Print() => Console.WriteLine($"Имя: {name}");      }  } |

Здесь на первой строке мы столкнемся с ошибкой, так как Visual Studio не может найти класс Person. Чтобы все-таки обратиться к классу Person, необходимо использовать полное имя этого класса с учетом пространства имен:

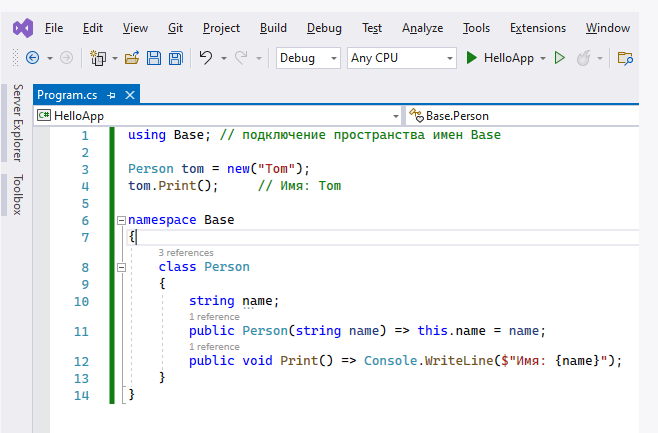
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | Base.Person tom = new("Tom");  tom.Print();          // Имя: Tom    namespace Base  {      class Person      {          string name;          public Person(string name) => this.name = name;          public void Print() => Console.WriteLine($"Имя: {name}");      }  } |

Поскольку класс Person располагается в пространстве имен Base, то его полное имя - **Base.Person**. Соответственно, используя это имя, мы можем обращаться к данному классу вне его пространства имен.

### Подключение пространства имен

Однако полное имя класса с учетом пространства имен добавляет в код избыточность - особенно, если пространство имен содержит множество классов, которые мы хотим использовать. И чтобы не писать полное имя класса, мы можем просто подключить пространство имен с помощью директивы **using**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using Base; // подключение пространства имен Base    Person tom = new("Tom");  tom.Print();     // Имя: Tom    namespace Base  {      class Person      {          string name;          public Person(string name) => this.name = name;          public void Print() => Console.WriteLine($"Имя: {name}");      }  } |



### **Вложенные пространства имен**

Одни пространства имен могут содержать другие. Например:

|  |  |
| --- | --- |
| 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 | namespace Base  {      namespace PersonTypes      {          class Person          {              string name;              OrganisationTypes.Company company;              public Person(string name, OrganisationTypes.Company company)              {                  this.name = name;                  this.company = company;              }              public void Print()              {                  Console.WriteLine($"Имя: {name} ");                  company.Print();              }          }      }      namespace OrganisationTypes      {          class Company          {              string title;              public Company(string title) => this.title = title;              public void Print() => Console.WriteLine($"Название компании: {title}");          }      }  } |

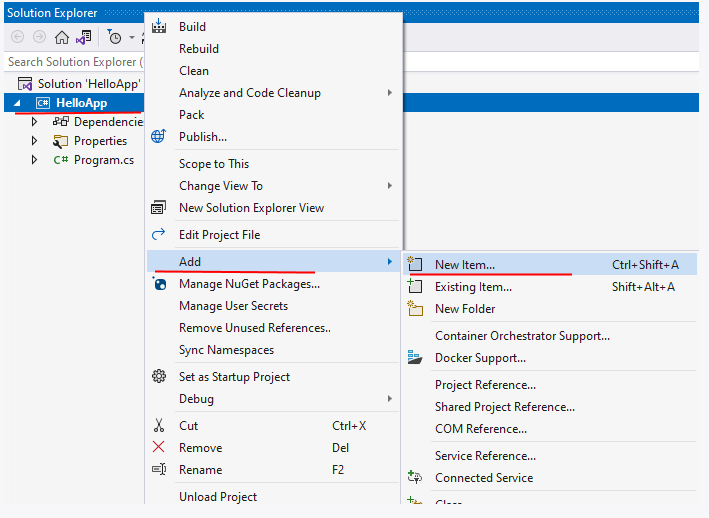
В данном случае классы Person и Company находятся в разных вложенных пространствах имен, поэтому чтобы в классе Person использовать класс Company, надо прописывать имя класса с учетом его пространства имен: OrganisationTypes.Company (так как оба класса в итоге находятся в общем пространстве - Base, то его имя можно не указывать в названии класса) или подключать пространство имен класса Company с помощью директивы using.

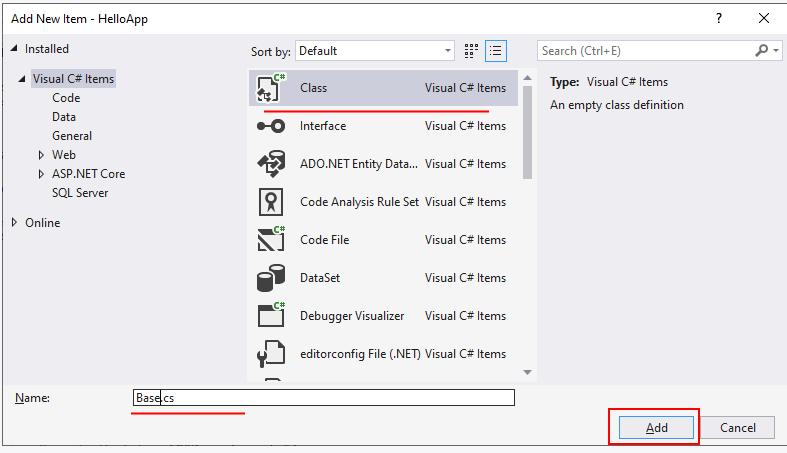
Для обращения к этим классам вне пространства Base необходимо использовать всю цепочку пространств имен:

|  |  |
| --- | --- |
| 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 | using Base.PersonTypes; // подключение пространства имен Base.PersonTypes для класса Person    Base.OrganisationTypes.Company microsoft = new("Microsoft");  Person tom = new("Tom", microsoft);  tom.Print();     // Имя: Tom   Название компании: Microsoft    namespace Base  {      namespace PersonTypes      {          class Person          {              string name;              OrganisationTypes.Company company;              public Person(string name, OrganisationTypes.Company company)              {                  this.name = name;                  this.company = company;              }              public void Print()              {                  Console.WriteLine($"Имя: {name} ");                  company.Print();              }          }      }      namespace OrganisationTypes      {          class Company          {              string title;              public Company(string title) => this.title = title;              public void Print() => Console.WriteLine($"Название компании: {title}");          }      }  } |

### **Пространства имен уровня файла**

Начиная с .NET 6 и C# 10 можно определять пространства имен на уровне файла. Например, добавим в проект новый файл с кодом c#. Для этого в Visual Studio нажмем на название проекта правой кнопкой мыши и в появившемся меню выберем пункт **Add -> New Item...**

В окне добавления нового элемента выберем пункт **Class** и поле **Name** в качестве имени файла укажем **Base.cs**



После этого в проект будет добавлен файл **Base.cs**. Удалим из него все содержимое по умолчанию и определим в нем следующий код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | namespace Base;    class Person  {      string name;      public Person(string name) => this.name = name;      public void Print() => Console.WriteLine($"Имя: {name} ");  } |

Директива namespace Base в начале файла указывает, что содержимое файла будет представлять пространство имен Base.

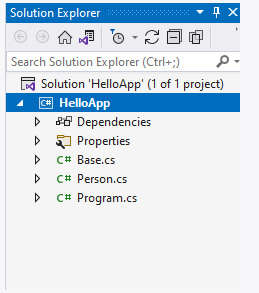
Теперь подключим это пространство имен в файле **Program.cs**:

|  |  |
| --- | --- |
| 1  2  3  4 | using Base; // подключение пространства имен Base    Person tom = new("Tom");  tom.Print(); |

## Глобальные пространства имен

Если мы хотим использовать какое-нибудь пространство имен в файлах кода в проекте, то по умолчанию нам надо его подключать во все файлы, где мы планируем его использовать.

Например, пусть у нас в проекте есть три файла с кодом:



В файле **Base.cs** определяется класс Company в пространстве **Base**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | namespace Base;  class Company  {      string title;      public Company(string title) => this.title = title;      public void Print() => Console.WriteLine($"Компания: {title}");  } |

В файле **Person.cs** определен класс Person, который использует класс Company:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | using Base;    class Person  {      string name;      Company company;      public Person(string name, Company company)      {          this.name = name;          this.company = company;      }      public void Print()      {          Console.WriteLine($"Имя: {name}");          company.Print();      }  } |

Чтобы использовать класс Company мы подключаем его пространство имен: using Base;

В классе **Program.cs** используются классы Person и Company:

|  |  |
| --- | --- |
| 1  2  3  4  5 | using Base;    Company microsoft = new("Microsoft");  Person tom = new("Tom", microsoft);  tom.Print(); |

Опять же чтобы использовать класс Company, подключается его пространство имен: using Base;

И таких файлов, где надо подключать пространство Base (или какое-то другое) может быть множество. Это не очень удобно. И в .NET 6 для этого предложена концепция **глобальных пространств имен**, который подключаются один раз но сразу во все файлы кода в проекте. Для этого нам достаточно в одном файле прописать директиву:

|  |  |
| --- | --- |
| 1 | global using подключаемое\_пространство\_имен; |

Например, изменим файл **Program.cs** следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5 | global using Base;    Company microsoft = new("Microsoft");  Person tom = new("Tom", microsoft);  tom.Print(); |

Теперь пространство Base подключается во все файлы кода в проекте. И из файла **Person.cs** мы можем убрать строку

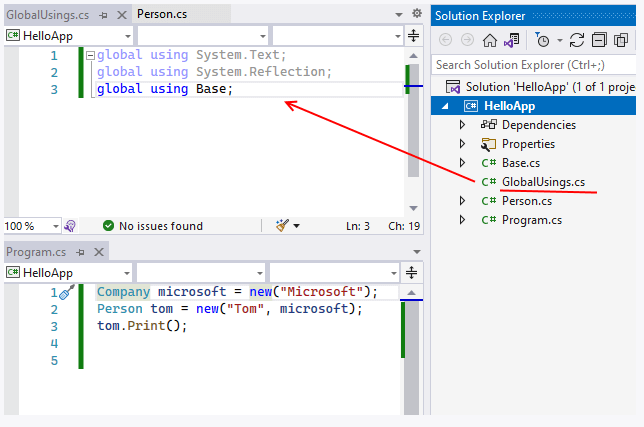
|  |  |
| --- | --- |
| 1 | using Base; |

### Определение глобальных пространств в отдельном файле

Однако вышеуказанный подход опять же может быть не очень удобным, поскольку проще определить глобальные пространство имен, которые подключаются во весь проект где-то в одном месте. И для этого в Visual Studio 2022 мы можем добавить в проект новый файл с кодом C# и в нем определить набор подключаемых пространств имен. Например, добавим в проект файл, который назовем **GlobalUsings.cs** и в котором определим следующее содержимое:

|  |  |
| --- | --- |
| 1  2  3 | global using System.Text;  global using System.Reflection;  global using Base; |

И этот набор пространств имен будет автоматически подключаться во все файлы кода в проекте.

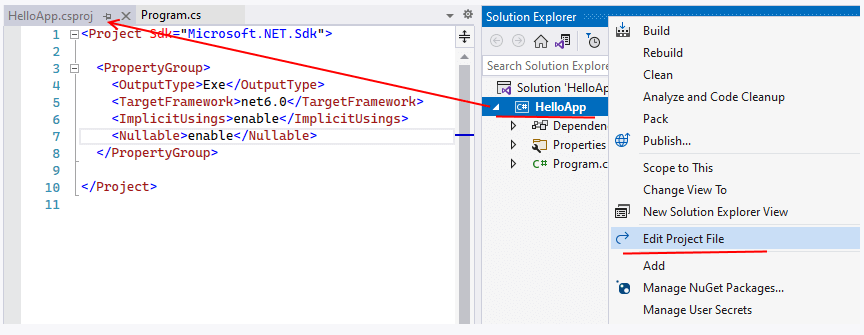


## Подключение пространств имен по умолчанию

Все классы существуют в некотором пространстве имен, и чтобы эти классы использовать, необходимо подключить их пространства имен, либо использовать полное название класса с указанием его пространства имен. Однако начиная с Visual Studio 2022 и .NET 6 и C# 10 мы можем просто в файле программы написать:

|  |  |
| --- | --- |
| 1 | Console.WriteLine("Hello"); |

При этом не подключая явно пространство имен System, где располагается класс Console, тем не менее этот класс будет доступен, и мы его сможем использовать в любом файле кода C# в проекте. Дело в том, что начиная с .NET 6 в проекте по умолчанию подключается ряд наиболее часто используемых пространств имен, поэтому нам их не надо явно подключать. Эта настройка действует на уровне всего проекта. Так, откроем файл проекта. Для этого либо двойным кликом левой кнопкой мыши нажмем на проект, либо нажмем на проект правой кнопкой мыши и в появившемся меню выберем пункт **Edit Project File**



После этого Visual Studio откроет нам файл проекта, который будет выглядеть примерно следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | <Project Sdk="Microsoft.NET.Sdk">      <PropertyGroup>      <OutputType>Exe</OutputType>      <TargetFramework>net6.0</TargetFramework>      <ImplicitUsings>enable</ImplicitUsings>      <Nullable>enable</Nullable>    </PropertyGroup>    </Project> |

Здесь строка

|  |  |
| --- | --- |
| 1 | <ImplicitUsings>enable</ImplicitUsings> |

точнее элемент <ImplicitUsings> задает подключение некоторого набора пространств имен по умолчанию. Значение **enable** указывает, что эта настройка будет применяться.

Если мы запустим проект на выполнение, то после компиляции проект в проекте в папке **obj/Debug/net6.0** можно будет увидеть файл **[Имя\_проекта].GlobalUsings.g.cs** (например, у меня проект называется HelloApp, соответственно файл будет называться **HelloApp.GlobalUsings.g.cs**). Если мы откроем этот файл, то увидим там следующее содержимое:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | global using global::System;  global using global::System.Collections.Generic;  global using global::System.IO;  global using global::System.Linq;  global using global::System.Net.Http;  global using global::System.Threading;  global using global::System.Threading.Tasks; |

Здесь перечислены все те пространства имен, которые подключаются в проект по умолчанию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | System  System.Collections.Generic  System.IO  System.Linq  System.Net.Http  System.Threading  System.Threading.Tasks |

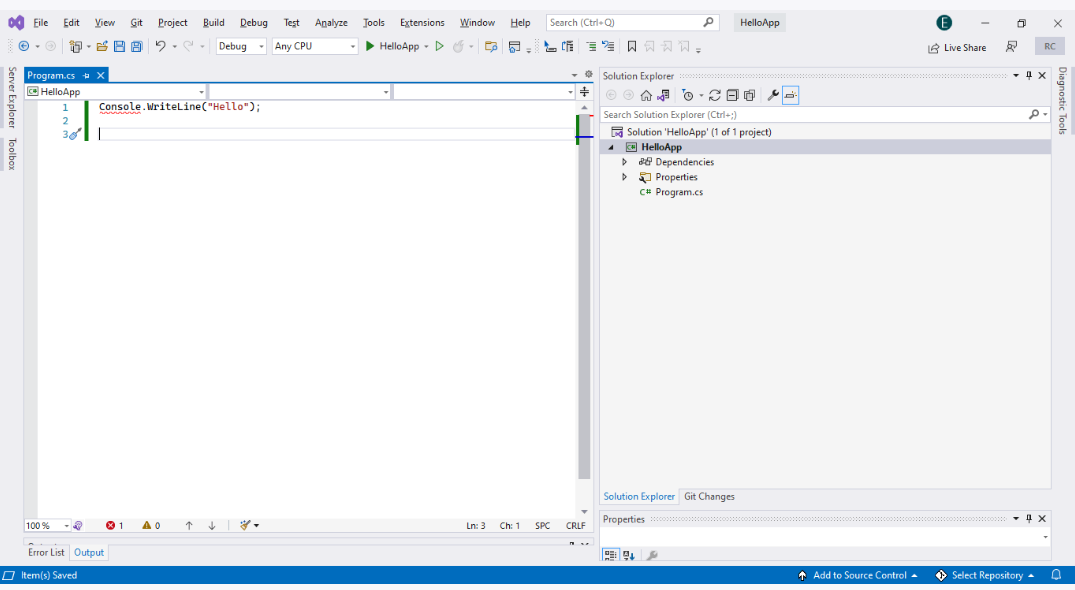
Стоит отметить, что это набор пространств имен, которые подключаются в проект консольного приложения по умолчанию, у других типов проектов набор подключаемых пространств может отличаться.

### Отключение пространств по умолчанию

В проектах, создаваемых в Visual Studio 2022 для .NET 6 и C# 10 эта возможность включена по умолчанию. Тем не менее при необходимости мы ее можем отключить. Для этого изменим эту настройку следующим образом:

|  |  |
| --- | --- |
| 1 | <ImplicitUsings>disable</ImplicitUsings> |

После этого Visual Studio 2022 подчеркнет нам класс Console, так как он нигде не определен в проекте, а его пространство имен не подключено:



В этом случае нам потребуется либо подключить пространство имен System, где располагается класс Console, либо использовать его полное имя:

|  |  |
| --- | --- |
| 1  2  3  4  5 | using System;   // или подключить пространство System  Console.WriteLine("Hello world");    // или использовать полное имя класса  System.Console.WriteLine("Hello work"); |

### Подключение и отключение пространств по отдельности

Вместо подключения по умолчанию некоторого набора пространств имен по умолчанию также можно подключать и отключать по отдельности пространств имен. Так, изменим файл проекта следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | <Project Sdk="Microsoft.NET.Sdk">      <PropertyGroup>      <OutputType>Exe</OutputType>      <TargetFramework>net6.0</TargetFramework>      <ImplicitUsings>disable</ImplicitUsings>      <Nullable>enable</Nullable>    </PropertyGroup>      <ItemGroup>      <Using Include="System" />      <Using Include="System.Threading.Tasks" />    </ItemGroup>    </Project> |

Здесь атрибут **Include** элемента **Using** подключает пространства имен глобально по всему проекту. Например:

|  |  |
| --- | --- |
| 1 | <Using Include="System" /> |

Подключает пространство System. Соответственно после этого нам не надо его подключать в файлы кода в проекте.

Обратная ситуация - отключение некоторых ненужных пространств имен из тех, которые подключаются по умолчанию. Для этого применяется атрибут **Remove** элемента **Using**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | <Project Sdk="Microsoft.NET.Sdk">      <PropertyGroup>      <OutputType>Exe</OutputType>      <TargetFramework>net6.0</TargetFramework>      <ImplicitUsings>enable</ImplicitUsings>      <Nullable>enable</Nullable>    </PropertyGroup>      <ItemGroup>      <Using Remove="System" />      <Using Remove="System.Threading.Tasks"/>    </ItemGroup>    </Project> |

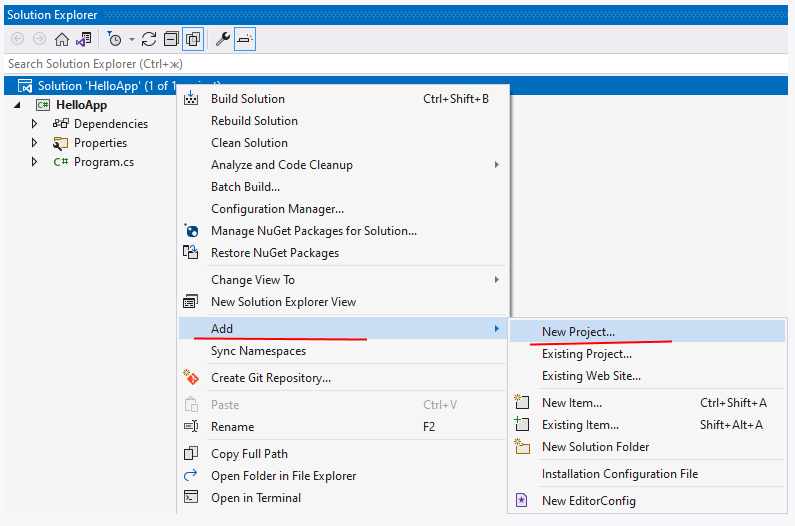
В данном случае отключаются пространства "System" и "System.Threading.Tasks"

## Создание библиотеки классов

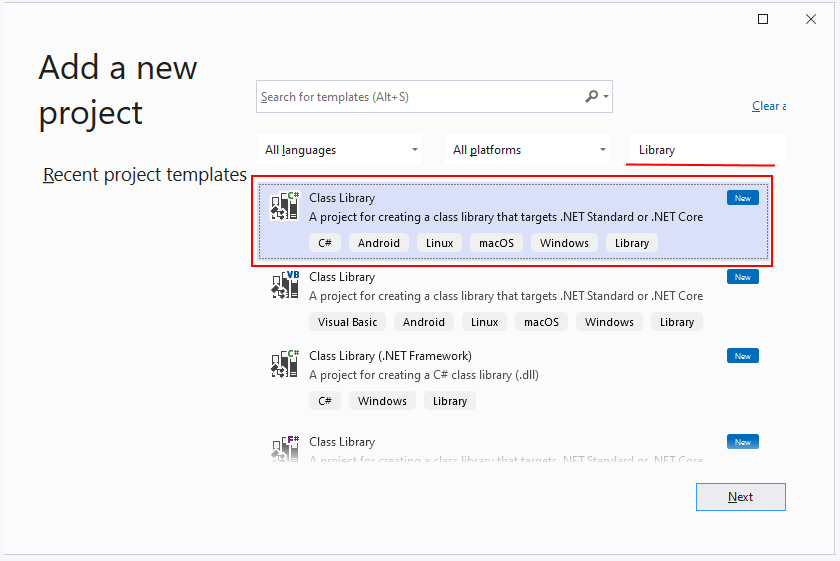
Нередко различные классы и структуры оформляются в виде отдельных библиотек, которые компилируются в файлы dll и затем могут подключаться в другие проекты. Благодаря этому мы можем определить один и тот же функционал в виде библиотеки классов и подключать в различные проекты или передавать на использование другим разработчикам.

**Создадим и подключим библиотеку классов.**

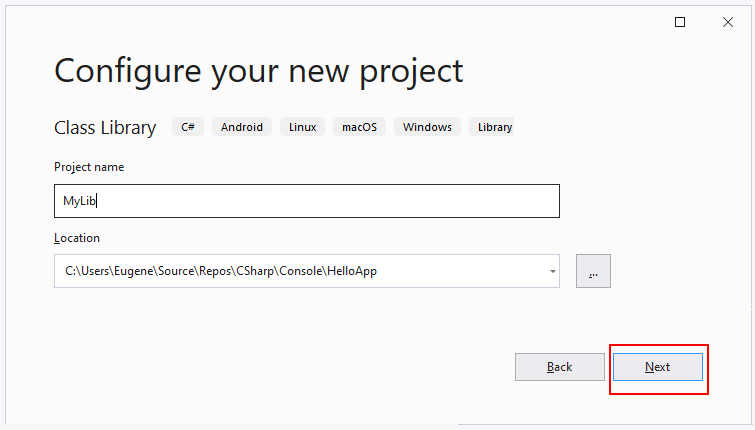
Возьмем имеющийся проект консольного приложения C#, например, созданный в прошлых темах. В структуре проекта нажмем правой кнопкой на название решения и далее в появившемся контекстном меню выберем **Add -> New Project...** (Добавить новый проект):



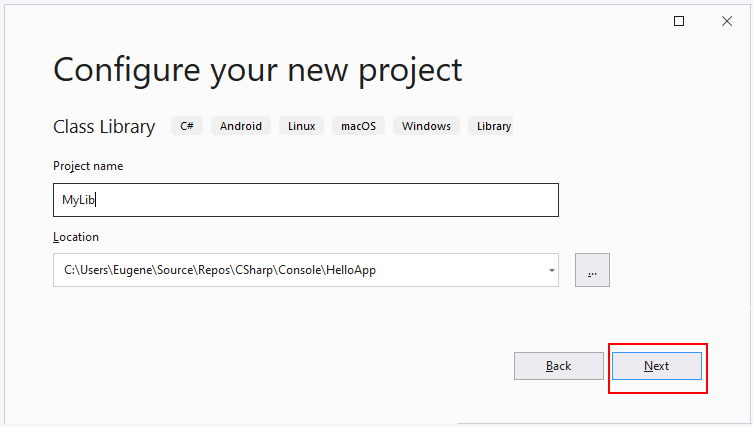
Далее в списке шаблонов проекта найдем пункт **Class Library**:



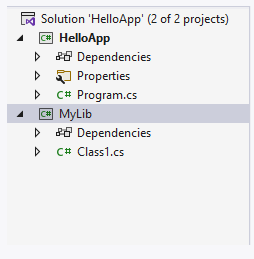
Затем дадим новому проекту какое-нибудь название, например, MyLib:



Далее будет предложено выбрать версию фреймворка, под которую создается библиоека:



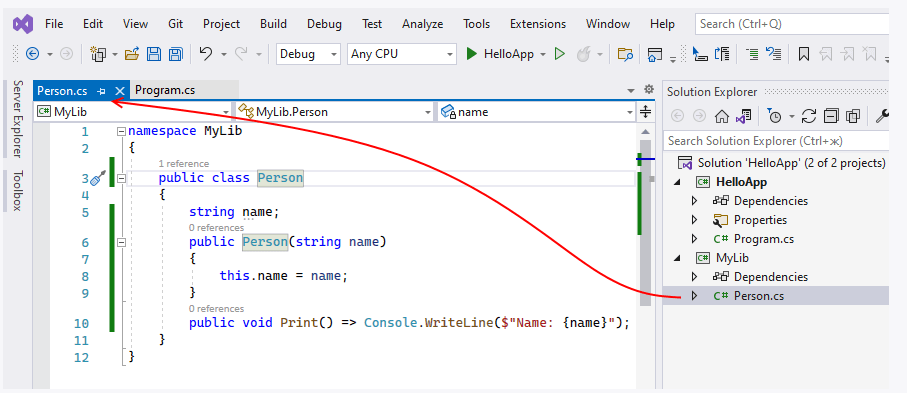
Оставим здесь **.NET 6.0 (Long-term support)** и нажмем на кнопку Create. После этого в решение будет добавлен новый проект, в моем случае с названием MyLib:



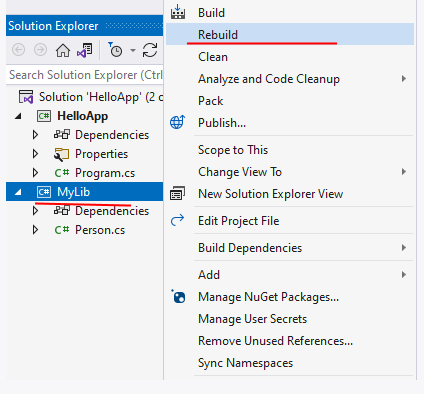
По умолчанию новый проект имеет один пустой класс Class1 в файле Class1.cs. Мы можем этот файл удалить или переименовать, как нам больше нравится.

Например, переименуем файл Class1.cs в Person.cs, а класс Class1 в Person. Определим в классе Person простейший код:

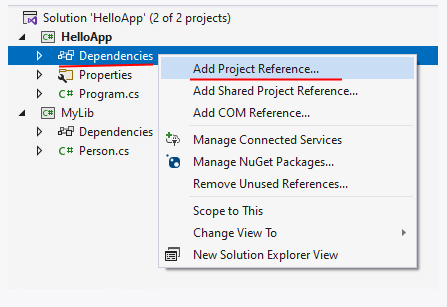
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class Person  {      string name;      public Person(string name)      {          this.name = name;      }      public void Print() => Console.WriteLine($"Name: {name}");  } |



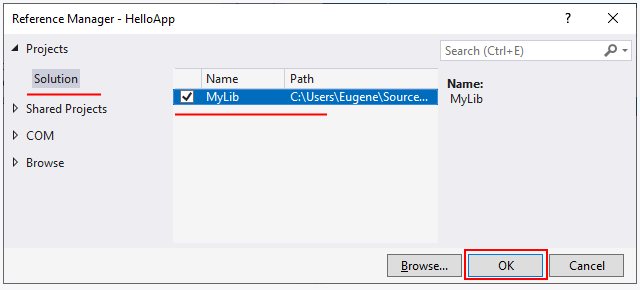
Теперь скомпилируем библиотеку классов. Для этого нажмем правой кнопкой на проект библиотеки классов и в контекстном меню выберем пункт **Rebuild**:



После компиляции библиотеки классов в папке проекта в каталоге *bin/Debug/net6.0* мы сможем найти скомпилированный файл dll (MyLib.dll). Подключим его в основной проект. Для этого в основном проекте нажмем правой кнопкой на узел **Dependencies** и в контекстном меню выберем пункт **Add Project Reference...**:



Далее нам откроется окно для добавления библиотек. В этом окне выберем пункт Solution,который позволяет увидеть все библиотеки классов из проектов текущего решения, поставим отметку рядом с нашей библиотекой и нажмем на кнопку OK:



Если наша библиотека вдруг представляет файл dll, который не связан ни с каким проектом в нашем решении, то с помощью кнопки **Browse** мы можем найти местоположение файла dll и также его подключить.

После успешного подключения библиотеки в главном проекте изменим файл **Program.cs**, чтобы он использовал класс Person из библиотеки классов:

|  |  |
| --- | --- |
| 1  2  3  4 | using MyLib;    // подключение пространства имен из библиотеки классов    Person tom = new("Tom");  tom.Print();    // Name: Tom |

## Модификаторы доступа

Все поля, методы и остальные компоненты класса имеют **модификаторы доступа**. Модификаторы доступа позволяют задать допустимую область видимости для компонентов класса. То есть модификаторы доступа определяют контекст, в котором можно употреблять данную переменную или метод.

В языке C# применяются следующие модификаторы доступа:

* **private**: закрытый или приватный компонент класса или структуры. Приватный компонент доступен только в рамках своего класса или структуры.
* **private protected**: компонент класса доступен из любого места в своем классе или в производных классах, которые определены в той же сборке.
* **protected**: такой компонент класса доступен из любого места в своем классе или в производных классах. При этом производные классы могут располагаться в других сборках.
* **internal**: компоненты класса или структуры доступен из любого места кода в той же сборке, однако он недоступен для других программ и сборок.
* **protected internal**: совмещает функционал двух модификаторов protected и internal. Такой компонент класса доступен из любого места в текущей сборке и из производных классов, которые могут располагаться в других сборках.
* **public**: публичный, общедоступный компонент класса или структуры. Такой компонент доступен из любого места в коде, а также из других программ и сборок.



Стоит отметить, что эти модификаторы могут применяться как к компонентам класса, так и к компонентам структуры за тем исключением, что структуры не могут использовать модификаторы private protected, protected и protected internal, поскольку структуры не могут быть унаследованы.

Все классы и структуры, определенные напрямую вне других типов (классов и структур) могут иметь только модификаторы public или internal.

Мы можем явно задать модификатор доступа, а можем его и не указывать:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class Person  {      string name;      public Person(string name)      {          this.name = name;      }      public void Print() => Console.WriteLine($"Name: {name}");  } |

Если для компонентов не определен модификатор доступа, то по умолчанию для них применяется модификатор **private**. Например, в примере выше переменная name неявно будет иметь модификатор **private**.

Классы и структуры, которые объявлены без модификатора и которые расположены вне других типов, по умолчанию имеют доступ **internal**, а вложенные классы и структуры, как и остальные компоненты классов/структур имеют модификатор **private**. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class Phone  {      struct Camera      {        }  } |

Здесь класс Phone не является вложенным ни в один другой класс/структуру, поэтому неявно имеет модификатор **internal**. А структура Camera является вложенной, поэтому, как и другие компоненты класса, неявно имеет модификатор **private**

### Модификаторы в рамках текущего проекта

Посмотрим на примере и создадим следующий класс State:

|  |  |
| --- | --- |
| 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 | class State  {      // все равно, что private string defaultVar;      string defaultVar ="default";      // поле доступно только из текущего класса      private string privateVar = "private";      // доступно из текущего класса и производных классов, которые определены в этом же проекте      protected private string protectedPrivateVar = "protected private";      // доступно из текущего класса и производных классов      protected string protectedVar = "protected";      // доступно в любом месте текущего проекта      internal string internalVar = "internal";      // доступно в любом месте текущего проекта и из классов-наследников в других проектах      protected internal string protectedInternalVar = "protected internal";      // доступно в любом месте программы, а также для других программ и сборок      public string publicVar = "public";        // по умолчанию имеет модификатор private      void Print() => Console.WriteLine(defaultVar);        // метод доступен только из текущего класса      private void PrintPrivate() => Console.WriteLine(privateVar);        // доступен из текущего класса и производных классов, которые определены в этом же проекте      protected private void PrintProtectedPrivate() => Console.WriteLine(protectedPrivateVar);        // доступен из текущего класса и производных классов      protected void PrintProtected() => Console.WriteLine(protectedVar);        // доступен в любом месте текущего проекта      internal void PrintInternal() => Console.WriteLine(internalVar);        // доступен в любом месте текущего проекта и из классов-наследников в других проектах      protected internal void PrintProtectedInternal() => Console.WriteLine(protectedInternalVar);        // доступен в любом месте программы, а также для других программ и сборок      public void PrintPublic() => Console.WriteLine(publicVar);  } |

Так как класс State не имеет явного модификатора, по умолчанию он имеет модификатор internal, поэтому он будет доступен из любого места данного проекта, однако не будет доступен из других программ и сборок.

Класс State имеет шесть полей для каждого уровня доступа. Плюс одна переменная без модификатора, которая является закрытой (private) по умолчанию. А также определено семь методов с разными модификаторами, которые выводят значения соответствующих переменных на консоль. Поскольку все модификаторы позволяют использовать компоненты класса внутри данного класса, то и все переменные класса, в том числе закрытые, у нас доступны всем его методам, так как все находятся в контексте класса State.

Теперь посмотрим, как мы сможем использовать переменные класса State в другом классе, который, допустим, будет называться StateConsumer и который расположен **в том же проекте**:

|  |  |
| --- | --- |
| 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 | class StateConsumer  {      public void PrintState()      {          State state = new State();            // обратиться к переменной defaultVar у нас не получится,          // так как она имеет модификатор private и класс StateConsumer ее не видит          Console.WriteLine(state.defaultVar); //Ошибка, получить доступ нельзя            // то же самое относится и к переменной privateVar          Console.WriteLine(state.privateVar); // Ошибка, получить доступ нельзя            // обратиться к переменной protectedPrivateVar не получится,          // так как класс StateConsumer не является классом-наследником класса State          Console.WriteLine(state.protectedPrivateVar); // Ошибка, получить доступ нельзя            // обратиться к переменной protectedVar тоже не получится,          // так как класс StateConsumer не является классом-наследником класса State          Console.WriteLine(state.protectedVar); // Ошибка, получить доступ нельзя            // переменная internalVar с модификатором internal доступна из любого места текущего проекта          // поэтому спокойно присваиваем ей значение          Console.WriteLine(state.internalVar);            // переменная protectedInternalVar так же доступна из любого места текущего проекта          Console.WriteLine(state.protectedInternalVar);            // переменная publicVar общедоступна          Console.WriteLine(state.publicVar);      }  } |

Таким образом, в классе StateConsumer мы смогли только обратиться к переменным internalVar, protectedInternalVar и publicVar, так как их модификаторы позволяют использовать в данном контексте.

Аналогично дело обстоит и с методами:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | class StateConsumer  {      public void PrintState()      {          State state = new State();            state.Print(); //Ошибка, получить доступ нельзя            state.PrintPrivate(); // Ошибка, получить доступ нельзя            state.PrintProtectedPrivate(); // Ошибка, получить доступ нельзя            state.PrintProtected(); // Ошибка, получить доступ нельзя            state.PrintInternal();    // норм            state.PrintProtectedInternal();  // норм            state.PrintPublic();      // норм      }  } |

Здесь нам оказались доступны только три метода: PrintInternal, PrintProtectedInternal, PrintPublic, которые имееют соответственно модификаторы internal, protected internal, public.

### **Модификаторы в рамках сборок**

Допустим, у нас есть проект (и соответственно сборка) MyLib, в которой определены три класса:

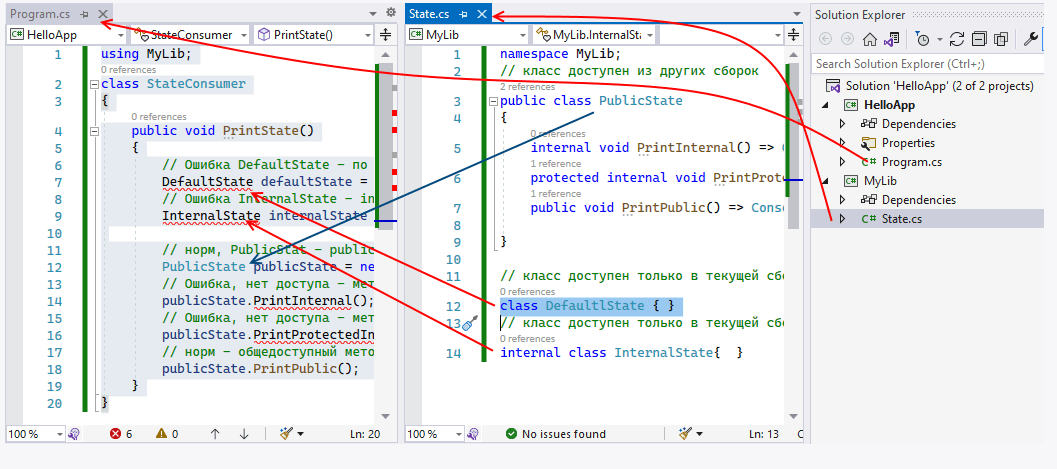
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | namespace MyLib;  // класс доступен из других сборок  public class PublicState  {      internal void PrintInternal() => Console.WriteLine("internal");      protected internal void PrintProtectedInternal() => Console.WriteLine("protected internal");      public void PrintPublic() => Console.WriteLine("public");    }    // класс доступен только в текущей сборке - по умолчанию internal  class DefaultlState { }  // класс доступен только в текущей сборке  internal class InternalState{  } |

Здесь классы DefaultlState и InternalState имеют модификатор **internal**, поэтому доступны только в текущем проекте.

Класс PublicState модификатором **public** доступен из других проектов. Однако его методы PrintInternal() и PrintProtectedInternal() доступны только в текущем проекте. Вне текущего проекта доступен только его метод PrintPublic.

Допустим, мы подключаем сборку этого проекта MyLib в другой проект, где есть класс StateConsumer:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | using MyLib;  class StateConsumer  {      public void PrintState()      {          // Ошибка DefaultState - по умолчанию internal, поэтому нет доступа          DefaultState defaultState = new DefaultState();          // Ошибка InternalState - internal, поэтому нет доступа          InternalState internalState = new InternalState();            // норм, PublicState - public, доступен из других программ          PublicState publicState = new PublicState();          // Ошибка, нет доступа - метод доступен только в свой сборке          publicState.PrintInternal();          // Ошибка, нет доступа - метод доступен только в свой сборке          publicState.PrintProtectedInternal();  // нет доступа          // норм - общедоступный метод          publicState.PrintPublic();      // норм      } |



В классе StateConsumer есть доступ только к классу PublicState и его методу PrintPublic, потому что они имеют модификатор **public**. К остальной функциональности подключенной сборки StateConsumer доступа не имеет.

Благодаря такой системе модификаторов доступа можно скрывать некоторые моменты реализации класса от других частей программы.

## Свойства

Кроме обычных методов в языке C# предусмотрены специальные методы доступа, которые называют **свойства**. Они обеспечивают простой доступ к полям классов и структур, узнать их значение или выполнить их установку.

### **Определение свойств**

Стандартное описание свойства имеет следующий синтаксис:

|  |  |
| --- | --- |
| 1  2  3  4  5 | [модификаторы] тип\_свойства название\_свойства  {      get { действия, выполняемые при получении значения свойства}      set { действия, выполняемые при установке значения свойства}  } |

Вначале определения свойства могут идти различные модификаторы, в частности, модификаторы доступа. Затем указывается тип свойства, после которого идет название свойства. Полное определение свойства содержит два блока: **get** и **set**.

В блоке **get** выполняются действия по получению значения свойства. В этом блоке с помощью оператора **return** возвращаем некоторое значение.

В блоке **set** устанавливается значение свойства. В этом блоке с помощью параметра **value** мы можем получить значение, которое передано свойству.

Блоки **get** и **set** еще называются акссесорами или методами доступа (к значению свойства), а также геттером и сеттером.

То есть по сути свойство ничего не хранит, оно выступает в роли посредника между внешним кодом и переменной name.

Рассмотрим пример:

|  |  |
| --- | --- |
| 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 | Person person = new Person();    // Устанавливаем свойство - срабатывает блок Set  // значение "Tom" и есть передаваемое в свойство value  person.Name = "Tom";    // Получаем значение свойства и присваиваем его переменной - срабатывает блок Get  string personName = person.Name;  Console.WriteLine(personName);  // Tom    class Person  {      private string name = "Undefined";        public string Name      {          get          {              return name;    // возвращаем значение свойства          }          set          {              name = value;   // устанавливаем новое значение свойства          }      }  } |

Здесь в классе Person определено приватное поле name, которая хранит имя пользователя, и есть общедоступное свойство Name. Хотя они имеют практически одинаковое название за исключением регистра, но это не более чем стиль, названия у них могут быть произвольные и не обязательно должны совпадать.

Через это свойство мы можем управлять доступом к переменной name. В свойстве в блоке **get** возвращаем значение поля:

|  |  |
| --- | --- |
| 1 | get { return name; } |

А в блоке **set** устанавливаем значение переменной name. Параметр **value** представляет передаваемое значение, которое передается переменной name.

|  |  |
| --- | --- |
| 1 | set { name = value; } |

В программе мы можем обращаться к этому свойству, как к обычному полю. Если мы ему присваиваем какое-нибудь значение, то срабатывает блок **set**, а передаваемое значение передается в параметр **value**:

|  |  |
| --- | --- |
| 1 | person.Name = "Tom"; |

Если мы получаем значение свойства, то срабатывает блок **get**, который по сути возвращает значение переменной name:

|  |  |
| --- | --- |
| 1 | string personName = p.Name; |

Возможно, может возникнуть вопрос, зачем нужны свойства, если мы можем в данной ситуации обходиться обычными полями класса? Но свойства позволяют вложить дополнительную логику, которая может быть необходима при установке или получении значения. Например, нам надо установить проверку по возрасту:

|  |  |
| --- | --- |
| 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 | Person person = new Person();    Console.WriteLine(person.Age);  // 1  // изменяем значение свойства  person.Age = 37;  Console.WriteLine(person.Age);  // 37  // пробуем передать недопустимое значение  person.Age = -23;               // Возраст должен быть в диапазоне от 1 до 120  Console.WriteLine(person.Age);  // 37 - возраст не изменился    class Person  {      int age = 1;      public int Age      {          set          {              if (value < 1 || value > 120)                  Console.WriteLine("Возраст должен быть в диапазоне от 1 до 120");              else                  age = value;          }          get { return age; }      }  } |

В данном случае переменная age хранит возраст пользователя. Напрямую мы не можем обратиться к этой переменной - только через свойство Age. Причем в блоке **set** мы устанавливаем значение, если оно соответствует некоторому разумному диапазону. Поэтому при передаче свойству Age значения, которое не входит в этот диапазон, значение переменной не будет изменяться:

|  |  |
| --- | --- |
| 1 | person.Age = -23; |

Консольный вывод программы:

1

37

Возраст должен быть в диапазоне от 1 до 120

37

Таким образом, свойство позволяет опосредовать и контролировать доступ к данным объекта.

### Свойства только для чтения и записи

Блоки set и get не обязательно одновременно должны присутствовать в свойстве. Если свойство определяет только блок **get**, то такое свойство доступно только **для чтения** - мы можем получить его значение, но не установить.

И, наоборот, если свойство имеет только блок **set**, тогда это свойство доступно только для записи - можно только установить значение, но нельзя получить:

|  |  |
| --- | --- |
| 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 | Person person = new Person();    // свойство для чтения - можно получить значение  Console.WriteLine(person.Name);  // Tom  // но нельзя установить  // person.Name = "Bob";    // ! Ошибка    // свойство для записи - можно устновить значение  person.Age = 37;  // но нелзя получить  // Console.WriteLine(person.Age);  // ! Ошибка    person.Print();    class Person  {      string name = "Tom";      int age = 1;      // свойство только для записи      public int Age      {          set { age = value; }      }      // свойство только для чтения      public string Name      {          get { return name; }      }        public void Print()=> Console.WriteLine($"Name: {name}  Age: {age}");  } |

Здесь свойство Name доступно только для чтения, поскольку оно имеет только блок **get**:

|  |  |
| --- | --- |
| 1  2  3  4 | public string Name  {      get { return name; }  } |

Мы можем получить его значение, но НЕ можем установить:

|  |  |
| --- | --- |
| 1  2 | Console.WriteLine(person.Name);  // получить можно  person.Name = "Bob";    // ! Ошибка - установить нельзя |

А свойство Age, наоборот, доступно только для записи, поскольку оно имеет только блок **set**:

|  |  |
| --- | --- |
| 1  2  3  4 | public int Age  {      set { age = value; }  } |

Можно установить его значение, но нельзя получить:

|  |  |
| --- | --- |
| 1  2 | person.Age = 37; // установить можно  Console.WriteLine(person.Age);  // ! Ошибка - получить значение нельзя |

### **Вычисляемые свойства**

Свойства необзательно связаны с определенной переменной. Они могут вычисляться на основе различных выражений

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | Person tom = new("Tom", "Smith");  Console.WriteLine(tom.Name);    // Tom Smith  class Person  {      string firstName;      string lastName;      public string Name      {          get { return  $"{firstName} {lastName}"; }      }      public Person(string firstName, string lastName)      {          this.firstName = firstName;          this.lastName = lastName;      }  } |

В данном случае класс Person имеет свойство Name, которое доступно только для чтения и которое возвращает общее значение на основе значений переменных firstName и lastName.

### **Модификаторы доступа**

Мы можем применять модификаторы доступа не только ко всему свойству, но и к отдельным блокам get и set:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | Person tom = new("Tom");    // Ошибка - set объявлен с модификатором private  //tom.Name = "Bob";  Console.WriteLine(tom.Name);    // Tom  class Person  {      string name = "";      public string Name      {          get { return name; }            private set { name = value; }      }      public Person(string name) => Name = name;  } |

Теперь закрытый блок set мы сможем использовать только в данном классе - в его методах, свойствах, конструкторе, но никак не в другом классе:

При использовании модификаторов в свойствах следует учитывать ряд ограничений:

* Модификатор для блока set или get можно установить, если свойство имеет оба блока (и set, и get)
* Только один блок set или get может иметь модификатор доступа, но не оба сразу
* Модификатор доступа блока set или get должен быть более ограничивающим, чем модификатор доступа свойства. Например, если свойство имеет модификатор public, то блок set/get может иметь только модификаторы protected internal, internal, protected, private protected и private

### **Автоматические свойства**

Свойства управляют доступом к полям класса. Однако что, если у нас с десяток и более полей, то определять каждое поле и писать для него однотипное свойство было бы утомительно. Поэтому в .NET были добавлены автоматические свойства. Они имеют сокращенное объявление:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | class Person  {      public string Name { get; set; }      public int Age { get; set; }        public Person(string name, int age)      {          Name = name;          Age = age;      }  } |

На самом деле тут также создаются поля для свойств, только их создает не программист в коде, а компилятор автоматически генерирует при компиляции.

В чем преимущество автосвойств, если по сути они просто обращаются к автоматически создаваемой переменной, почему бы напрямую не обратиться к переменной без автосвойств? Дело в том, что в любой момент времени при необходимости мы можем развернуть автосвойство в обычное свойство, добавить в него какую-то определенную логику.

Стоит учитывать, что нельзя создать автоматическое свойство только для записи, как в случае со стандартными свойствами.

Автосвойствам можно присвоить значения по умолчанию (инициализация автосвойств):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | Person tom = new();    Console.WriteLine(tom.Name);    // Tom  Console.WriteLine(tom.Age);    // 37    class Person  {      public string Name { get; set; } = "Tom";      public int Age { get; set; } = 37;  } |

И если мы не укажем для объекта Person значения свойств Name и Age, то будут действовать значения по умолчанию.

Автосвойства также могут иметь модификаторы доступа:

|  |  |
| --- | --- |
| 1  2  3  4  5 | class Person  {      public string Name { private set; get;}      public Person(string name) => Name = name;  } |

Мы можем убрать блок set и сделать автосвойство доступным только для чтения. В этом случае для хранения значения этого свойства для него неявно будет создаваться поле с модификатором readonly, поэтому следует учитывать, что подобные get-свойства можно установить либо из конструктора класса, как в примере выше, либо при инициализации свойства:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class Person  {      // через инициализацию свойства      public string Name { get; } = "Tom";      // через конструктор      public Person(string name) => Name = name;  } |

### Блок init

Начиная с версии C# 9.0 сеттеры в свойствах могут определяться с помощью оператора **init** (от слова "инициализация" - это есть блок init призван инициализировать свойство). Для установки значений свойств с **init** можно использовать только инициализатор, либо конструктор, либо при объявлении указать для него значение. После инициализации значений подобных свойств их значения изменить нельзя - они доступны только для чтения. В этом плане init-свойства сближаются со свойствами для чтения. Разница состоит в том, что init-свойства мы также можем установить в инициализаторе (свойства для чтения установить в инициализаторе нельзя). Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Person person = new();  //person.Name = "Bob";    //! Ошибка - после инициализации изменить значение нельзя    Console.WriteLine(person.Name); // Undefined  public class Person  {      public string Name { get; init; } = "Undefined";  } |

В данном случае класс Person для свойства Name вместо сеттера использует оператор **init**. В итоге на строке

|  |  |
| --- | --- |
| 1 | Person person = new(); |

предполагается создание объекта с инициализацией всех его свойств. В данном случае свойство Name получит в качестве значения строку "Undefined". Однако поскольку инициализация свойства уже произошла, то на строке

|  |  |
| --- | --- |
| 1 | person.Name = "Bob";    // Ошибка |

мы получим ошибку.

Как можно установить подобное свойство? Выше продемонстрирован один из способов - установка значения при определении свойства. Второй способ - через конструктор:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Person person = new("Tom");  Console.WriteLine(person.Name); // Tom  public class Person  {      public Person(string name) => Name = name;      public string Name { get; init; }  } |

Третий способ - через инициализатор:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Person person = new() { Name = "Bob"};  Console.WriteLine(person.Name); // Bob    public class Person  {      public string Name { get; init; } = "";  } |

В принцпе есть еще четверый способ - установка через другое свойство с модификатором **init**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | var person = new Person() { Name = "Sam" };  Console.WriteLine(person.Name);     // Sam  Console.WriteLine(person.Email);    // Sam@gmail.com  public class Person  {      string name = "";      public string Name      {          get { return name; }          init          {              name = value;              Email = $"{value}@gmail.com";          }      }      public string Email { get; init; } = "";  } |

В данном случае в init-свойстве Name разворачивается в полное свойство, которое управляет полем для чтения name. Благодаря этому перед установкой значения свойства мы можем произвести некоторую предобработку. Кроме того, в выражении init установливается другое init-свойство - Email, которое для установки значения использует значение свойства Name - из имени получаем значение для электронного адреса.

Причем если если при объявлении свойства указано значение, то в конструкторе мы можем его изменить. Значение, установленное в конструкторе, можно изменить в инициализаторе. Однако дальше процесс инициализации заканчивается. И значение не может быть изменено.

### Сокращенная запись свойств

Как и методы, мы можем сокращать определения свойств. Поскольку блоки **get** и **set** представляют специальные методы, то как и обычные методы, если они содержат одну инструкцию, то мы их можем сократить с помощью оператора **=>**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class Person  {      string name;      public string Name      {          get => name;          set => name = value;      }  } |

Также можно сокращать все свойство в целом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class Person  {      string name;        // эквивалентно public string Name { get { return name; } }      public string Name => name;  } |

## Перегрузка методов

Иногда возникает необходимость создать один и тот же метод, но с разным набором параметров. И в зависимости от имеющихся параметров применять определенную версию метода. Такая возможность еще называется **перегрузкой методов** (method overloading).

И в языке C# мы можем создавать в классе несколько методов с одним и тем же именем, но разной сигнатурой. Что такое сигнатура? **Сигнатура** складывается из следующих аспектов:

* Имя метода
* Количество параметров
* Типы параметров
* Порядок параметров
* Модификаторы параметров

Но названия параметров в сигнатуру НЕ входят. Например, возьмем следующий метод:

|  |  |
| --- | --- |
| 1  2  3  4 | public int Sum(int x, int y)  {      return x + y;  } |

У данного метода сигнатура будет выглядеть так: Sum(int, int)

И перегрузка метода как раз заключается в том, что методы имеют разную сигнатуру, в которой совпадает только название метода. То есть методы должны отличаться по:

* Количеству параметров
* Типу параметров
* Порядку параметров
* Модификаторам параметров

Например, пусть у нас есть следующий класс:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | class Calculator  {      public void Add(int a, int b)      {          int result = a + b;          Console.WriteLine($"Result is {result}");      }      public void Add(int a, int b, int c)      {          int result = a + b + c;          Console.WriteLine($"Result is {result}");      }      public int Add(int a, int b, int c, int d)      {          int result = a + b + c + d;          Console.WriteLine($"Result is {result}");          return result;      }      public void Add(double a, double b)      {          double result = a + b;          Console.WriteLine($"Result is {result}");      }  } |

Здесь представлены четыре разных версии метода Add, то есть определены четыре перегрузки данного метода.

Первые три версии метода отличаются по количеству параметров. Четвертая версия совпадает с первой по количеству параметров, но отличается по их типу. При этом достаточно, чтобы хотя бы один параметр отличался по типу. Поэтому это тоже допустимая перегрузка метода Add.

То есть мы можем представить сигнатуры данных методов следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4 | Add(int, int)  Add(int, int, int)  Add(int, int, int, int)  Add(double, double) |

После определения перегруженных версий мы можем использовать их в программе:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Calculator calc = new Calculator();  calc.Add(1, 2); // 3  calc.Add(1, 2, 3); // 6  calc.Add(1, 2, 3, 4); // 10  calc.Add(1.4, 2.5); // 3.9 |

Консольный вывод:

Result is 3

Result is 6

Result is 10

Result is 3.9

Также перегружаемые методы могут отличаться по используемым модификаторам. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | void Increment(ref int val)  {      val++;      Console.WriteLine(val);  }    void Increment(int val)  {      val++;      Console.WriteLine(val);  } |

В данном случае обе версии метода Increment имеют одинаковый набор параметров одинакового типа, однако в первом случае параметр имеет модификатор ref. Поэтому обе версии метода будут корректными перегрузками метода Increment.

А отличие методов по возвращаемому типу или по имени параметров не является основанием для перегрузки. Например, возьмем следующий набор методов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | int Sum(int x, int y)  {      return x + y;  }  int Sum(int number1, int number2)  {      return number1 + number2;  }  void Sum(int x, int y)  {      Console.WriteLine(x + y);  } |

Сигнатура у всех этих методов будет совпадать:

|  |  |
| --- | --- |
| 1 | Sum(int, int) |

Поэтому данный набор методов не представляет корректные перегрузки метода Sum и **работать не будет**.

## Статические члены и модификатор static

Кроме обычных полей, методов, свойств классы и структуры могут иметь статические поля, методы, свойства. Статические поля, методы, свойства относятся ко всему классу/всей структуре и для обращения к подобным членам необязательно создавать экземпляр класса / структуры.

### **Статические поля**

Статические поля хранят состояние всего класса / структуры. Статическое поле определяется как и обычное, только перед типом поля указывается ключевое слово **static**. Например, рассмотрим класс Person, который представляет человека:

|  |  |
| --- | --- |
| 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 | Person bob = new(68);  bob.СheckAge();     // Уже на пенсии    Person tom = new(37);  tom.СheckAge();     // Сколько лет осталось до пенсии: 28    // получение статического поля  Console.WriteLine(Person.retirementAge); // 65  // изменение статического поля  Person.retirementAge = 67;    class Person  {      int age;      public static int retirementAge = 65;      public Person(int age)      {          this.age = age;      }      public void СheckAge()      {          if (age >= retirementAge)              Console.WriteLine("Уже на пенсии");          else              Console.WriteLine($"Сколько лет осталось до пенсии: {retirementAge - age}");      }  } |

В данном случае класс Person имеет два поля: age (хранит возраст человека) и retirementAge (хранит пенсионный возраст). Однако поле retirementAge является статическим. Оно относится не к конкретному человеку, а ко всем людям. (В данном случае для упрощения пренебрежем тем фактом, что в зависимости от пола и профессии пенсионный возраст может отличаться.) Таким образом, поле retirementAge относится не к отдельную объекту и хранит значение НЕ отдельного объекта класса Person, а относится ко всему классу Person и хранит общее значение для всего класса.

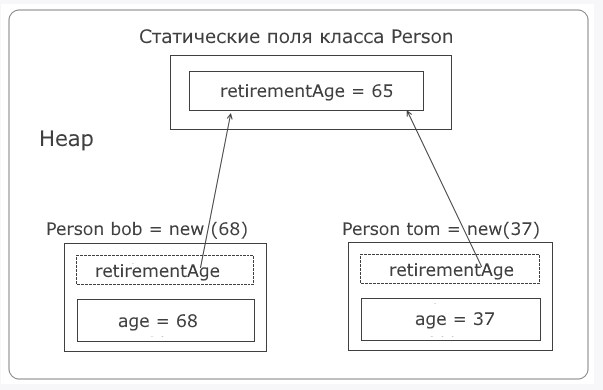
Причем в самом классе мы можем использовать это поле как и любые другие. Так, в методе СheckAge(), который поверяет пенсионный статус человека, для проверки используем это поле:

|  |  |
| --- | --- |
| 1 | if (age >= retirementAge) |

Но если мы хотим обратиться к этому полю вне своего класса, то мы можем обращаться к этому полю по имени класса:

|  |  |
| --- | --- |
| 1  2 | Console.WriteLine(Person.retirementAge);  Person.retirementAge = 67; |

На уровне памяти для статических полей будет создаваться участок в памяти, который будет общим для всех объектов класса.



При этом память для статических переменных выделяется даже в том случае, если не создано ни одного объекта этого класса.

### **Статические свойства**

Подобным образом мы можем создавать и использовать статические свойства:

|  |  |
| --- | --- |
| 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 | Person bob = new(68);  bob.СheckAge();    Console.WriteLine(Person.RetirementAge); // 65    class Person  {      int age;      static int retirementAge = 65;      public static int RetirementAge      {          get { return retirementAge; }          set { if (value > 1 && value < 100) retirementAge = value; }      }      public Person(int age)      {          this.age = age;      }      public void СheckAge()      {          if (age >= retirementAge)              Console.WriteLine("Уже на пенсии");          else              Console.WriteLine($"Сколько лет осталось до пенсии: {retirementAge - age}") ;      }  } |

В данном случае доступ к статической переменной retirementAge опосредуется с помощью статического свойства RetirementAge.

Таким образом, переменные и свойства, которые хранят состояние, общее для всех объектов класса / структуры, следует определять как статические.

Нередко статические поля и свойства применяются для хранения счетчиков. Например, мы хотим иметь счетчик, который позволял бы узнать, сколько объектов Person создано:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | var tom = new Person();  var bob = new Person();  var sam = new Person();  Console.WriteLine(Person.Counter);  // 3    class Person  {      static int counter = 0;      public static int Counter => counter;      public Person()      {          counter++;      }  } |

В данном случае в классе Person счетчик хранится в приватной переменной counter, значение которой увеличивается на единицу при создании объекта в конструкторе. А с помощью статического свойства Counter, которое доступно только для чтения, мы можем получить значение счетчика.

### **Статические методы**

Статические методы определяют общее для всех объектов поведение, которое не зависит от конкретного объекта. Для обращения к статическим методам также применяется имя класса / структуры:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | Person bob = new(68);  Person.CheckRetirementStatus(bob);    class Person  {      public int Age { get; set; }      static int retirementAge = 65;      public Person(int age) => Age = age;      public static void CheckRetirementStatus(Person person)      {          if (person.Age >= retirementAge)              Console.WriteLine("Уже на пенсии");          else              Console.WriteLine($"Сколько лет осталось до пенсии: {retirementAge - person.Age}") ;      }  } |

В данном случае в классе Person определен статический метод CheckRetirementStatus(), который в качестве параметра принимает объект Person и проверяет его пенсионный статус.

Следует учитывать, что статические методы могут обращаться только к статическим членам класса. Обращаться к нестатическим методам, полям, свойствам внутри статического метода мы не можем.

### **Статический конструктор**

Кроме обычных конструкторов у класса также могут быть статические конструкторы. Статические конструкторы имеют следующие отличительные черты:

* Статические конструкторы не должны иметь модификатор доступа и не принимают параметров
* Как и в статических методах, в статических конструкторах нельзя использовать ключевое слово this для ссылки на текущий объект класса и можно обращаться только к статическим членам класса
* Статические конструкторы нельзя вызвать в программе вручную. Они выполняются автоматически при самом первом создании объекта данного класса или при первом обращении к его статическим членам (если таковые имеются)

Статические конструкторы обычно используются для инициализации статических данных, либо же выполняют действия, которые требуется выполнить только один раз

Определим статический конструктор:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | Console.WriteLine(Person.RetirementAge);    class Person  {      static int retirementAge;      public static int RetirementAge => retirementAge;      static Person()      {          if (DateTime.Now.Year == 2022)              retirementAge = 65;          else              retirementAge = 67;      }  } |

В данном случае с помощью встроенной структуры **DateTime** получаем текущий год. Для этого используется свойство DateTime.Now.Year. если он равен 2022, устанавливаем один пенсионный возраст. При другом значении текущего года устанавливается другое значение пенсионного возраста.

### **Статические классы**

Статические классы объявляются с модификатором static и могут содержать только статические поля, свойства и методы. Например, определим класс, который выполняет ряд арифметических операций:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | Console.WriteLine(Operations.Add(5, 4));         // 9  Console.WriteLine(Operations.Subtract(5, 4));    // 1  Console.WriteLine(Operations.Multiply(5, 4));    // 20    static class Operations  {      public static int Add(int x, int y) => x + y;      public static int Subtract(int x, int y) => x - y;      public static int Multiply(int x, int y) => x \* y;  } |

## Поля и структуры для чтения

### **Поля для чтения и модификатор readonly**

**Поля для чтения** представляют такие поля класса или структуры, значение которых нельзя изменить. Таким полям можно присвоить значение либо при непосредственно при их объявлении, либо в конструкторе. В других местах программы присваивать значение таким полям нельзя, можно только считывать их значение.

Поле для чтения объявляется с ключевым словом **readonly**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | Person tom = new Person("Tom");  Console.WriteLine(tom.name);    //tom.name = "Sam"; // !Ошибка: нельзя изменить  class Person  {      public readonly string name = "Undefined"; // можно так инициализировать      public Person(string name)      {          this.name = name; // в конструкторе также можно присвоить значение полю для чтения      }      public void ChangeName(string otherName)      {          //this.name = otherName;    // так нельзя      }  } |

Здесь поле name хранит имя пользователя и представляет поле для чтения. Ему можно присвоить значение непосредственно при объявлении:

|  |  |
| --- | --- |
| 1 | public readonly string name = "Undefined"; |

Либо можно присвоить значение в конструкторе

|  |  |
| --- | --- |
| 1  2  3  4 | public Person(string name)  {      this.name = name;  } |

В остальных местах программы, в том числе в самом классе Person присвоить значение полю для чтения нельзя.

### **Сравнение констант**

* Константы должны быть определены во время компиляции, а поля для чтения могут быть определены во время выполнения программы.

Соответственно значение константы можно установить только при ее определении.

Поле для чтения можно инициализировать либо при его определении, либо в конструкторе класса.

* Константы не могут использовать модификатор static, так как уже неявно являются статическими. Поля для чтения могут быть как статическими, так и не статическими.

### **Структуры для чтения**

Кроме полей для чтения в C# можно определять структуры для чтения. Для этого они предваряются модификатором **readonly**:

|  |  |
| --- | --- |
| 1 | readonly struct Person { } |

Особенностью таких структур является то, что все их поля должны быть также полями для чтения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | readonly struct Person  {      public readonly string name;      public Person(string name)      {          this.name = name;      }  } |

То же самое касается и свойств, которые должны быть доступны только для чтения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | readonly struct Person  {      public readonly string Name { get; } // указывать readonly необязательно      public int Age { get; } // свойство только для чтения      public Person(string name, int age)      {          Name = name;          Age = age;      }  } |

## Null и ссылочные типы

Кроме стандартных значений типа чисел, строк, язык C# имеет специальное значение - **null**, которое фактически указывает на отсутствие значения как такового, отсутствие данных. До сих пор значение **null** выступает как значение по умолчанию для ссылочных типов.

До версии C# 8.0 всем ссылочным типам спокойно можно было присваивать значение **null**:

|  |  |
| --- | --- |
| 1  2 | string name = null;  Console.WriteLine(name); |

Но начиная с версии C# 8.0 в язык была введена концепция **ссылочных nullable-типов** (nullable reference types) и **nullable aware context** - nullable-контекст, в котором можно использовать ссылочные nullable-типы.

Чтобы определить переменную/параметр ссылочного типа, как переменную/параметр, которым можно присваивать значение **null**, после названия типа указывается знак вопроса **?**

|  |  |
| --- | --- |
| 1  2 | string? name = null;  Console.WriteLine(name);    // ничего не выведет |

К примеру встроенный метод Console.ReadLine(). который считывает с консоли строку, возвращает именно значение **string?**, а не просто string:

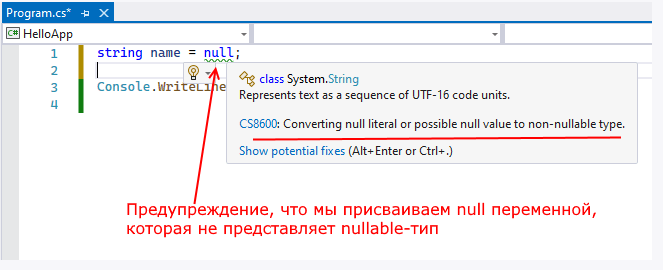
|  |  |
| --- | --- |
| 1 | string? name = Console.ReadLine(); |

Зачем нужно это значение null? В различных ситуациях бывает удобно, чтобы объекты могли принимать значение null, то есть были бы не определены. Стандартный пример - работа с базой данных, которая может содержать значения null. И мы можем заранее не знать, что мы получим из базы данных - какое-то определенное значение или же null.

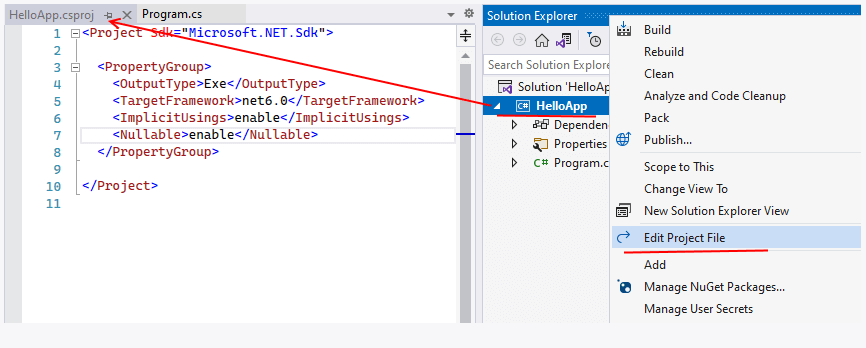
При этом подобные ссылочные типы, которые допускают присвоение значения null, доступно только в nullable-контексте. Для nullable-контекста характерны следующие особенности:

* Переменную ссылочного типа следует инициализировать конкретным значением, ей не следует присваивать значение **null**
* Переменной ссылочного nullable-типа можно присвоить значение **null**, но перед использование необходимо проверять ее на значение **null**.

Начиная с .NET 6 и C# 10 nullable-контекст по умолчанию распространяется на все файлы кода в проекта. Например, если мы наберем в Visual Studio 2022 для проекта .NET 6 предыдущий пример, то мы столкнемся с предупреждением:



Хотя nullable-контекст - это опция, которой мы можем управлять. Так, откроем файл проекта. Для этого либо двойным кликом левой кнопкой мыши нажмем на проект, либо нажмем на проект правой кнопкой мыши и в появившемся меню выберем пункт **Edit Project File**



После этого Visual Studio откроет нам файл проекта, который будет выглядеть примерно следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | <Project Sdk="Microsoft.NET.Sdk">      <PropertyGroup>      <OutputType>Exe</OutputType>      <TargetFramework>net6.0</TargetFramework>      <ImplicitUsings>enable</ImplicitUsings>      <Nullable>enable</Nullable>    </PropertyGroup>    </Project> |

Здесь строка

|  |  |
| --- | --- |
| 1 | <Nullable>enable</Nullable> |

точнее элемент <Nullable> со значением **enable** указывает, что эта nullable-контекст будет распространяться на весь проект.

Чем так плох **null**? Дело в том, что это значение означает, отсутствие данных. Но, допустим, у нас есть ситуация, когда мы получаем извне некоторую строку и пытаемся обратиться к ее функциональности. Например, в примере ниже у строки вызывается метод ToUpper(), который переводит все символы строки в верхний регистр:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | string name = null;  PrintUpper(name);  // ! NullReferenceException    void PrintUpper(string text)  {      Console.WriteLine(text.ToUpper());  } |

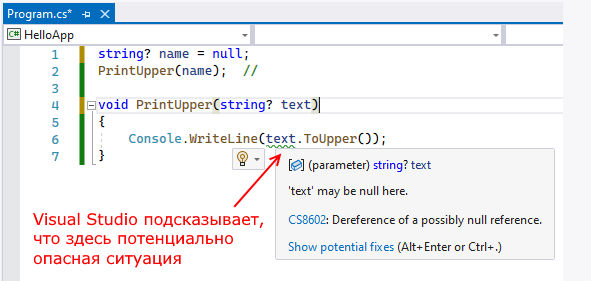
Здесь при выполнении вызова PrintUpper(name) мы столкнемся с исключением **NullReferenceException**, и программа аварийно завершит свою работу. Кто-то может сказать, что ситуация искуственная - мы же явно знаем, что в функцию передается null. Однако в реальности данные могут приходить извне, например, из базы данных, откуда-то из сети и т.д. И мы можем явно не знать, есть ли в реальности данные или нет. И использование ссылочных nullable-типов позволяет частично решить эту ситуацию. Частично - поскольку предупреждения все равно не мешают нам скомпилировать и запустить программу выше. Однако nullable-контекст позволяет воспользоваться возможностями статического анализа, благодаря которому можно увидеть потентиально опасные куски кода, где мы можем столкнуться с NullReferenceException.

Кроме того, есть вероятность, что Microsoft изменит отношение в отношении null и NullReferenceException, и подобные предупреждения превратятся в будущих версиях в ошибки, поэтому лучше уже сейчас быть к этому готовым

Например, изменим предыдущий пример следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | string? name = null;  PrintUpper(name);  //    void PrintUpper(string? text)  {      Console.WriteLine(text.ToUpper());  } |

Здесь статический анализ подскажет, что в методе PrintUpper потенциально опасная ситуация, поскольку параметр text может быть равен **null**.



#### **Отключение nullable-контекста**

Для отключения nullable-контекста в файле конфигурации проекта достаточно изменить значение опции **Nullable**, например, на "disable":

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | <Project Sdk="Microsoft.NET.Sdk">      <PropertyGroup>      <OutputType>Exe</OutputType>      <TargetFramework>net6.0</TargetFramework>      <ImplicitUsings>enable</ImplicitUsings>      <Nullable>disable</Nullable>    </PropertyGroup>    </Project> |

Отключив nullable-контекст, мы больше не сможем использовать в файлах кода в проекте ссылочные nullable-типы и соответственно воспользоваться встроенным статическим анализом потенциально опасных ситуаций, где можно столкнуться с NullReferenceException.

#### **nullable-контекст на уровне участка кода**

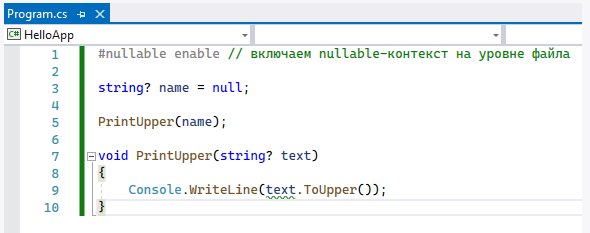
Мы также можем включить nullable-контекст на урове отдельных участков кода с помощью директивы **#nullable enable**. Допустим, глобально у нас отключен nullable-контекст:

|  |  |
| --- | --- |
| 1 | <Nullable>disable</Nullable> |

Определим в файле **Program.cs** следующий код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | #nullable enable // включаем nullable-контекст на уровне файла    string? name = null;    PrintUpper(name);    void PrintUpper(string? text)  {      Console.WriteLine(text.ToUpper());  } |

Первая строка позволяет включить на уровне всего файла nullable-контекст.



### **Оператор ! (null-forgiving operator)**

Оператор **!** (null-forgiving operator) позволяет указать, что переменная ссылочного типа не равна **null**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | string? name = null;    PrintUpper(name!);    void PrintUpper(string text)  {      if(text == null) Console.WriteLine("null");      else Console.WriteLine(text.ToUpper());  } |

Здесь если бы мы не использовали оператор !, а написали бы PrintUpper(name), то компилятор высветил бы нам предупреждение. Но в самом методе мы итак проверяем на null, поэтому даже если в метод передается null, то мы не столкнемся ни с какими проблемами. И чтобы убрать ненужное предупреждение, применяется данный оператор. То есть данный оператор не оказывает никакого влияния во время выполнения кода и предназначен только для статического анализа компилятора. Во время выполнения выражение name! будет аналогично значению name

### **Исключение кода из nullable-контекста**

С помощью специальной директивы **#nullable disable** можно исключить какой-то определенный кусок кода из nullable-контекста. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5 | #nullable disable      string text = null; // здесь nullable-контекст не действует  #nullable restore    string? name = null;   // здесь nullable-контекст снова действует |

Любой код между директивами **#nullable disable** и **#nullable restore** будет исключен из nullable-контекста и тем самым не будет подлежать статическому анализу.

## Null и значимые типы

В отличие от ссылочных типов переменным/параметрам значимых типов нельзя напрямую присвоить значение **null**. Тем не менее нередко бывает удобно, чтобы переменная/параметр значимого типа могли принимать значение null, Например, получаем числовое значение из базы данных, которое в бд может отсутствовать. То есть, если значение в базе данных есть - получим число, если нет - то null.

Чтобы присвоения переменной или параметру значимого типа значения **null**, эти переменная/параметр значимого типа должны представлять тип **nullable**. Для этого после названия типа указывается знак вопроса **?**

|  |  |
| --- | --- |
| 1  2 | int? val = null;  Console.WriteLine(val); |

Здесь переменная val представляет не просто тип int, а тип **int?** - тип, переменные/параметры которого могут принимать как значения типа int, так и значение null. В данном случае мы передаем ей значение **null**. Но также можно передать и значение типа int:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | int? val = null;  IsNull(val);    // null  val = 22;  IsNull(val);    // 22    void IsNull(int? obj)  {      if (obj == null) Console.WriteLine("null");      else Console.WriteLine(obj);  } |

Однако если переменная/параметр представляет значимый не nullable-тип, то присвоить им значение **null** не получится:

|  |  |
| --- | --- |
| 1 | int val = null;  // ! ошибка, переменная val НЕ представляет тип nullable |

Стоит отметить, что фактически запись ? для значимых типов является упрощенной формой использования структуры **System.Nullable<T>**. Параметр T в угловых скобках представляет универсальный параметр, вместо которого в программе подставляется конкретный тип данных. Следующие виды определения переменных будут эквивалентны:

|  |  |
| --- | --- |
| 1  2 | int? number1 = 5;  Nullable<int> number2 = 5; |

#### **Свойства Value и HasValue и метод GetValueOrDefault**

Структура **Nullable<T>** имеет два свойства:

* **Value** - значение объекта
* **HasValue**: возвращает true, если объект хранит некоторое значение, и false, если объект равен null.

Мы можем использовать эти свойства для проверки наличия и получения значения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | PrintNullable(5);       // 5  PrintNullable(null);    // параметр равен null    void PrintNullable(int? number)  {      if (number.HasValue)      {          Console.WriteLine(number.Value);          // аналогично          Console.WriteLine(number);      }      else      {          Console.WriteLine("параметр равен null");      }  } |

Однако если мы попробуем получить через свойство Value значение переменной, которая равна null, то мы столкнемся с ошибкой:

|  |  |
| --- | --- |
| 1  2  3 | int? number = null;  Console.WriteLine(number.Value);    // ! Ошибка  Console.WriteLine(number);          // Ошибки нет - просто ничего не выведет |

Также структура **Nullable<T>** имеет метод **GetValueOrDefault()**. Он возвращает значение переменной/параметра, если они не равны **null**. Если они равны **null**, то возвращается значение по умолчанию. Значение по умолчанию можно передать в метод. Если в метод не передается данных, то возвращается значение по умолчанию для данного типа данных (например, для числовых данных это число 0).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | int? number = null; // если значения нет, метод возвращает значение по умолчанию  Console.WriteLine(number.GetValueOrDefault());      // 0  - значение по умолчанию для числовых типов  Console.WriteLine(number.GetValueOrDefault(10));    // 10    number = 15;    // если значение задано, оно возвращается методом  Console.WriteLine(number.GetValueOrDefault());    // 15  Console.WriteLine(number.GetValueOrDefault(10));  // 15 |

### Преобразование значимых nullable-типов

Рассмотрим возможные преобразования:

* явное преобразование от T? к T

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | int? x1 = null;  if(x1.HasValue)  {      int x2 = (int)x1;      Console.WriteLine(x2);  } |

* неявное преобразование от T к T?

|  |  |
| --- | --- |
| 1  2  3 | int x1 = 4;  int? x2 = x1;  Console.WriteLine(x2); |

* неявные расширяющие преобразования от V к T?

|  |  |
| --- | --- |
| 1  2  3 | int x1 = 4;  long? x2 = x1;  Console.WriteLine(x2); |

* явные сужающие преобразования от V к T?

|  |  |
| --- | --- |
| 1  2 | long x1 = 4;  int? x2 = (int?)x1; |

* Подобным образом работают явные сужающие преобразования от V? к T?

|  |  |
| --- | --- |
| 1  2 | long? x1 = 4;  int? x2 = (int?)x1; |

* явные сужающие преобразования от V? к T

|  |  |
| --- | --- |
| 1  2  3  4  5 | long? x1 = null;  if (x1.HasValue)  {      int x2 = (int)x1;  } |

### Операции с nullable-типами

nullable-типы поддерживают тот же набор операций, что и их не-nullable двойники. Но следует учитывать, что если в операции участвует nullable-тип, то результатом также будет значение nullable-типа

|  |  |
| --- | --- |
| 1  2  3  4 | int? x = 5;  int z = x + 7;          // нельзя  int? w = x + 7;         // можно  int d = x.Value + 7;    // можно |

В арифметических операциях, если один из операндов равен null, то результатом операции также будет null:

|  |  |
| --- | --- |
| 1  2 | int? x = null;  int? w = x + 7;         // w = null |

В операциях сравнения, если хотя бы один из операндов равен null, то возвращается false

## Проверка на null, операторы ?. и ??

### Проверка на null. Null guard

Если мы собираемся использовать переменную или параметр, которые допускают значение **null**, то есть представляют nullable-тип (не важно значимый или ссылочный), то, чтобы избежать возникновения NullReferenceException, мы можем проверить на null:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | void PrintUpper(string? text)  {      if (text!=null)      {          Console.WriteLine(text.ToUpper());      }  } |

В данном случае если параметр text не равен null, то вызываем у строки метод ToUpper(), который переводит символы строки в верхний регистр.

Кроме того, с помощью оператора **is** мы можем проверить значение объекта:

|  |  |
| --- | --- |
| 1 | объект is значение |

Если объект слева от оператора **is** имеет значение справа от оператора. тогда оператор is возвращает true, иначе возвращается false

Например, проверка параметра/переменной на значение null:

|  |  |
| --- | --- |
| 1  2  3  4  5 | void PrintUpper(string? text)  {      if (text is  null) return;      Console.WriteLine(text.ToUpper());  } |

Или, наоборот, с помощью **is not** можно проверить отсутствие значения:

|  |  |
| --- | --- |
| 1  2  3  4  5 | void PrintUpper(string? text)  {      if (text is not null)          Console.WriteLine(text.ToUpper());  } |

Также можно проверить на соответствие типу, значение которого мы собираемся использовать:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | void PrintUpper(string? text)  {      if (text is string)          Console.WriteLine(text.ToUpper());      else          Console.WriteLine("NULL");  } |

Подобные проверки еще называются **null guard** или условно говоря "защита от null".

### **Оператор ??**

Оператор ?? называется оператором **null-объединения**. Он применяется для установки значений по умолчанию для типов, которые допускают значение null:

|  |  |
| --- | --- |
| 1 | левый\_операнд ?? правый\_операнд |

Оператор ?? возвращает левый операнд, если этот операнд не равен null. Иначе возвращается правый операнд. При этом левый операнд должен принимать null. Посмотрим на примере:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | string? text = null;  string name = text ?? "Tom";  // равно Tom, так как text равен null  Console.WriteLine(name);    // Tom    int? id = 200;  int personid = id ?? 1; // равно 200, так как id не равен null  Console.WriteLine(personid);    // 200 |

Но мы не можем написать следующим образом:

|  |  |
| --- | --- |
| 1  2 | int x = 44;  int y = x ?? 100; |

Здесь переменная x представляет значимый тип int и не может принимать значение null, поэтому в качестве левого операнда в операции ?? она использоваться не может.

Также можно использовать производный оператора **??=**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | string? text = null;  text ??= "Sam";  // аналогично  // text = text ?? "Sam";  Console.WriteLine(text);    // Sam    int? id = 100;  id ??= 1;  // аналогично  //id = id ?? 1;  Console.WriteLine(id); // 100 |

### Оператор условного null

Иногда при работе с объектами, которые принимают значение null, мы можем столкнуться с ошибкой: мы пытаемся обратиться к объекту, а этот объект равен null. Например, пусть у нас есть следующая система классов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class Person  {      public Company? Company { get; set; }   // место работы  }  class Company  {      public string? WebSite { get; set; }    // веб-сайт компании  } |

Объект Person представляет человека. Его свойство Company представляет компанию, где человек работает. Но человек может не работать, поэтому свойство Company имеет тип Company?, то есть может иметь значение null.

Класс Company в свою очередь содержит свойство WebSite, которое представляет веб-сайт компании. Но у компании может и не быть собственного веб-сайта. Поэтому это свойство имеет тип string?, то есть также допускает значение null.

Допустим, нам надо вывести на консоль заглавными буквами веб-сайт компании, где работает человек (если он, конечно, работает и если у компании, где он работает, есть сайт). На первый взгляд мы можем написать следующую конструкцию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | void PrintWebSite(Person? person)  {      if (person != null)      {          if(person.Company != null)          {              if(person.Company.WebSite != null)              {                  Console.WriteLine(person.Company.WebSite.ToUpper());              }          }      }  }  class Person  {      public Company? Company { get; set; }   // место работы  }  class Company  {      public string? WebSite { get; set; }    // веб-сайт компании  } |

В методе PrintWebSite() принимаем объект Person? и, чтобы избежать исключения NullReferenceException, последовательно проверяем все используемые значения на null, чтобы в конце с помощью метода ToUpper() вывести заглавными буквами название сайта.

Хоть это и рабочий способ, но для простого вывода строки получается многоэтажная конструкция, но на самом деле ее можно сократить:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | void PrintWebSite(Person? person)  {      if (person != null && person.Company != null && person.Company.WebSite != null)      {          Console.WriteLine(person.Company.WebSite.ToUpper());      }  } |

Конструкция намного проще, но все равно получается довольно большой. И чтобы ее упростить, в C# есть оператор условного null (Null-Conditional Operator) - оператор **?.**:

|  |  |
| --- | --- |
| 1 | объект?.компонент |

Если объект не равен **null**, то происходит обращение к компоненту объекта - полю, свойству, методу. Если объект представляет значение null, обращение к компаненту метода не происходит.

Применим данный оператор, изменив предыдущий пример:

|  |  |
| --- | --- |
| 1  2  3  4 | void PrintWebSite(Person? person)  {      Console.WriteLine(person?.Company?.WebSite?.ToUpper());  } |

Таким образом, если person не равен null, то происходит обращение к его свойству Company. Если свойство Company не равно null, то идет обрашение к свойству WebSite объекта Company. Если свойство WebSite не равно null, то идет обращение к методу ToUpper().

## Псевдонимы типов и статический импорт

### **Псевдонимы**

Для различных классов и структур мы можем использовать псевдонимы. Затем в программе вместо названия типа используется его псевдоним. Например, для вывода строки на экран применяется метод Console.WriteLine(). Но теперь зададим для класса **Console** псевдоним:

|  |  |
| --- | --- |
| 1  2  3  4 | using printer = System.Console;    printer.WriteLine("Laudate omnes gentes laudate");  printer.WriteLine("Magnificat in secula"); |

С помощью выражения using printer = System.Console указываем, что псевдонимом для класса System.Console будет имя printer. Это выражение не имеет ничего общего с подключением пространств имен в начале файла, хотя и использует оператор using. При этом используется полное имя класса с учетом пространства имен, в котором класс определен. И далее, чтобы вывести строку, применяется выражение printer.WriteLine("Laudate omnes gentes laudate").

Данная возможность может применяться, когда имена классов и структур довольно длинные, и мы хотим их сократить для большей лаконичности кода, либо когда мы хотим иметь более выразительные на наш субъективный взгляд имена типов.

И еще пример. Определим класс и для него псевдоним:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | using printer = System.Console;  using User = Person;    User tom = new User("Tom");  printer.WriteLine(tom.Name);    // Tom    class Person  {      public string Name { get; set; }      public Person(string name) => Name = name;  } |

Класс называется Person, но в программе для него используется псевдоним User.

### **Статический импорт**

Также в C# имеется возможность импорта статической функциональности классов. Например, импортируем возможности класса Console:

|  |  |
| --- | --- |
| 1  2  3 | using static System.Console;    WriteLine("Hello from C# 8.0"); |

Выражение **using static** подключает в программу все статические методы и свойства, а также константы. И после этого мы можем не указывать название класса при вызове метода.

Подобным образом можно определять свои классы и импортировать их:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | using static System.Console;  using static Operation;    WriteLine(Sum(5, 4));       // 9  WriteLine(Subtract(5, 4));  // 1  WriteLine(Multiply(5, 4)); // 20    static class Operation  {      public static int Sum(int a, int b) => a + b;      public static int Subtract(int a, int b) => a - b;      public static int Multiply(int a, int b) => a \* b;  } |

**Глава 4. Объектно-ориентированное программирование**