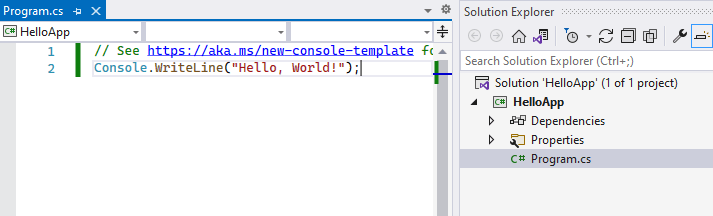
Основы программирования на C#

## 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:



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#.

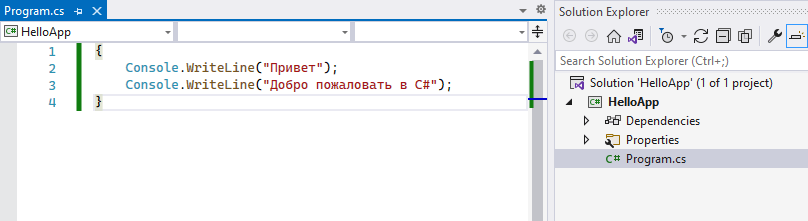
### **Инструкции**

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:

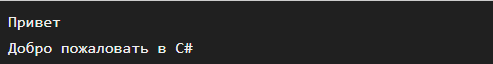


Данная строка представляет вызов метода Console.WriteLine, который выводит на консоль строку. В данном случае вызов метода является инструкцией и поэтому завершается точкой с запятой.

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:

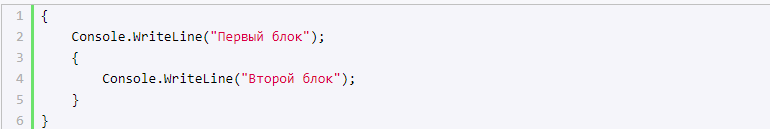


Здесь блок кода содержит две инструкции. И при выполении этого кода, консоль выведет две строки



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

Одни блоки кода могут содержать другие блоки:



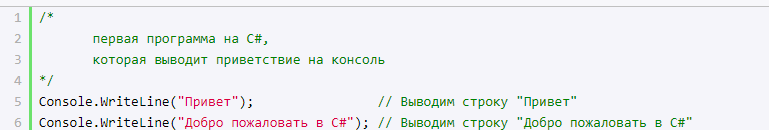
**Регистрозависимость**

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.

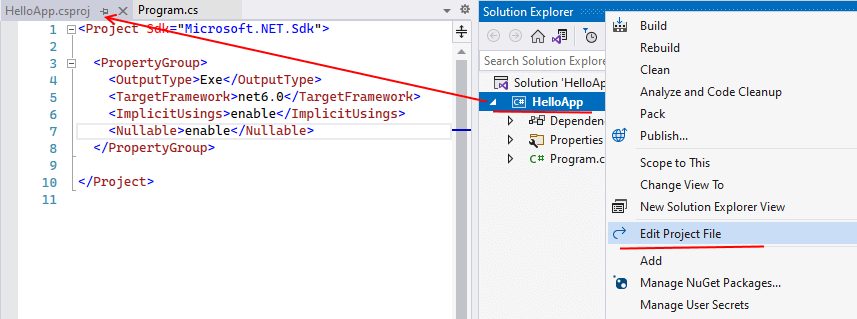
### **Комментарии**

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.

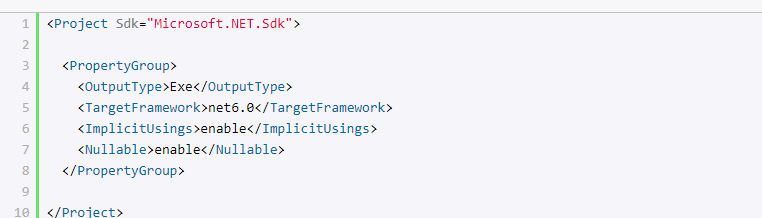
есть два типа комментариев: однострочный и многострочный. Однострочный комментарий размещается на одной строке после двойного слеша //. А многострочный комментарий заключается между символами /\* текст комментария \*/. Он может размещаться на нескольких строках. Например: **Файл проекта**



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: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: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.

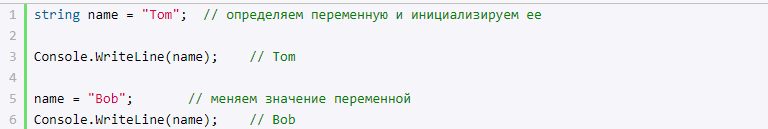


При этом следует учитывать, что C# является регистрозависимым языком, поэтому следующие два определения переменных будут представлять две разные переменные:После определения переменной можно присвоить некоторое значение:Так как переменная name представляет тип string, то есть строку, то мы можем присвоить 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: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:



Консольный вывод программы:



### **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:



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 изменить 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.

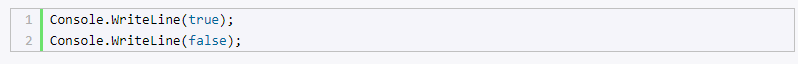
#### Вопрос для самопроверки: 3/3

## **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.

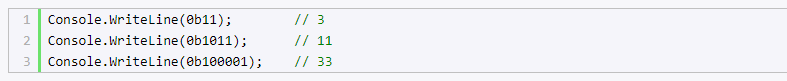
### **Логические литералы**

Есть две логических константы - **true** (истина) и **false** (ложь):**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:Числа в двоичной форме предваряются символами 0b, после которых идет набор из нулей и единиц: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:**Вещественные литералы**

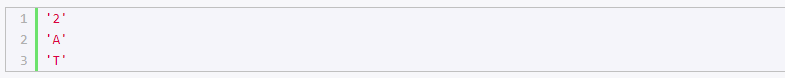


Вещественные литералы представляют дробные числа. Этот тип литералов имеет две формы. Первая форма - вещественные числа с фиксированной запятой, при которой дробную часть отделяется от целой части точкой. Например:Также вещественные литералы могут определяться в экспоненциальной форме MEp, где M — мантисса, E - экспонента, которая фактически означает "\*10^" (умножить на десять в степени), а p — порядок. Например:**Символьные литералы**



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

Символьные литералы бывают нескольких видов. Прежде всего это обычные символы:Также мы можем передать их вывести на консоль с помощью 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.

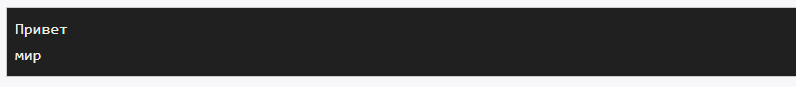
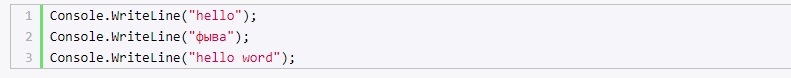
Также символы могут определяться в виде шестнадцатеричных кодов, также заключенный в одинарные кавычки.

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":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':**Строковые литералы**



Строковые литералы представляют строки. Строки заключаются в двойные кавычки:Если внутри строки необходимо вывести двойную кавычку, то такая внутренняя кавычка предваряется обратным слешем:Также в строках можно использовать управляющие последовательности. Например, последовательность '\n' осуществляет перевод на новую строку:При выводе на консоль слово "мир" будет перенесено на новую строку:



### **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  
**byte**: хранит целое число от 0 до 255 и занимает 1 байт. Представлен системным типом System.Byte  
**sbyte**: хранит целое число от -128 до 127 и занимает 1 байт. Представлен системным типом System.SByte  
**short**: хранит целое число от -32768 до 32767 и занимает 2 байта. Представлен системным типом System.Int16**ushort**: хранит целое число от 0 до 65535 и занимает 2 байта. Представлен системным типом System.UInt16  
**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:  
**uint**: хранит целое число от 0 до 4294967295 и занимает 4 байта. Представлен системным типом System.UInt32  
**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  
**ulong**: хранит целое число от 0 до 18 446 744 073 709 551 615 и занимает 8 байт. Представлен системным типом System.UInt64  
**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:  
**string**: хранит набор символов Unicode. Представлен системным типом System.String. Этому типу соответствуют строковые литералы.  
**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.

### **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.

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**:

**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:

**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:

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:

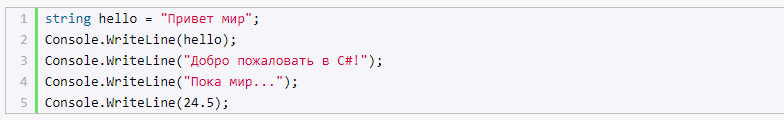
Во-вторых, мы не можем указать в качестве значения неявно типизируемой переменной null:

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:



Консольный вывод:



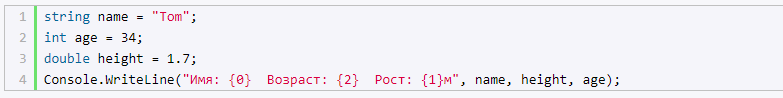
Нередко возникает необходимость вывести на консоль в одной строке значения сразу нескольких переменных. В этом случае мы можем использовать прием, который называется интерполяцией:Для встраивания отдельных значений в выводимую на консоль строку используются фигурные скобки, в которые заключается встраиваемое 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:



Есть другой способ вывода на консоль сразу нескольких значений: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 вставляться вместо первого плейсхолдера - {0} (нумерация начинается с нуля), height - вместо {1}, а age - вместо {2}. Поэтому в итоге пи выводе на консоль строкабудет заменена на**Console.Write**

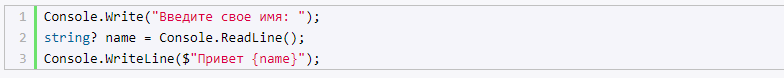


Кроме Console.WriteLine() можно также использовать метод **Console.Write()**, он работает точно так же за тем исключением, что не добавляет переход на следующую строку, то есть последующий консольный вывод будет выводиться на той же строке.



**Консольный ввод**

Кроме вывода информации на консоль мы можем получать информацию с консоли. Для этого предназначен метод **Console.ReadLine()**. Он позволяет получить введенную строку.В данном случае все, что вводит пользователь, с помощью метода Console.ReadLine() передается в переменную name.



Пример работы программы:

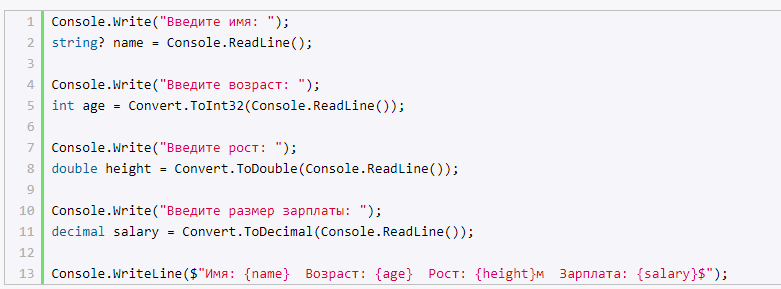


Особенностью метода Console.ReadLine() является то, что он может считать информацию с консоли только в виде строки. Кроме того, возможная ситуация, когда для метода Console.ReadLine не окажется доступных для считывания строк, то есть когда ему нечего считывать, он возвращаает значение **null**, то есть, 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.

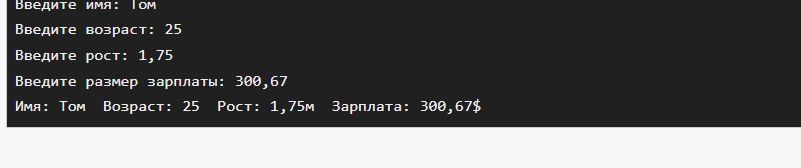
Однако, может возникнуть вопрос, как нам быть, если, допустим, мы хотим ввести возраст в переменную типа int или другую информацию в переменные типа double или decimal? По умолчанию платформа .NET предоставляет ряд методов, которые позволяют преобразовать различные значения к типам int, double и т.д.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

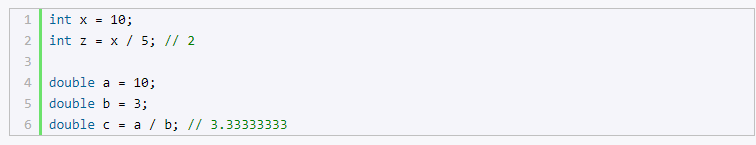


## **Арифметические операции языка C#**

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:

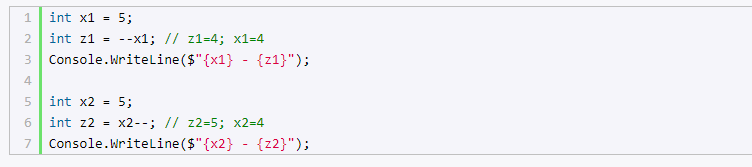
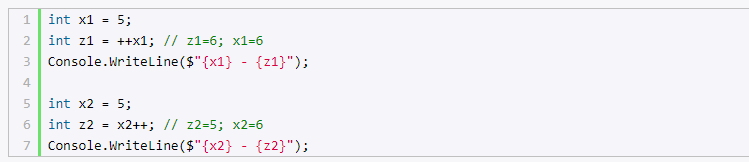
**+**Операция сложения двух чисел:  
**-**Операция вычитания двух чисел:  
**\***Операция умножения двух чисел:  
**/**операция деления двух чисел:  
При делении стоит учитывать, что если оба операнда представляют целые числа, то результат также будет округляться до целого числа:  
Несмотря на то, что 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:  
**%**Операция получение остатка от целочисленного деления двух чисел: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.

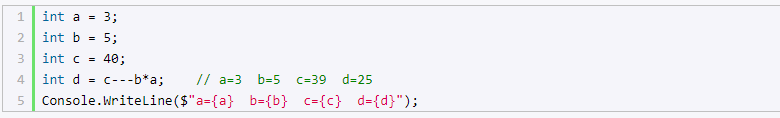
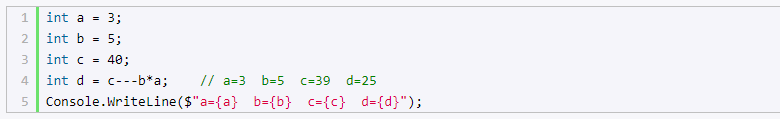
И также существует постфиксный инкремент: x++ - сначала значение переменной x возвращается в качестве результата операции, а затем к нему прибавляется 1.  
**--**Операция декремента или уменьшения значения на единицу. Также существует префиксная форма декремента (--x) и постфиксная (x--).При выполнении 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: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:Но с помощью скобок мы могли бы изменить порядок операций, например, следующим образом:**Ассоциативность операторов**



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

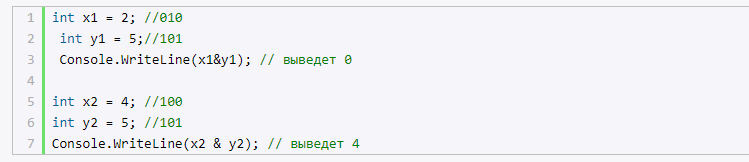
Все арифметические операторы являются левоассоциативными, то есть выполняются слева направо. Поэтому выражение 10 / 5 \* 2 необходимо трактовать как (10 / 5) \* 2, то есть результатом будет 4.

## **Поразрядные операции**

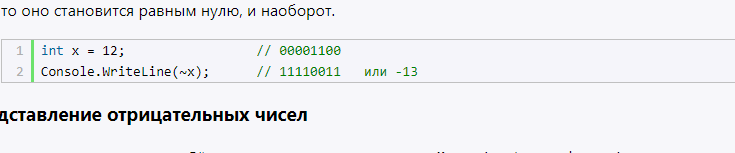
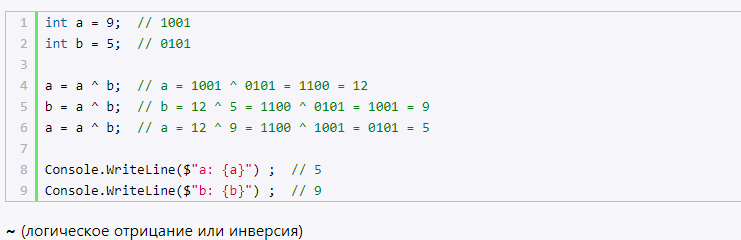
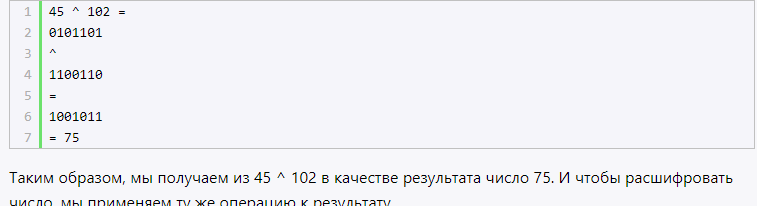
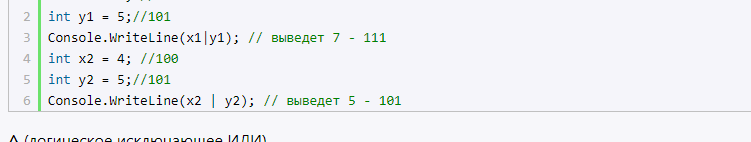
Особый класс операций представляют поразрядные операции. Они выполняются над отдельными разрядами числа. В этом плане числа рассматриваются в двоичном представлении, например, 2 в двоичном представлении 10 и имеет два разряда, число 7 - 111 и имеет три разряда.

### **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:  
  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.



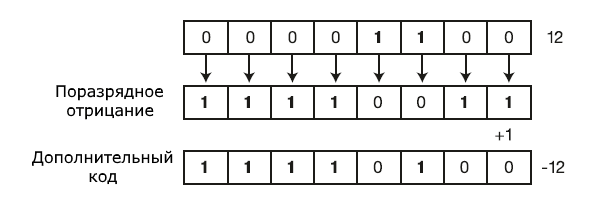
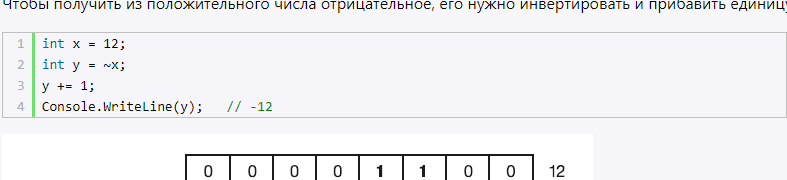
* **|** (логическое сложение)  
  Похоже на логическое умножение, операция также производится по двоичным разрядам, но теперь возвращается единица, если хотя бы у одного числа в данном разряде имеется единица. Например:  
  **^** (логическое исключающее ИЛИ)  
  Также эту операцию называют XOR, нередко ее применяют для простого шифрования:  
  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 45 ^ 102 в качестве результата число 75. И чтобы расшифровать число, мы применяем ту же операцию к результату.  
  Подобным образом можно обменять два положительных числа без использования дополнительной переменной:  
  **~** (логическое отрицание или инверсия)  
  Еще одна поразрядная операция, которая инвертирует все разряды: если значение разряда равно 1, то оно становится равным нулю, и наоборот.**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:

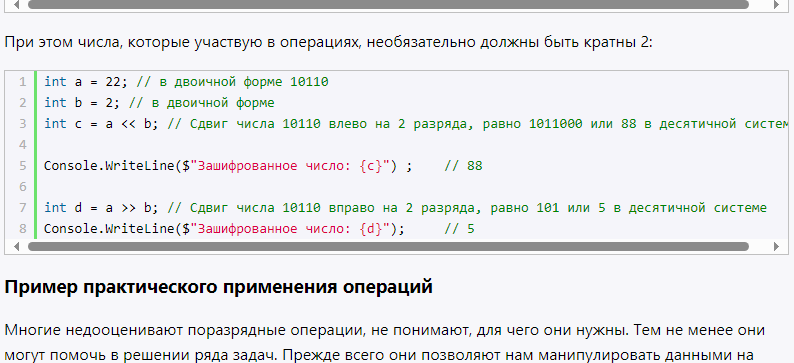
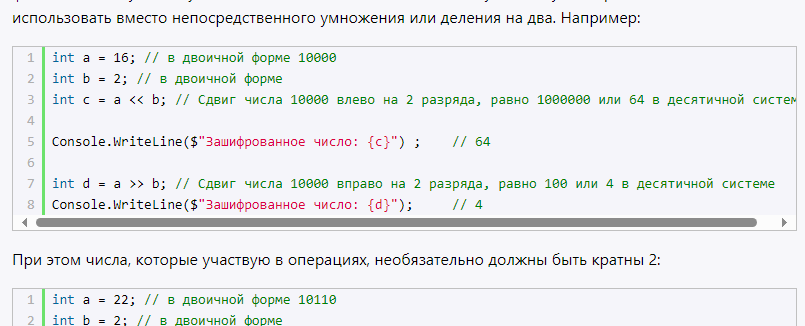


### **Операции сдвига**

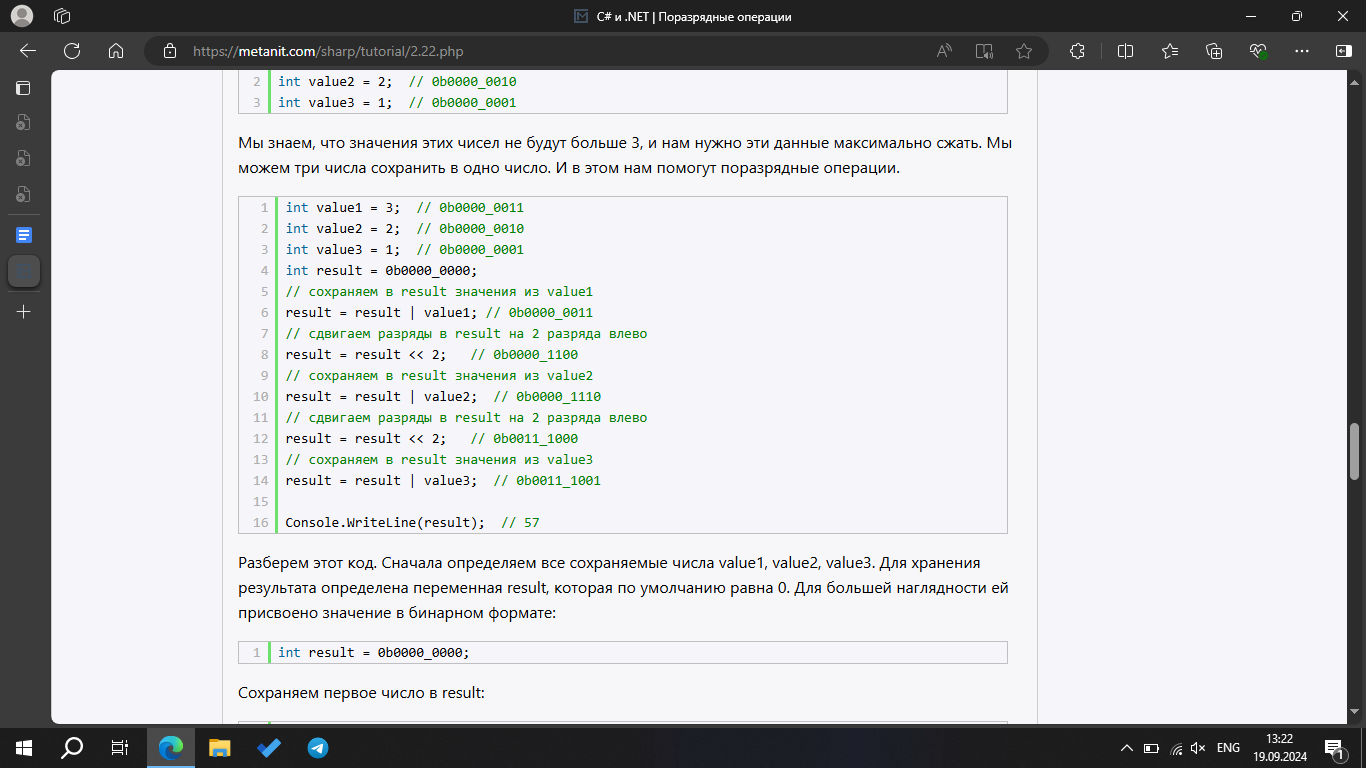
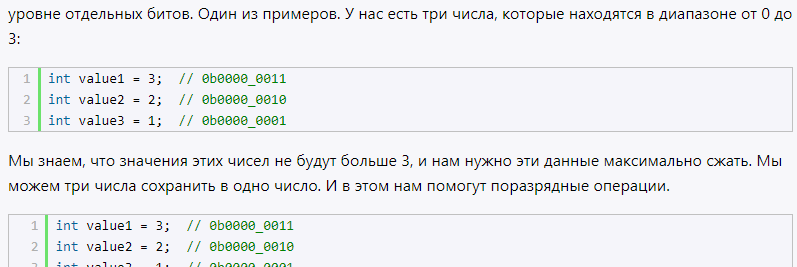
Операции сдвига также производятся над разрядами чисел. Сдвиг может происходить вправо и влево.

* 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.

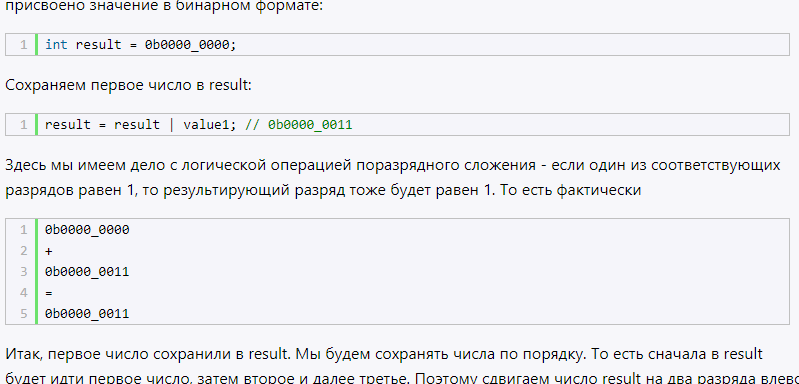
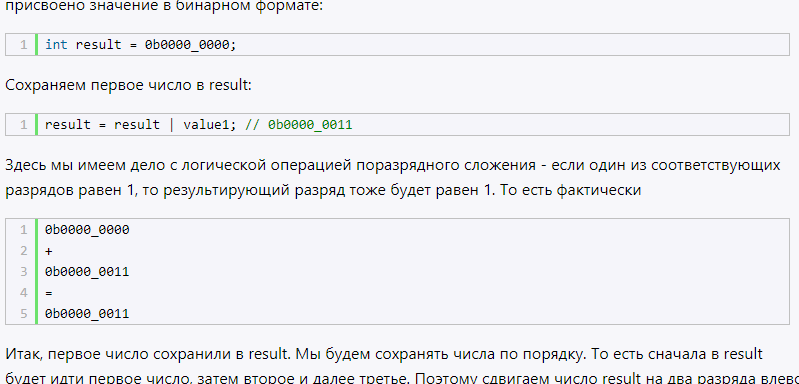
Таким образом, если исходное число, которое надо сдвинуть в ту или другую строну, делится на два, то фактически получается умножение или деление на два. Поэтому подобную операцию можно использовать вместо непосредственного умножения или деления на два. Например:При этом числа, которые участвую в операциях, необязательно должны быть кратны 2:**Пример практического применения операций**



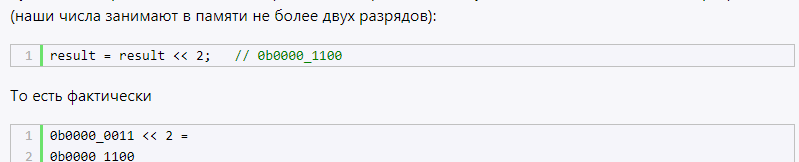
Многие недооценивают поразрядные операции, не понимают, для чего они нужны. Тем не менее они могут помочь в решении ряда задач. Прежде всего они позволяют нам манипулировать данными на уровне отдельных битов. Один из примеров. У нас есть три числа, которые находятся в диапазоне от 0 до 3:Мы знаем, что значения этих чисел не будут больше 3, и нам нужно эти данные максимально сжать. Мы можем три числа сохранить в одно число. И в этом нам помогут поразрядные операции.



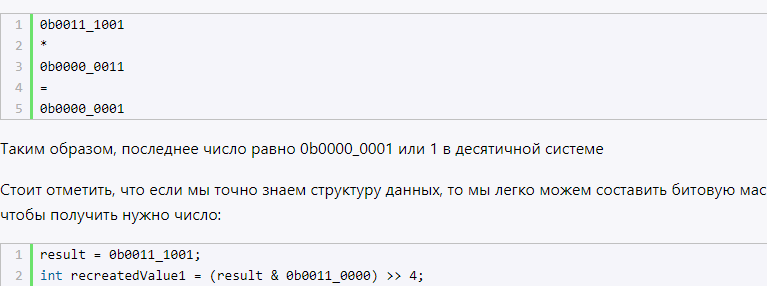
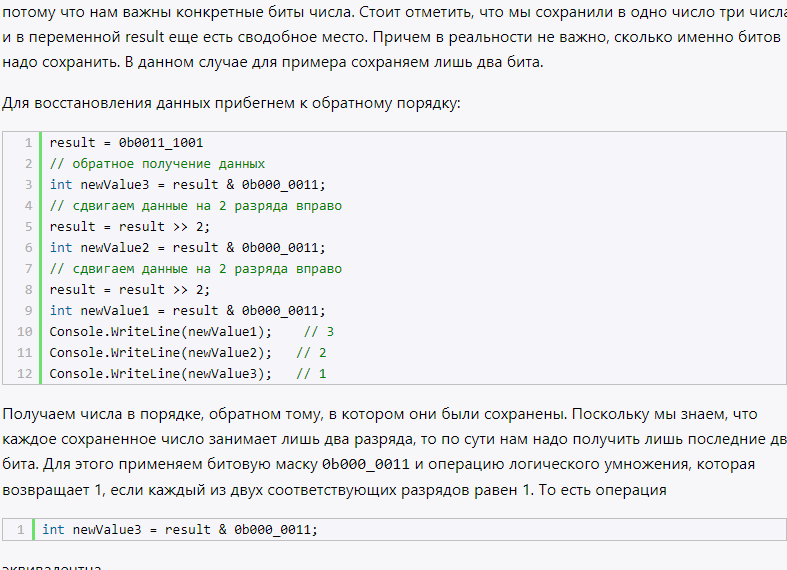
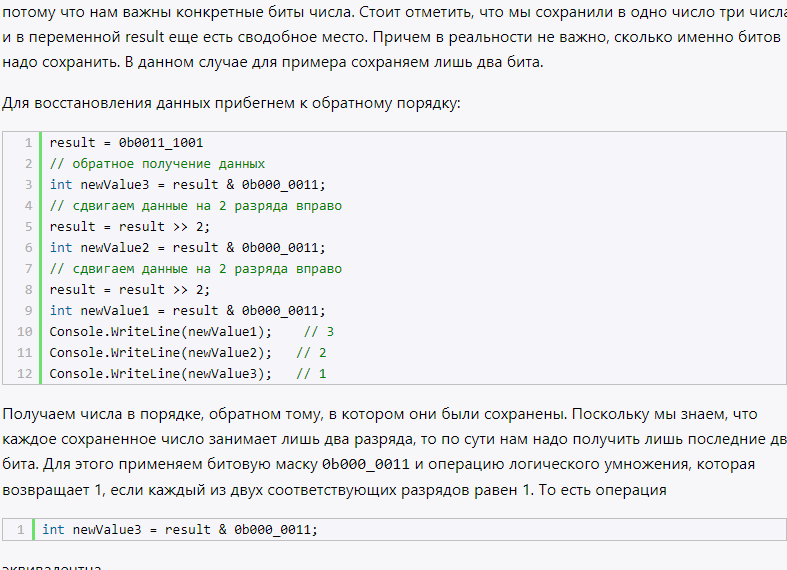
Разберем этот код. Сначала определяем все сохраняемые числа value1, value2, value3. Для хранения результата определена переменная result, которая по умолчанию равна 0. Для большей наглядности ей присвоено значение в бинарном формате:Сохраняем первое число в result:Здесь мы имеем дело с логической операцией поразрядного сложения - если один из соответствующих разрядов равен 1, то результирующий разряд тоже будет равен 1. То есть фактически



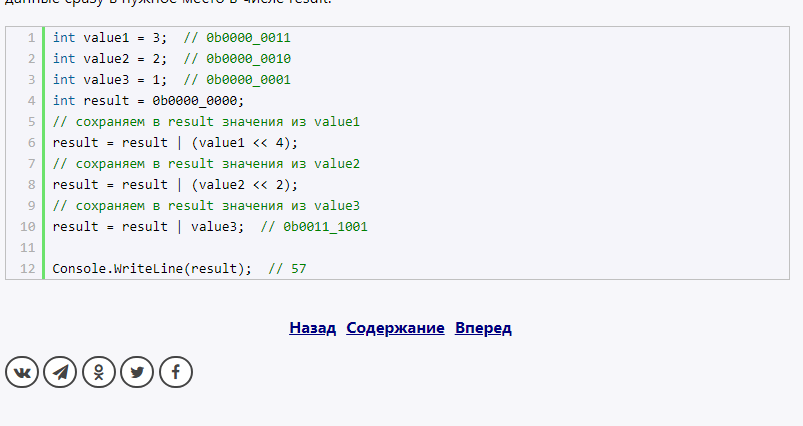
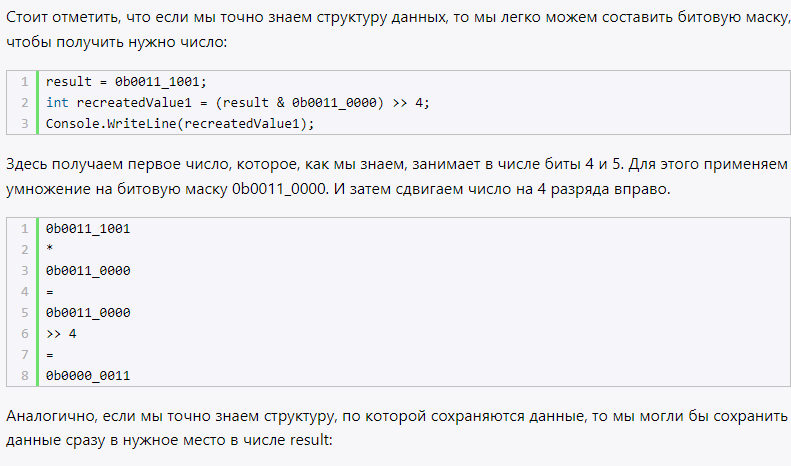
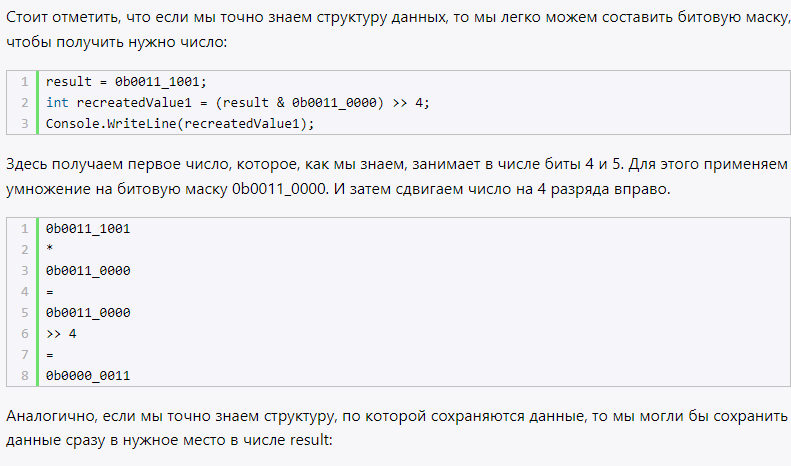
Итак, первое число сохранили в result. Мы будем сохранять числа по порядку. То есть сначала в result будет идти первое число, затем второе и далее третье. Поэтому сдвигаем число result на два разряда влево (наши числа занимают в памяти не более двух разрядов):То есть фактическиДалее повторяем логическую операцию сложения, сохраняем второе число:что эквивалентноДалее повторяем сдвиг на два разряда влево и сохраняем третье число. В итоге мы получим в двоичном представлении число 0b0011\_1001. В десятично системе это число равно 57. Но это не имеет значения, потому что нам важны конкретные биты числа. Стоит отметить, что мы сохранили в одно число три числа, и в переменной result еще есть сводобное место. Причем в реальности не важно, сколько именно битов надо сохранить. В данном случае для примера сохраняем лишь два бита.



Для восстановления данных прибегнем к обратному порядку:Получаем числа в порядке, обратном тому, в котором они были сохранены. Поскольку мы знаем, что каждое сохраненное число занимает лишь два разряда, то по сути нам надо получить лишь последние два бита. Для этого применяем битовую маску 0b000\_0011 и операцию логического умножения, которая возвращает 1, если каждый из двух соответствующих разрядов равен 1. То есть операцияэквивалентнаТаким образом, последнее число равно 0b0000\_0001 или 1 в десятичной системе



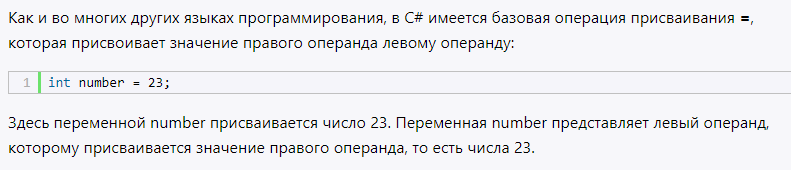
Стоит отметить, что если мы точно знаем структуру данных, то мы легко можем составить битовую маску, чтобы получить нужно число:Здесь получаем первое число, которое, как мы знаем, занимает в числе биты 4 и 5. Для этого применяем умножение на битовую маску 0b0011\_0000. И затем сдвигаем число на 4 разряда вправо.Аналогично, если мы точно знаем структуру, по которой сохраняются данные, то мы могли бы сохранить данные сразу в нужное место в числе result:



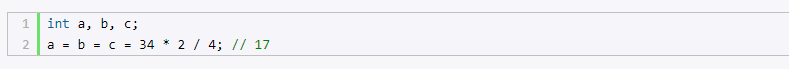
## **Операции присваивания**

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



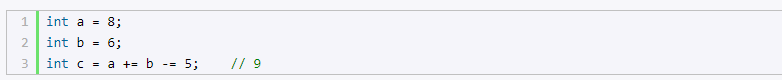
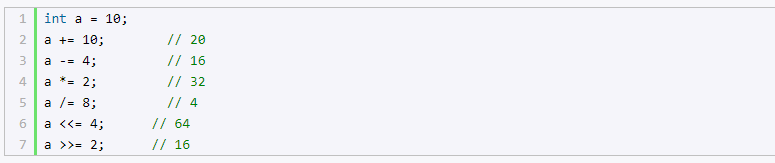
Также можно выполнять множественно присвоение сразу нескольких переменным одновременно:Стоит отметить, что операции присвоения имеют низкий приоритет. И вначале будет вычисляться значение правого операнда и только потом будет идти присвоение этого значения левому операнду. Например:Сначала будет вычисляться 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:Операции присвоения являются правоассоциативными, то есть выполняются справа налево. Например: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:



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

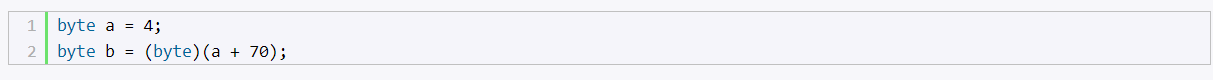
Но теперь попробуем применить сложение к двум объектам типа **byte**:Здесь поменялся только тип переменной, которая получает результат сложения - с int на byte. Однако при попытке скомпилировать программу мы получим ошибку на этапе компиляции. И если мы работаем в Visual Studio, среда подчеркнет вторую строку красной волнистой линией, указывая, что в ней ошибка.



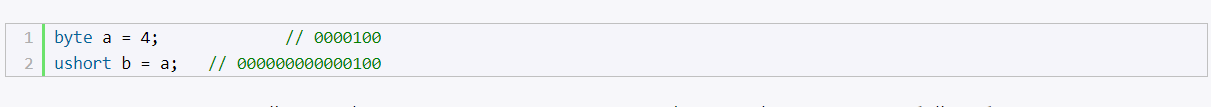
При операциях мы должны учитывать диапазон значений, которые может хранить тот или иной тип. Но в данном случае число 74, которое мы ожидаем получить, вполне укладывается в диапазон значений типа byte, тем не менее мы получаем ошибку.

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.

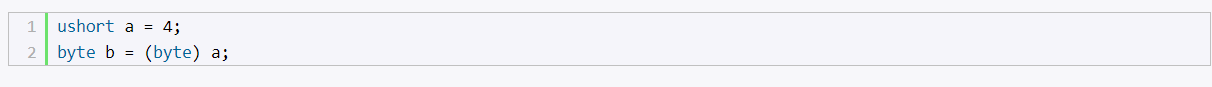
И чтобы выйти из этой ситуации, необходимо применить операцию преобразования типов. **Операция преобразования** типов предполагает указание в скобках того типа, к которому надо преобразовать значение:Так, изменим предыдущий пример, применив операцию преобразования типов:**Сужающие и расширяющие преобразования**



Преобразования могут быть сужающие (narrowing) и расширяющие (widening). Расширяющие преобразования расширяют размер объекта в памяти. Например: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:Значение типа ushort занимает 2 байта (16 бит). И при присвоении переменной b значение переменной a расширяется до 2 байтТо 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: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).

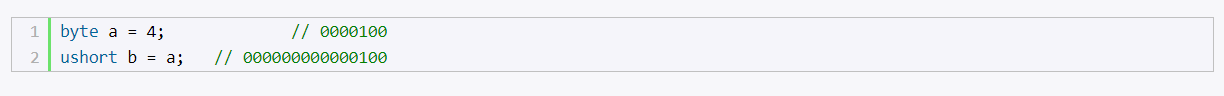


### **Явные и неявные преобразования**

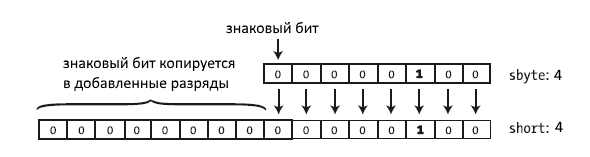
#### **Неявные преобразования**

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.

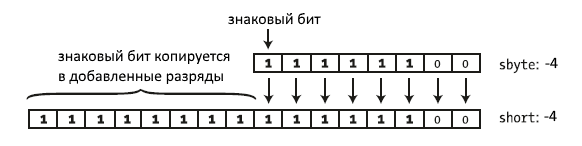
Если производится преобразование от безнакового типа меньшей разрядности к безнаковому типу большой разрядности, то добавляются дополнительные биты, которые имеют значени 0. Это называется **дополнение нулями** или zero extension.Если производится преобразование к знаковому типу, то битовое представление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

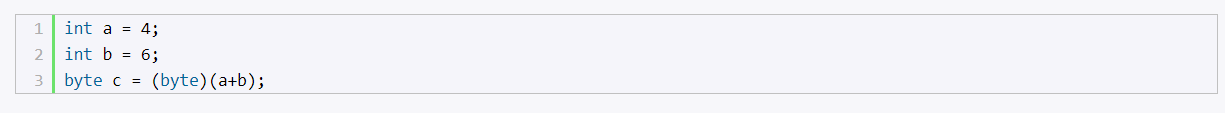


Преобразование отрицательного числа:



#### **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: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**

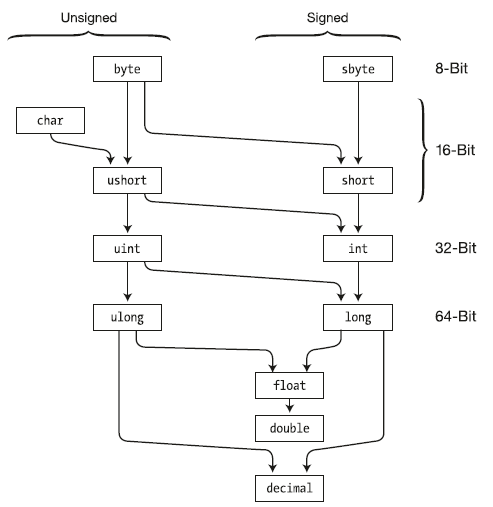
**int** -> **double**

**short** -> **float** -> **double**

**char** -> **int**

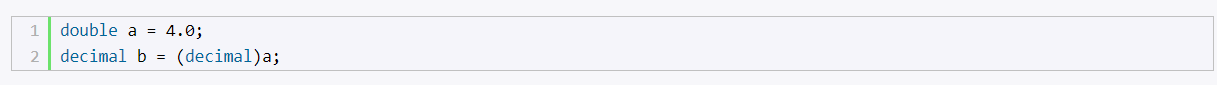
Все безопасные автоматические преобразования можно описать следующей таблицей:

|  |  |
| --- | --- |
| 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 |

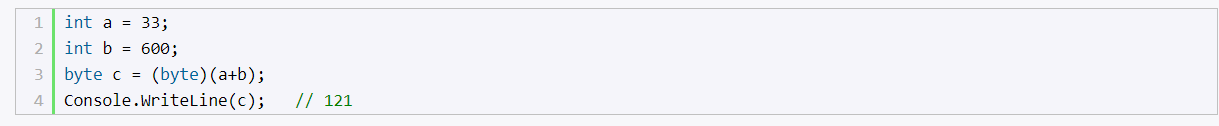


В остальных случаях следует использовать явные преобразования типов.

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:**Потеря точности данных**



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



## **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.

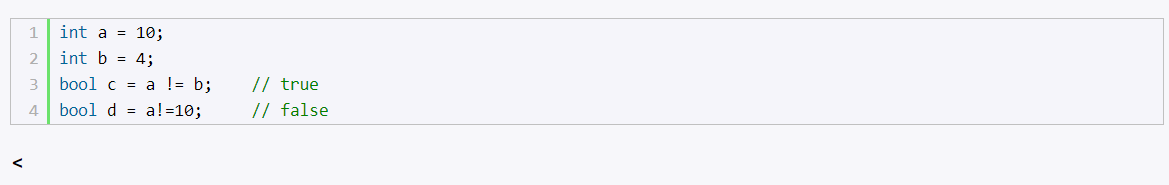
### **Операции сравнения**

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

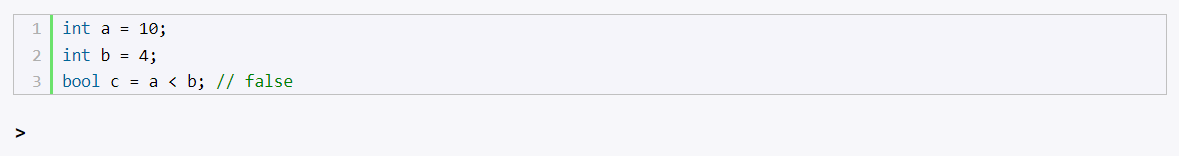
* **==**Сравнивает два операнда на равенство. Если они равны, то операция возвращает **true**, если не равны, то возвращается **false**:

|  |  |
| --- | --- |
| 1  2  3 |  |

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



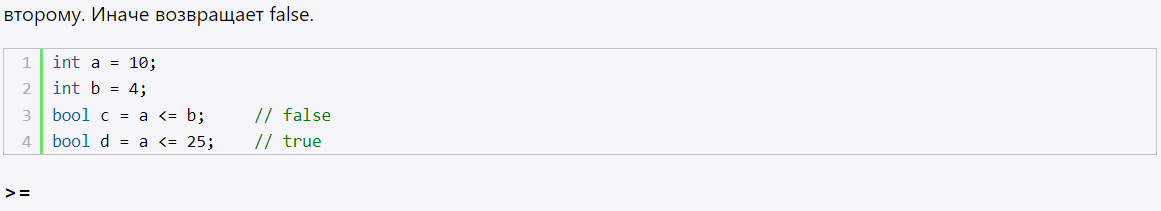
* **<**Операция "меньше чем". Возвращает true, если первый операнд меньше второго, и false, если первый операнд больше второго:



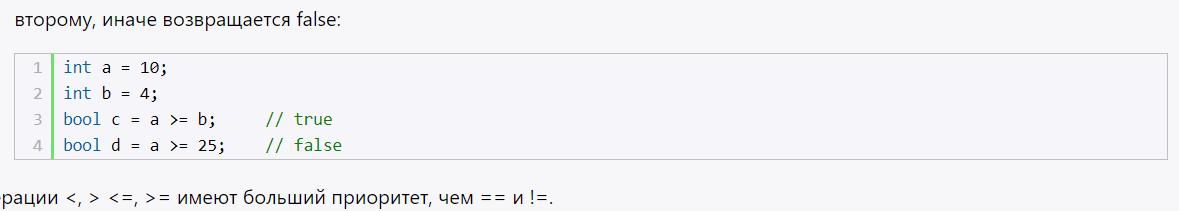
* **>**Операция "больше чем". Сравнивает два операнда и возвращает true, если первый операнд больше второго, иначе возвращает false:

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

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



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

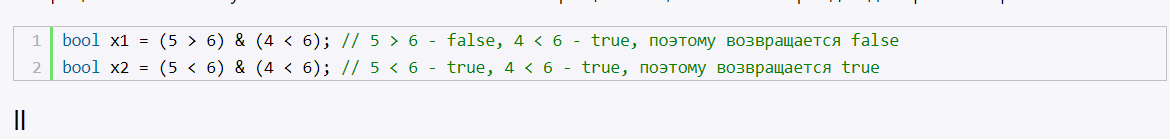
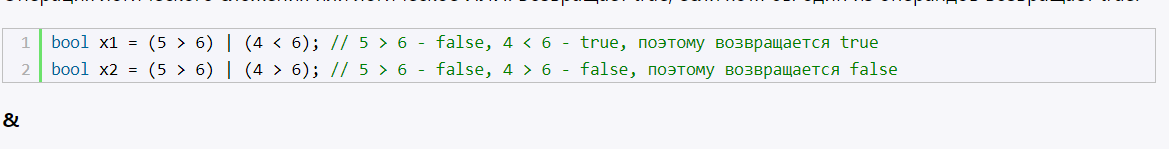


Операции <, > <=, >= имеют больший приоритет, чем == и !=.

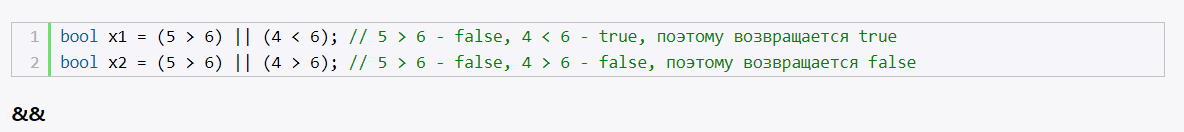
### **Логические операции**

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.

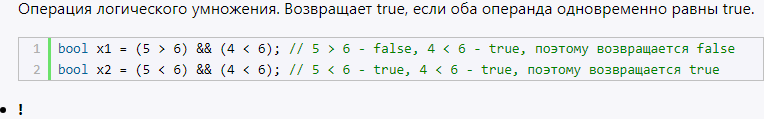
* **|**Операция логического сложения или логическое ИЛИ. Возвращает true, если хотя бы один из операндов возвращает true.**&**Операция логического умножения или логическое И. Возвращает true, если оба операнда одновременно равны true.



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



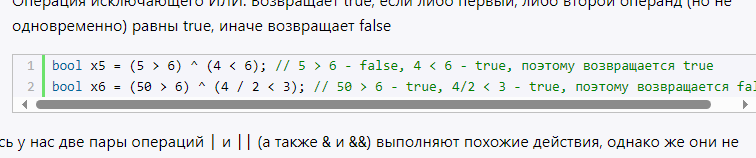
* **&&**Операция логического умножения. Возвращает true, если оба операнда одновременно равны true.



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

|  |  |
| --- | --- |
| 1  2 |  |

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

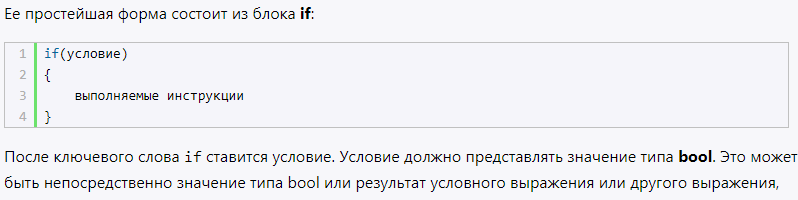
Поэтому операции || и && более удобны в вычислениях, так как позволяют сократить время на вычисление значения выражения, и тем самым повышают производительность. А операции | и & больше подходят для выполнения поразрядных операций над числами.

## **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

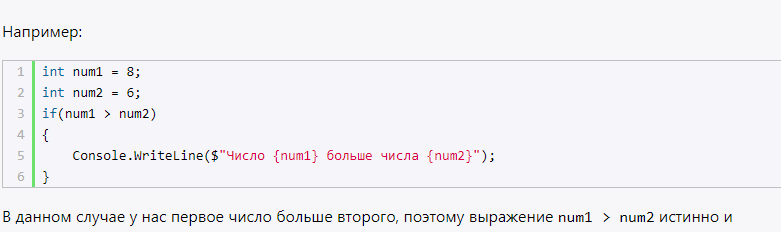
Конструкция if/else проверяет истинность некоторого условия и в зависимости от результатов проверки выполняет определенный код.

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

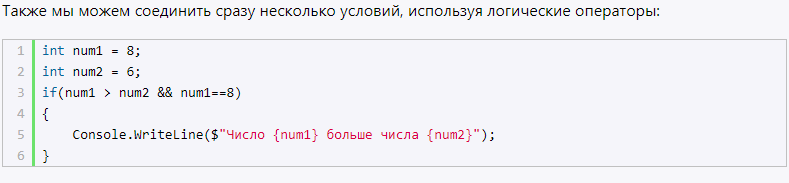
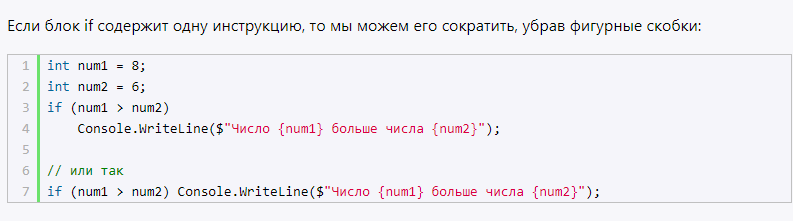


После ключевого слова if ставится условие. Условие должно представлять значение типа **bool**. Это может быть непосредственно значение типа bool или результат условного выражения или другого выражения, которое возвращает значение типа bool. И если это условие истинно (равно true), то срабатывает код,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

Например: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}");

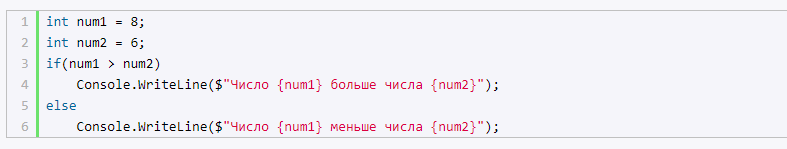
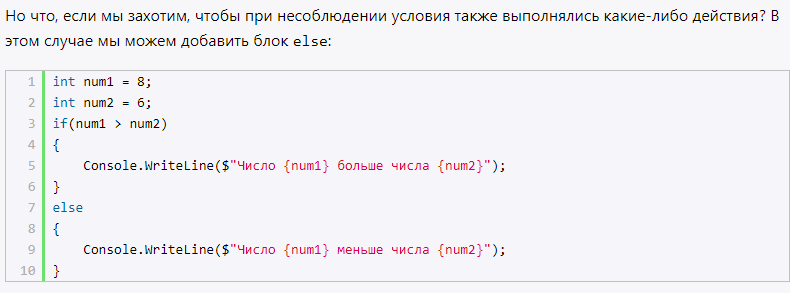


Если блок if содержит одну инструкцию, то мы можем его сократить, убрав фигурные скобки:Также мы можем соединить сразу несколько условий, используя логические операторы:В данном случае блок if будет выполняться, если num1 > num2 равно true и num1==8 равно true.



#### **Выражение else**

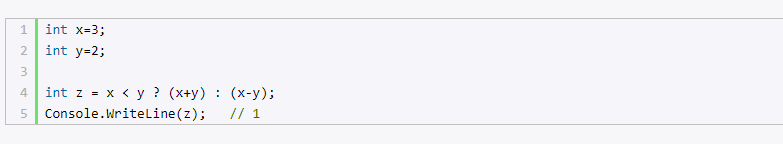
Но что, если мы захотим, чтобы при несоблюдении условия также выполнялись какие-либо действия? В этом случае мы можем добавить блок else:Блок else выполняется, если условие после if ложно, то есть равно false. Если блок else содержит толко одну инструкцию, то опять же мы можем его сократить, убрав фигурные скобки:**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:При необходимости можно добавить несколько выражений else if:**Тернарная операция**



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

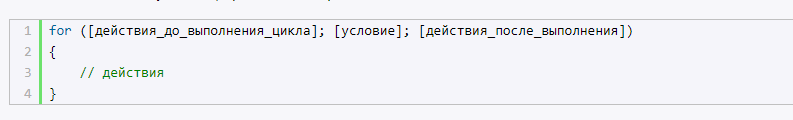
## **Циклы**

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: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:



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.

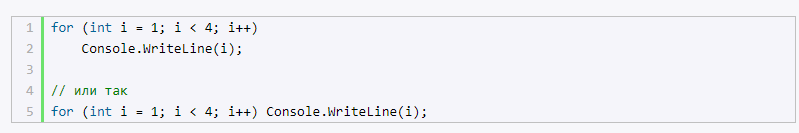
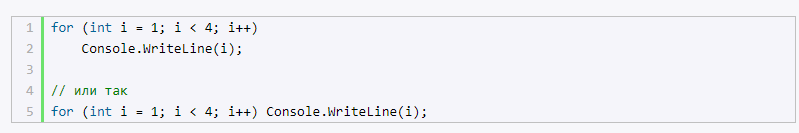
Весь процесс цикла можно представить следующим образом:

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

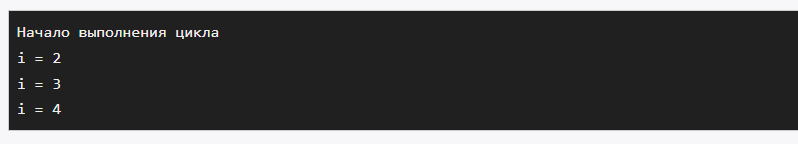
В итоге блок цикла сработает 3 раза, пока значение i не станет равным 4. И каждый раз это значение будет увеличиваться на 1. Однократное выполнение блока цикла называется **итерацией**. Таким образом, здесь цикл выполнит три итерации. Результат работы программы:



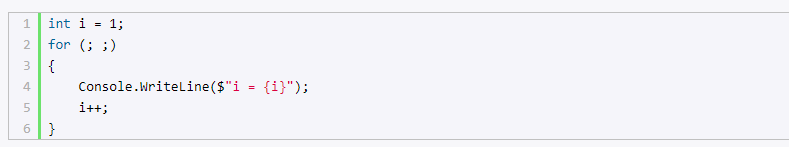
Если блок цикла for содержит одну инструкцию, то мы можем его сократить, убрав фигурные свобки:При этом необязательно именно в первой части цикла объявлять переменную, а в третий части изменять ее значение - это могут быть любые действия. Например:Здесь опять же цикл срабатывает, пока переменная i меньше 4, только приращение переменной i происходит в блоке цикла. Консольный вывод данной программы:



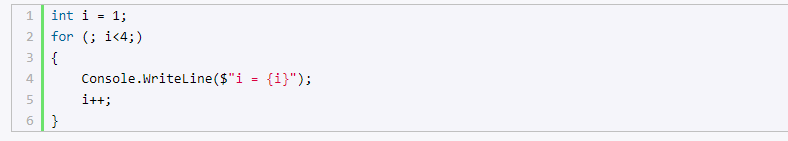
Начало выполнения цикла



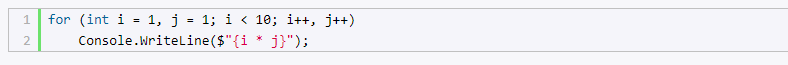
Нам необязательно указывать все условия при объявлении цикла. Например, мы можем написать так:Формально определение цикла осталось тем же, только теперь блоки в определении у нас пустые: for (; ;). У нас нет инициализированной переменной, нет условия, поэтому цикл будет работать вечно - бесконечный цикл.



Мы также можем опустить ряд блоков: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.

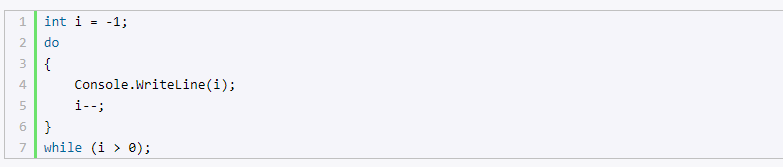
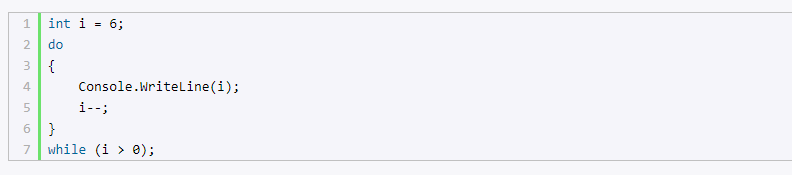
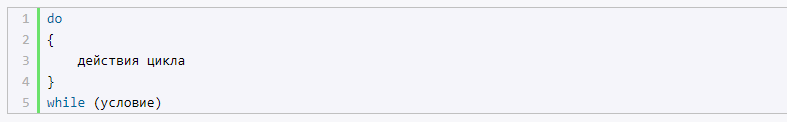


Также стоит отметить, что можно определять несколько переменных в объявлении цикла:Здесь в первой части объявления цикла определяются две переменных: i и j. Цикл выполняется, пока i не будет равна 10. После каждой итерации переменые i и j увеличиваются на единицу. Консольный вывод программы:



### **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.Например: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:Хотя у нас переменная i меньше 0, цикл все равно один раз выполнится.

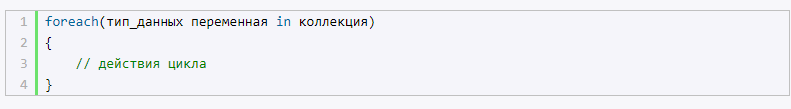


### **Цикл while**

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

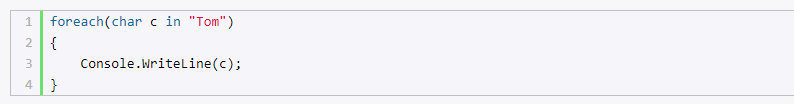


Цикл foreach предназначен для перебора набора или коллекции элементов. Его общее определение:После оператора foreach в скобках сначала идет определение переменной. Затем ключевое слово **in** и далее коллекция, элементы которой надо перебрать.

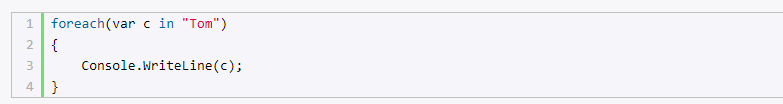


При выполнении цикл последовательно перебирает элементы коллекции и помещает их в переменную, и таким образом в блоке цикла мы можем выполнить с ними некоторые действия.

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



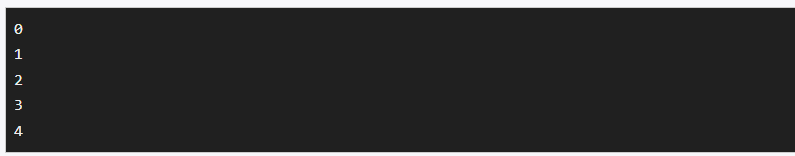
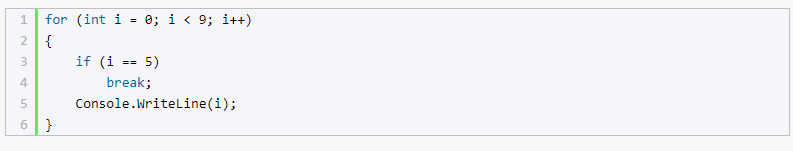
Стоит отметить, что переменная, которая определяется в объявлении цикла, должна по типу соответствовать типу элементов перебираемой коллекции. Так, элементы строки - значения типа char - символы. Поэтому переменная c имеет тип char. Однако в реальности не всегда бывает очевидно, какой тип представляют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



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



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

### **Вложенные циклы**

Одни циклы могут быть вложенными в другие. Например: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.

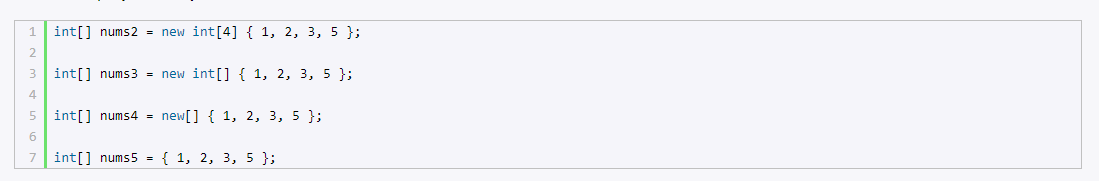


## **Массивы**

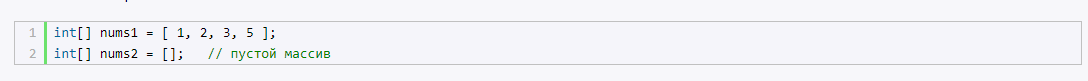
Массив представляет набор однотипных данных. Объявление массива похоже на объявление переменной за тем исключением, что после указания типа ставятся квадратные скобки:Например, определим массив целых чисел:После определения переменной массива мы можем присвоить ей определенное значение:Здесь вначале 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:Все перечисленные выше способы будут равноценны.

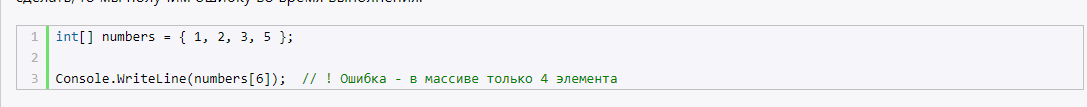
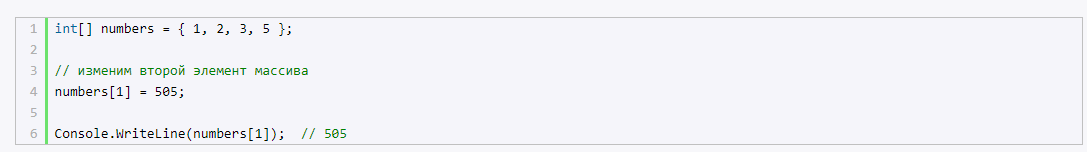
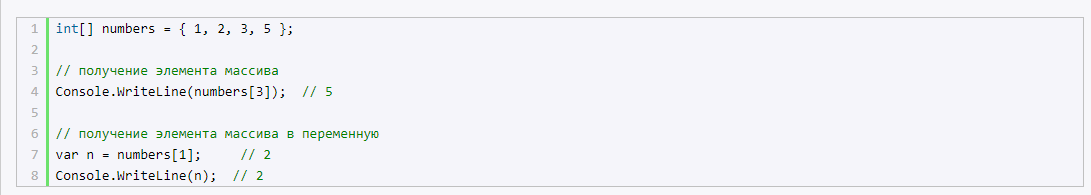


Подобным образом можно определять массивы и других типов, например, массив значений типа string:Начиная с версии **C# 12** для определения массивов можно использовать **выражения коллекций**, которые предполагают заключение элементов массива в квадратные скобки:**Индексы и получение элементов массива**

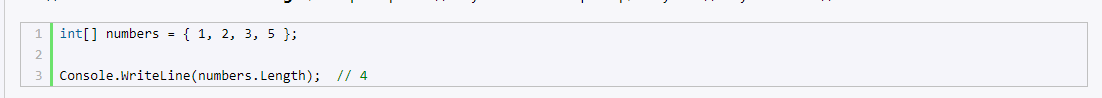


Для обращения к элементам массива используются **индексы**. Индекс представляет номер элемента в массиве, при этом нумерация начинается с нуля, поэтому индекс первого элемента будет равен 0, индекс четвертого элемента - 3.

Используя индексы, мы можем получить элементы массива:Также мы можем изменить элемент массива по индексу: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:**Свойство Length и длина массива**

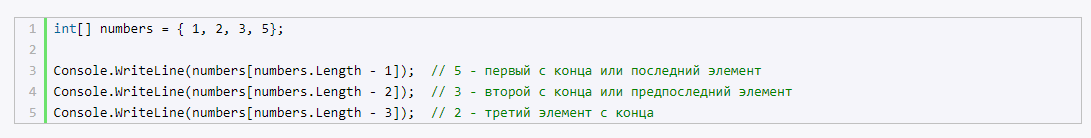


каждый массив имеет свойство **Length**, которое хранит длину массива. Например, получим длину выше созданного массива numbers:Для получения длины массива после названия массива через точку указывается свойство Length: numbers.Length.

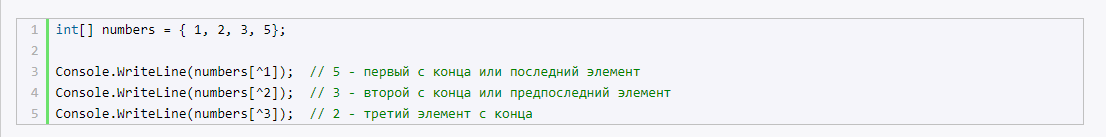


### **Получение элементов с конца массива**

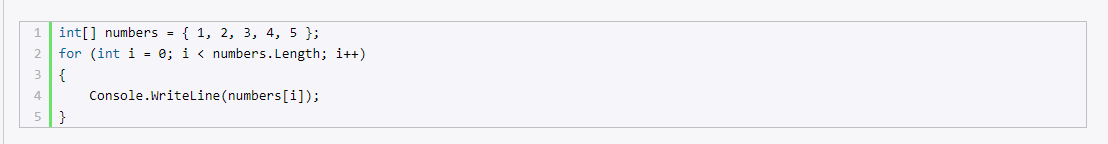
Благодаря наличию свойства Length, мы можем вычислить индекс последнего элемента массива - это длина массива - 1. Например, если длина массива - 4 (то есть массив имеет 4 элемента), то индекс последнего элемента будет равен 3. И, используя свойство Length, мы можем легко получить элементы с конца массива:Однако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



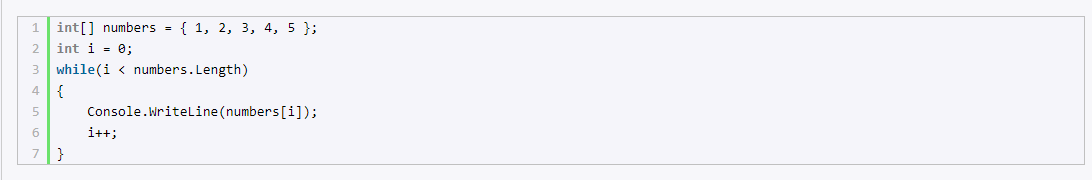
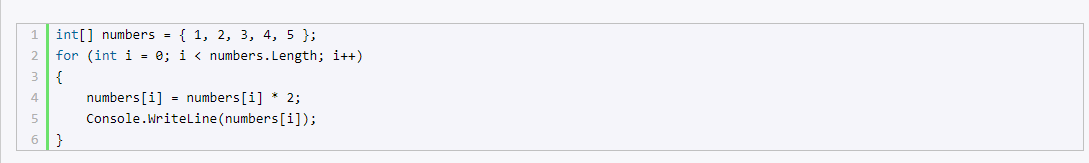
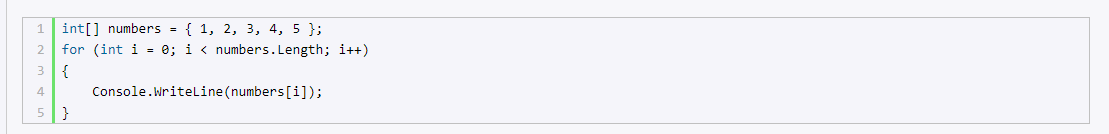
Перепишем предыдущий пример, применяя оператор **^**:**Перебор массивов**



Для перебора массивов мы можем использовать различные типы циклов. Например, цикл **foreach**:Здесь в качестве контейнера выступает массив данных типа int. Поэтому мы объявляем переменную с типом int

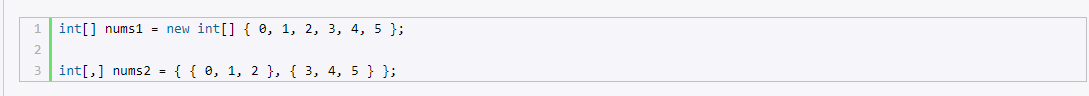


Подобные действия мы можем сделать и с помощью цикл for:В то же время цикл **for** более гибкий по сравнению с **foreach**. Если **foreach** последовательно извлекает элементы контейнера и только для чтения, то в цикле for мы можем перескакивать на несколько элементов вперед в зависимости от приращения счетчика, а также можем изменять элементы:Также можно использовать и другие виды циклов, например, while:**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:Визуально оба массива можно представить следующим образом:



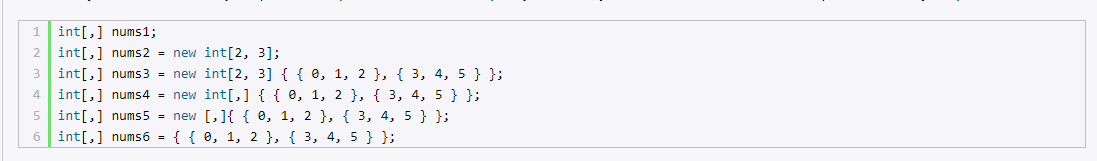
##### **Одномерный массив nums1**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 |

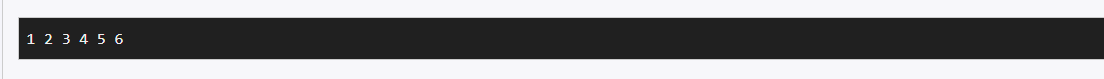
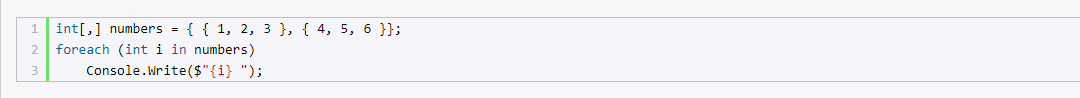
##### **Двухмерный массив nums2**

|  |  |  |
| --- | --- | --- |
| 0 | 1 | 2 |
| 3 | 4 | 5 |

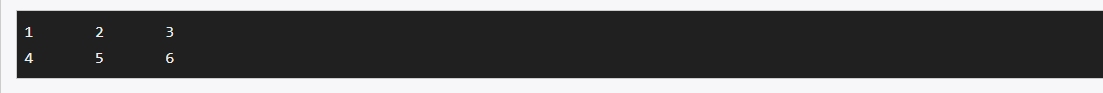
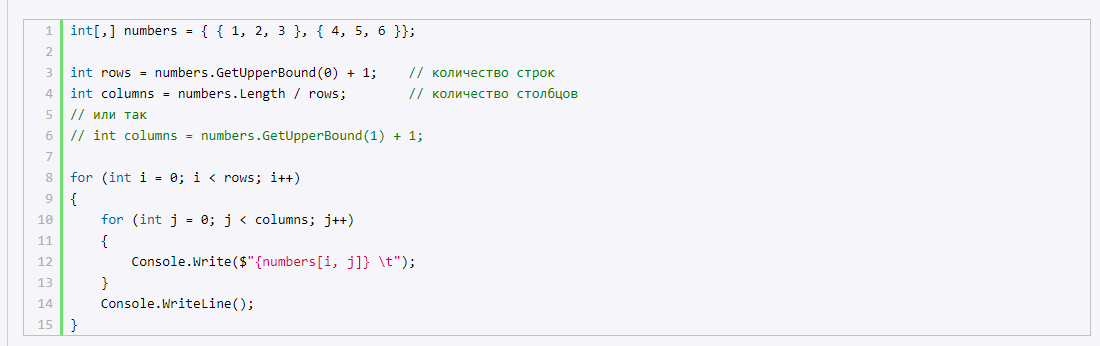
Поскольку массив nums2 двухмерный, он представляет собой простую таблицу. Все возможные способы определения двухмерных массивов:Массивы могут иметь и большее количество измерений. Объявление трехмерного массива могло бы выглядеть так:Соответственно могут быть и четырехмерные массивы и массивы с большим 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.В данном случае длина массива numbers равна 6. И цикл foreach выводит все элементы массива в строку:

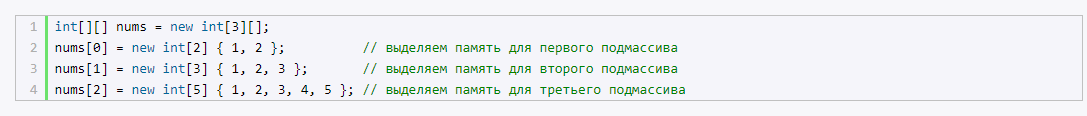


Но что если мы хотим отдельно пробежаться по каждой строке в таблице? В этом случае надо получить количество элементов в размерности. В частности, у каждого массива есть метод **GetUpperBound(номер\_размерности)**, который возвращает индекс последнего элемента в определенной размерности. И если мы говорим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

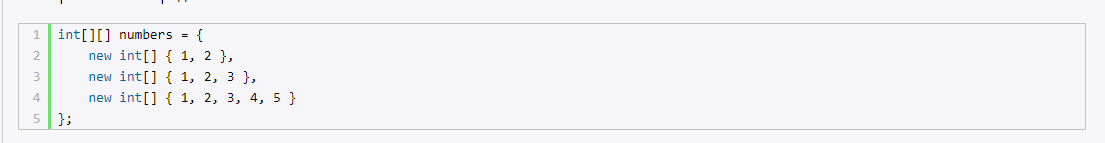


### **Массив массивов**

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

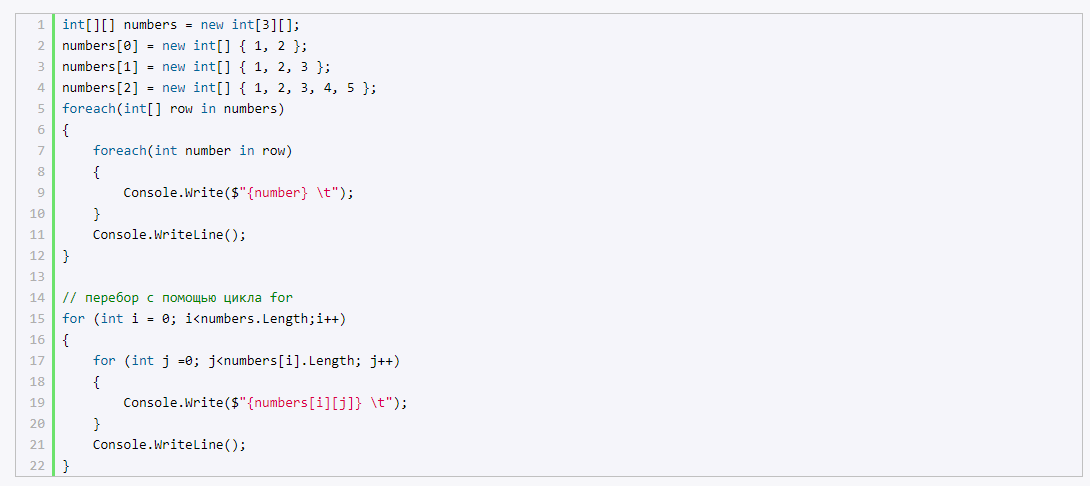


Альтернативное определение массива массивов:**Зубчатый массив nums**



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

Используя вложенные циклы, можно перебирать зубчатые массивы. Например:**Основные понятия массивов**



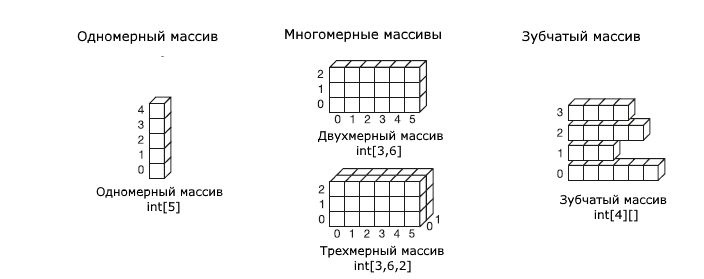
Суммируем основные понятия массивов:

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

Например, возьмем массивМассив 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:

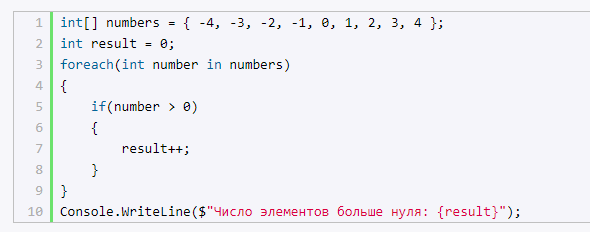


## **Задачи с массивами**

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

### **Количество положительных чисел**

Найдем количество положительных чисел в массиве: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: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:

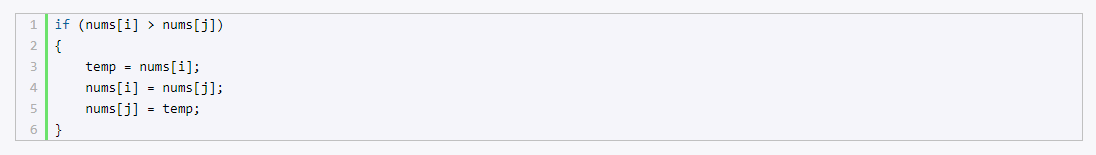


Для сортировки массива выполняем проходы по массиву и сравниваем элементы. Поскольку нам надо последовательно сравнивать каждый элемент массива с каждым (за исключением сравния с самим собой), то здесь применятся вложенный цикл.

Во внешнем цикле мы берем элемент, который будем сравнивать:



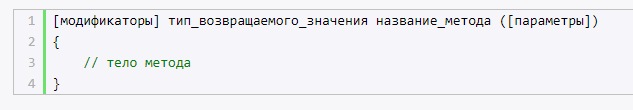
Далее запускаем вложенный цикл, который начинается, со следующего элемента, и из которого извлекаем элементы, с которыми будем сравнивать тот элемент, которые берется из массива во внешнем цикле:Если элемент с меньшим индексом больше элемента с большим индексом, то меняем элементы местами.В конце выводим 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.

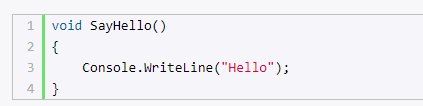
Общее определение методов выглядит следующим образом:Модификаторы и параметры необязательны.



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:



Здесь определен метод SayHello, который выводит некоторое сообщение. 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:



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).

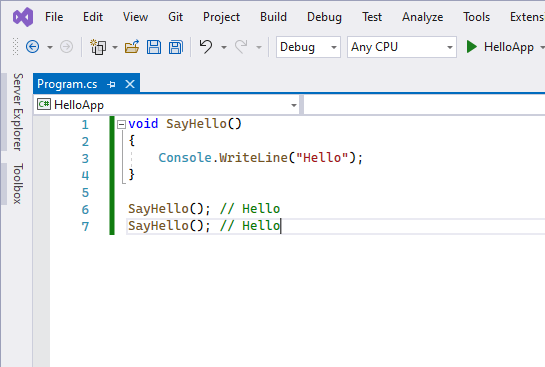
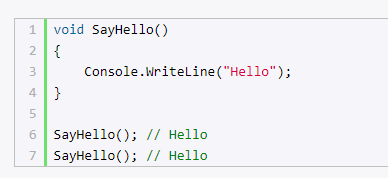


Например, вызов метода SayHello будет выглядеть следующим образом:

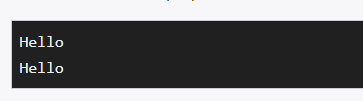


Поскольку метод не принимает никаких параметров, то после названия метода идут пустые скобки.

Объединим определение и вызов метода:



Консольный вывод программы:

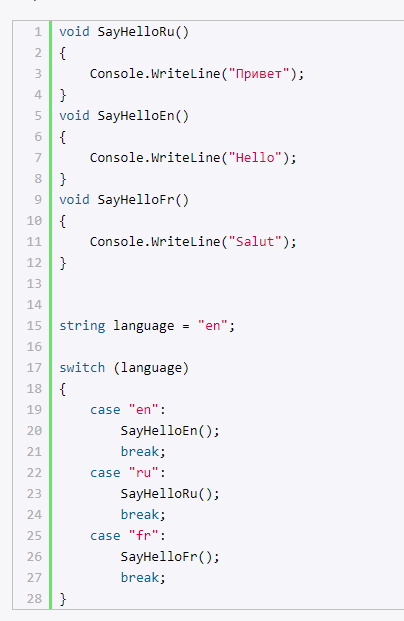


Преимуществом методов является то, что их можно повторно и многократно вызывать в различных частях программы. Например, в примере выше два раза вызывается метод SayHello.

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

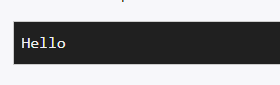


Определим и вызовем еще несколько методов:



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



### **Сокращенная запись методов**

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



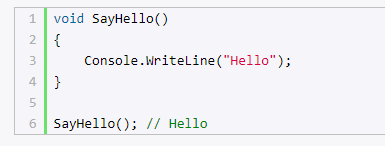
Мы можем его сократить следующим образом:



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

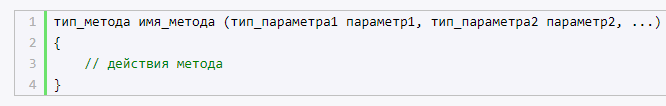
## **Параметры методов**

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



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



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



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

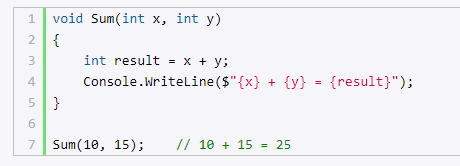
Чтобы выполнить метод, который имеет параметры, при вызове после имени метода в скобках ему передаются значения для его параметров, например:



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.

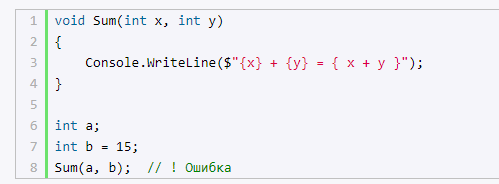
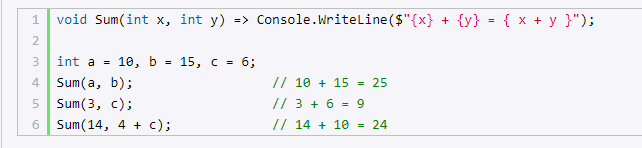
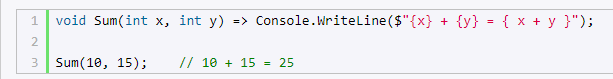
Определим еще один метод, который складывает два числа:



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.

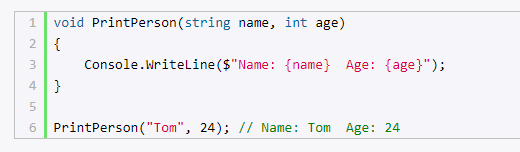
При вызове метода Sum значения передаются параметрам по позиции. Например, в вызове Sum(10, 15) число 10 передается параметру x, а число 15 - параметру y.

Также параметры могут использоваться в сокращеной версии метода:Передаваемые параметру значения могут представлять значения переменных или результат работы сложных выражений, которые возвращают некоторое значение:Если параметрами метода передаются значения переменных, то таким переменным должно быть присвоено значение. Например, следующая программа не скомпилируется:



**Соответствие параметов и аргументов по типу данных**

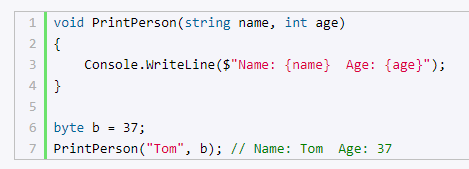
При передаче значений параметрам важно учитывать тип параметров: между аргументами и параметрами должно быть соответствие по типу. Например:



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.



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

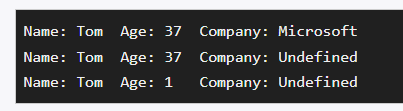
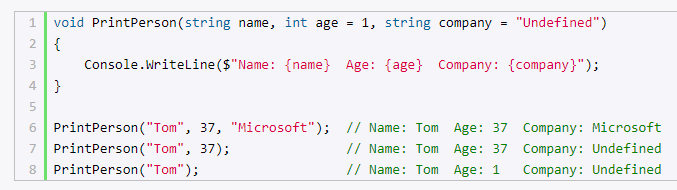
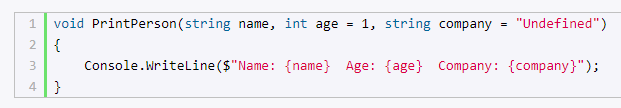


Здесь параметру типа int передается значение типа byte, но комилятор может автоматически преобразовать значение типа byte к тиу int. Поэтому здесь ошибки не возникнет. Какие преобразования типов могут быть выполнены автоматически, рассматривалось в одной из предыдущих тем: [Преобразования базовых типов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

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

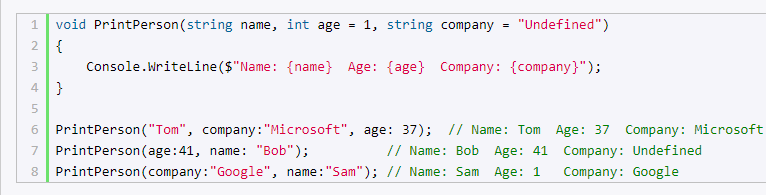


По умолчанию при вызове метода необходимо предоставить значения для всех его параметров. Но C# также позволяет использовать необязательные параметры. Для таких параметров нам необходимо объявить значение по умолчанию. Также следует учитывать, что после необязательных параметров все последующие параметры также должны быть необязательными:Здесь параметры age и company являются необязательными, так как им присвоены значения. Поэтому при вызове метода мы можем не передавать для них данные:Консольный вывод программы:

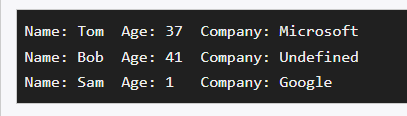


### **Именованные параметры**

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



Консольный вывод программы:



## **Возвращение значения и оператор return**

## Метод может возвращать значение, какой-либо результат. В примере выше были определены два метода, которые имели тип **void**. Методы с таким типом не возвращают никакого значения. Они просто выполняют некоторые действия.

Но методы также могут возвращать некоторое значение. Для этого применяется оператор **return**, после которого идет возвращаемое значение:

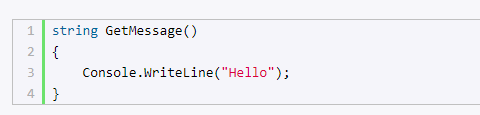


Например, определим метод, который возвращает значение типа string:

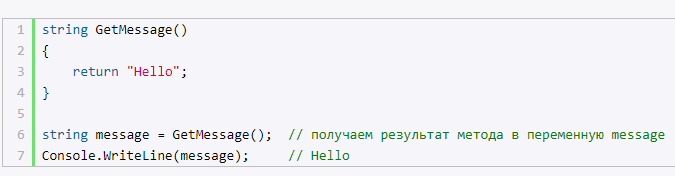
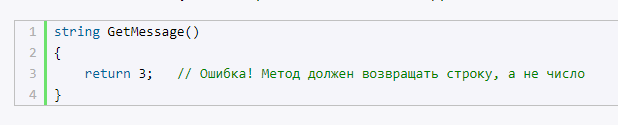


Метод GetMessage имеет тип string, следовательно, он должен возвратить строку. Поэтому в теле метода используется оператор **return**, после которого указана возвращаемая строка.

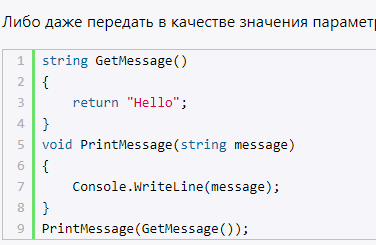
При этом методы, которые в качестве возвращаемого типа имеют любой тип, кроме **void**, обязательно должны использовать оператор return для возвращения значения. Например, следующее определение метода некорректно:



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:Результат методов, который возвращают значение, мы можем присвоить переменным или использовать иным образом в программе:Метод GetMessage() возвращает значение типа string. Поэтому мы можем присвоить это значение какой-нибудь переменной типа string: string message = GetMessage();

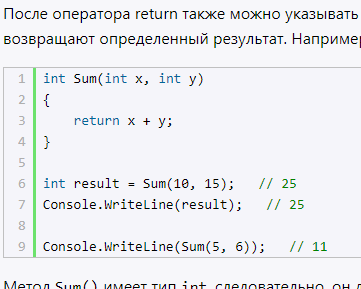


Либо даже передать в качестве значения параметру другого метода:



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

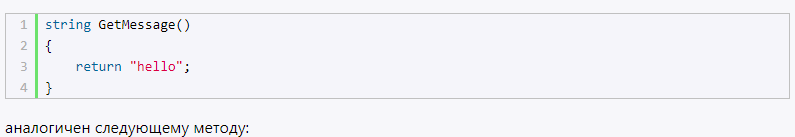
После оператора return также можно указывать сложные выражения или вызовы других методов, которые возвращают определенный результат. Например, определим метод, который возвращает сумму чисел:



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:



аналогичен следующему методу:



А метод

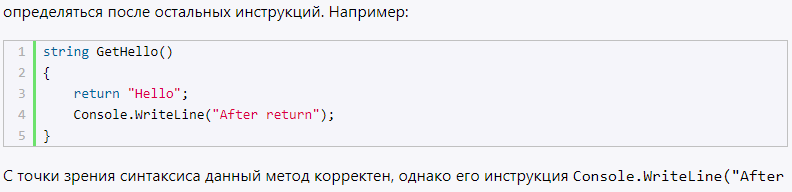


аналогичен следующему методу:

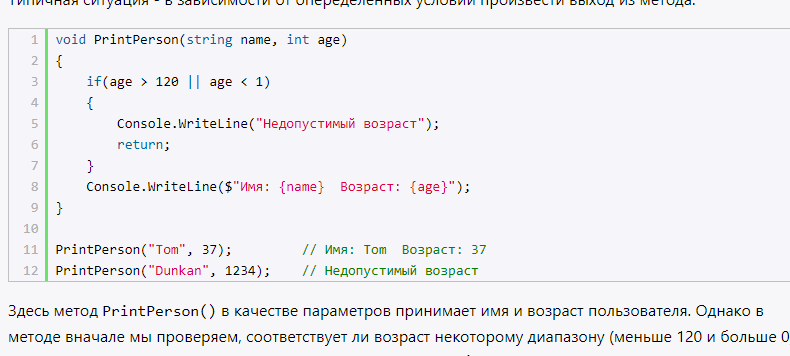


**Выход из метода**

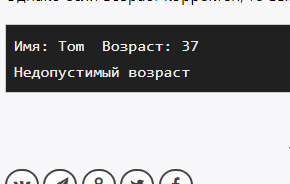
Оператор return не только возвращает значение, но и производит выход из метода. Поэтому он должен определяться после остальных инструкций. Например:С точки зрения синтаксиса данный метод корректен, однако его инструкция Console.WriteLine("After return") не имеет смысла - она никогда не выполнится, так 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: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:

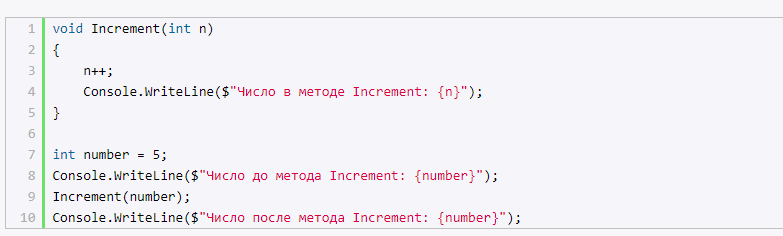


## **Передача параметров по ссылке и значению. Выходные параметры**

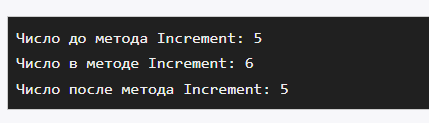
Существует два способа передачи параметров в метод в языке C#: **по значению** и **по ссылке**.

### **Передача параметров по значению**

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



Консольный вывод:

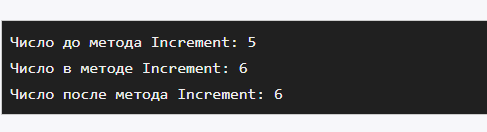


При передаче аргументов параметрам по значению параметр метода получает не саму переменную, а ее копию и далее работает с этой копией независимо от самой переменной.

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**:Консольный вывод:



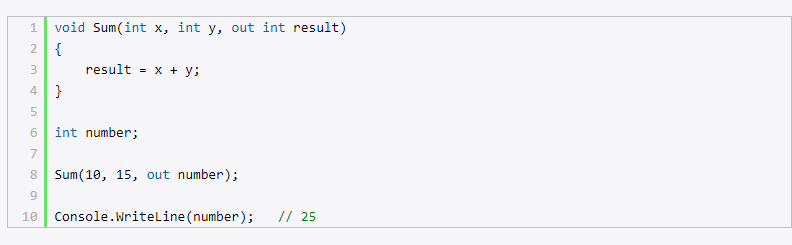
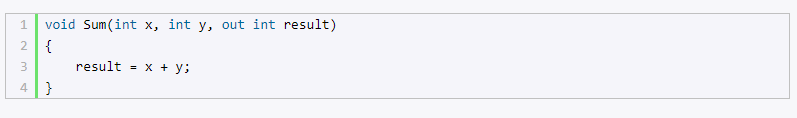
При передаче значений параметрам по ссылке метод получает адрес переменной в памяти. И, таким образом, если в методе изменяется значение параметра, передаваемого по ссылке, то также изменяется и значение переменной, которая передается на его место..

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.

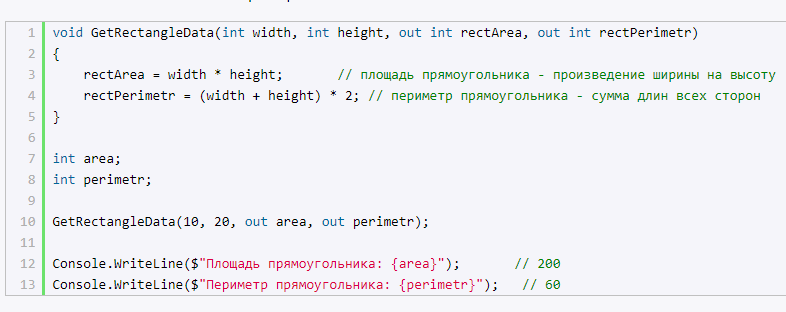
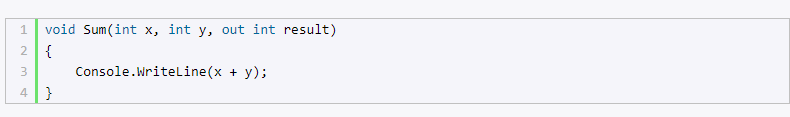
Обратите внимание, что модификатор ref указывается как перед параметром при объявлении метода, так и при вызове метода перед аргументом, который передается параметру.

### **Выходные параметры. Модификатор out**

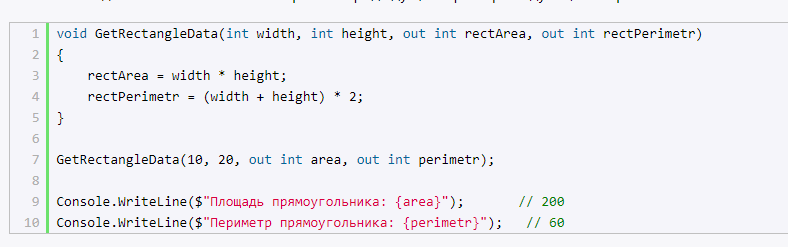
Выше мы использовали входные параметры. Но параметры могут быть также выходными. Чтобы сделать параметр выходным, перед ним ставится модификатор **out**:Здесь результат возвращается не через оператор return, а через выходной параметр result. Использование в программе:Причем, как и в случае с **ref** ключевое слово **out** используется как при определении метода, так и при его вызове.



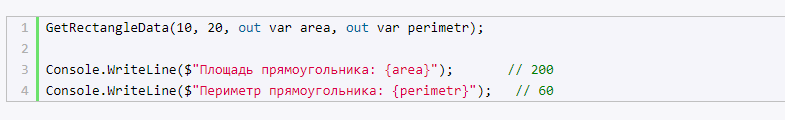
Также обратите внимание, что методы, использующие такие параметры, обязательно должны присваивать им определенное значение. То есть следующий код будет недопустим, так как в нем для out-параметра не указано никакого значения:Прелесть использования подобных параметров состоит в том, что по сути мы можем вернуть из метода не одно значение, а несколько. Например: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.



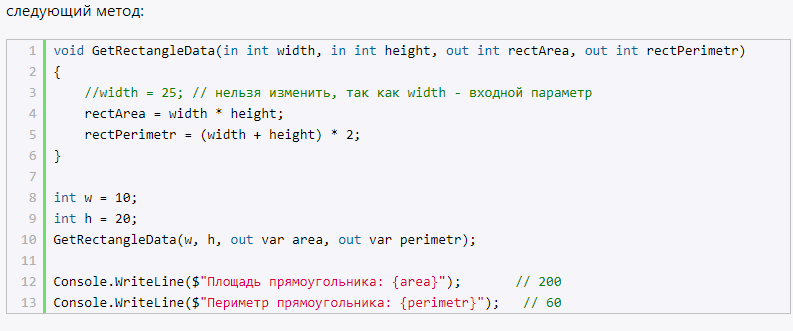
При этом можно определять переменные, которые передаются **out**-параметрам в непосредственно при вызове метода. То есть мы можем сократить предыдущий пример следующим образом:



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

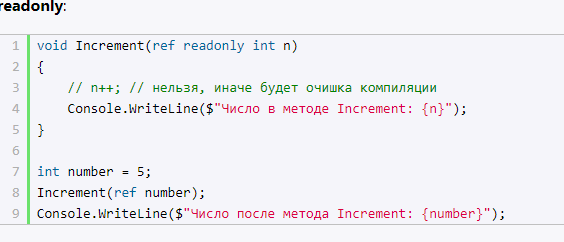


В данном случае через входные параметры width и height в метод передаются значения, но в самом методе мы не можем изменить значения этих параметров, так как они определены с модификатором **in**.

Передача по ссылке в некоторых случаях может увеличить произодительность, а использование оператора **in** гарантирует, что значения переменных, которые передаются параметрам, нельзя будет изменить в этом методе.

### **ref-параметры только для чтения**

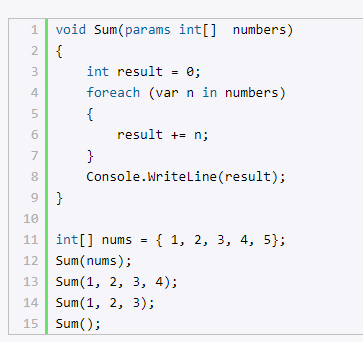
В примерах выше можно было изменять значение **ref**-параметра. Однако иногда это может быть нежелательно. И чтобы гарантировать, что **ref**-параметр не изменит своего значения, начиная с версии C# 12 можно применять **ref**-параметры только для чтения. Такие параметры предваряются ключевым словом **readonly**:



В данном случае в метод Increment параметр n передается по ссылке и при этом он доступен только для чтения. При попытке изменить его значение мы получим ошибку на этапе компиляции.

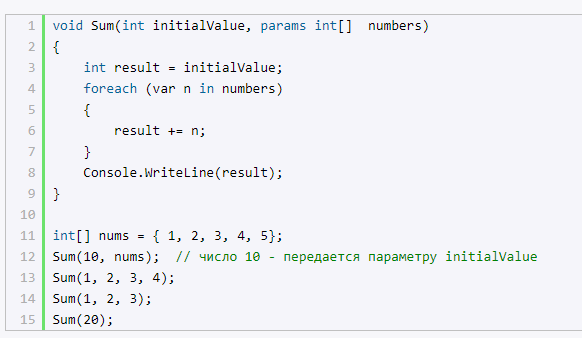
## **Массив параметров и ключевое слово params**

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

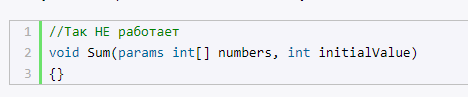


Сам параметр с ключевым словом **params** при определении метода должен представлять одномерный массив того типа, данные которого мы собираемся использовать. При вызове метода на место параметра с модификатором **params** мы можем передать как отдельные значения, так и массив значений, либо вообще не 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: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.

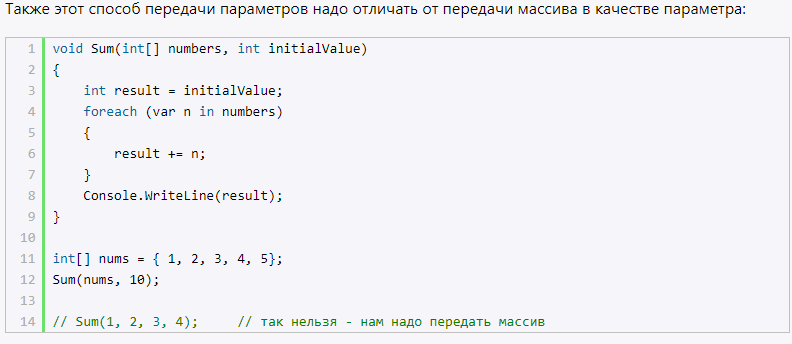


Однако после параметра с модификатором **params** мы НЕ можем указывать другие параметры. То есть следующее определение метода недопустимо:



**Массив в качестве параметра**

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



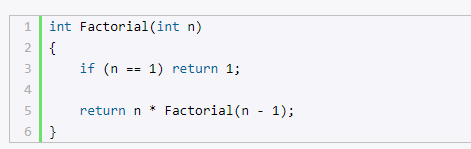
## **Рекурсивные функции**

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:



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. Факториалы всех остальных положительных чисел будет начинаться с вычисления факториала числа 1, который равен 1.

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



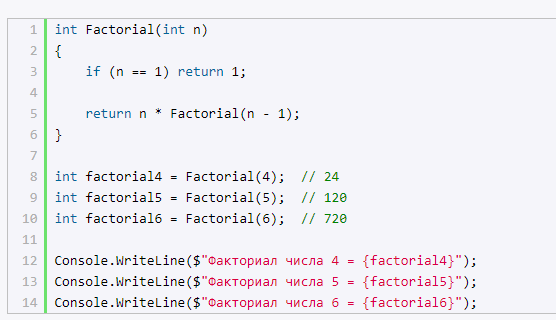
То есть, если вводимое число равно 1, то возвращается 1

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



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



1. Сначала идет проверка, равно ли число единице:  
   Но вначале n равно 4, поэтому это условие ложно, и соответственно выполняется код



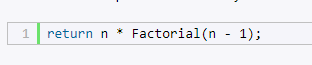
То есть фактически мы имеем:



Далее выполняется выражение:



Опять же n не равно 1, поэтому выполняется код  
То есть фактически:



Далее выполняется выражение:



Опять же n не равно 1, поэтому выполняется код  
То есть фактически:



Далее выполняется выражение:



Теперь n равно 1, поэтому выполняется код  
И возвращается 1.



В итоге выражение

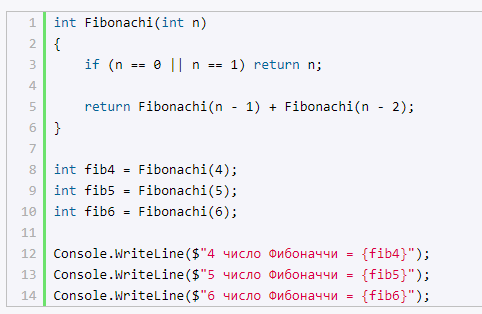


В реальности выливается в



**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:



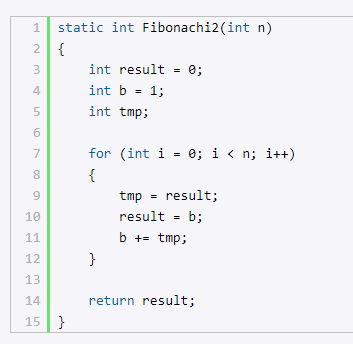
Здесь базовый вариант выглядит следующий образом:



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:

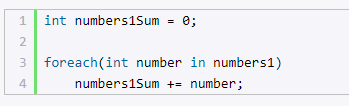
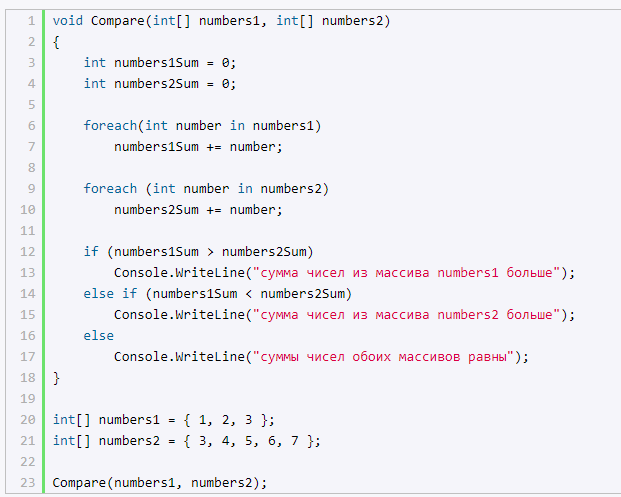


В то же время в некоторых ситуациях рекурсия предоставляет элегантное решение, например, при обходе различных древовидных представлений, к примеру, дерева каталогов и файлов.

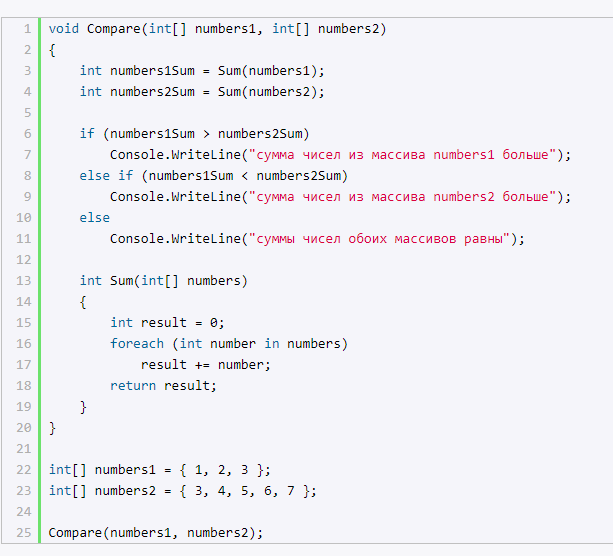
## **Локальные функции**

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: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:



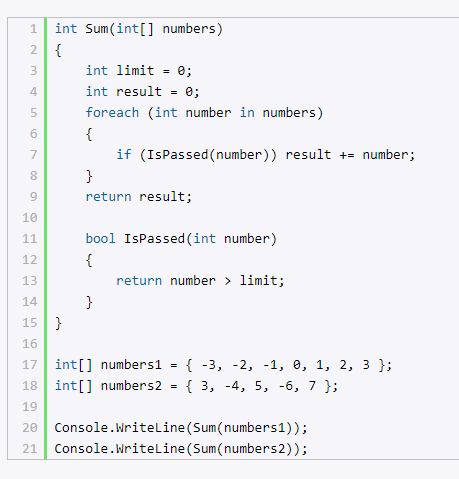
К тому а что, если мы захотим сравнивать сумму только положительных или четных чисел или как-то иначе изменить логику сравнения? В этом лучше вынести повторяющиеся действия в отдельный метод. Однако если эти действия нигде больше в прогамме не будут вызываться и будут использоваться только в одном методе,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

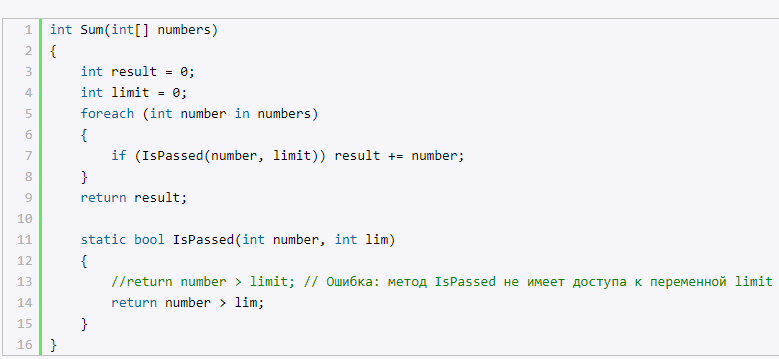
Локальные функции могут быть статическими. Такие функции определяются с помощью ключевого слова **static**. Их особенностью является то, что они не могут обращаться к переменным окружения, то есть метода, в котором статическая функция определена.

Сначала определим локальную функцию, которая имеет доступ окружению:



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.

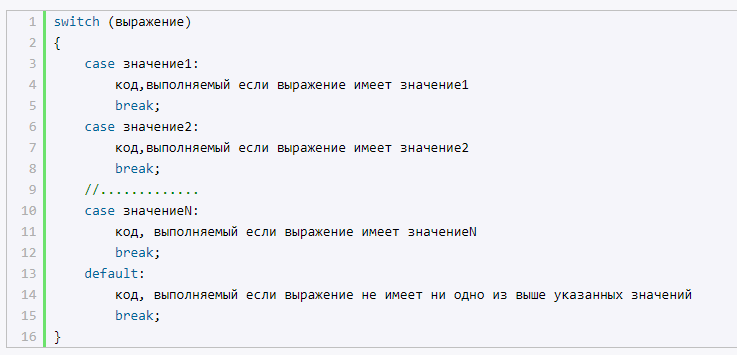
Теперь сделаем функцию IsPassed статической:Модификатор **static** указывается перед типом локальной функции. Теперь функция IsPassed не может обращаться к переменной limit, и в этом случае нам надо либо передать это значение в виде параметра, либо определить переменную limit непосредственно в локальной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



## **Конструкция switch**

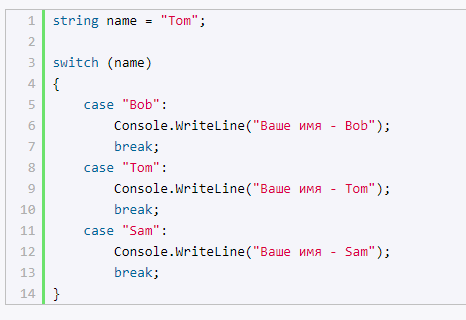
Конструкция **switch/case** оценивает некоторое выражение и сравнивает его значение с набором значений. И при совпадении значений выполняет определенный код.:

Конструкция switch имеет следующее формальное определение:После ключевого слова **switch** в скобках идет сравниваемое выражение. Значение этого выражения последовательно сравнивается со значениями, помещенными после оператора case. И если совпадение будет найдено, то будет выполняться определенный блок case.

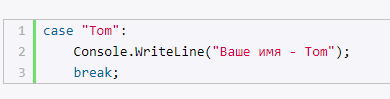


В конце каждого блока case должен ставиться один из операторов перехода: **break**, **goto case**, **return** или **throw**. Как правило, используется оператор **break**. При его применении другие блоки case выполняться не будут.

Например:



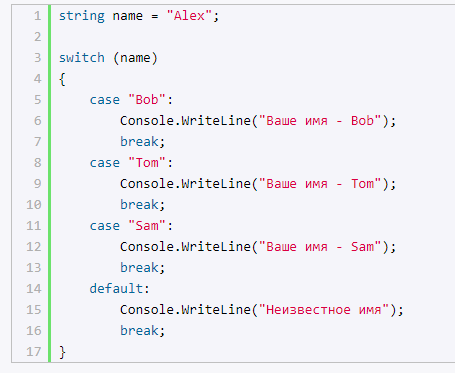
В данном случае конструкция switch последовательно сравнивает значение переменной name с набором значений, которые указаны после операторов case. Поскольку здесь значение переменной name - строка "Tom", то будет выполняться блок



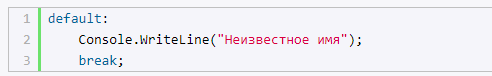
Соответственно мы увидим на консоли



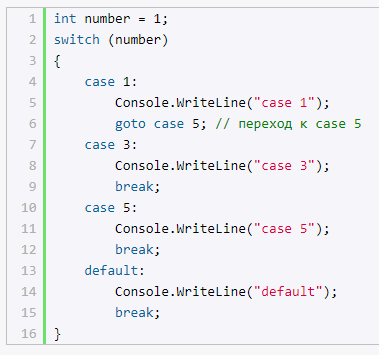
Если значение переменной name не совпадает ни с каким значением после операторов case, то ни один из блоков case не выполняется. Однако если даже в этом случае нам все равно надо выполнить какие-нибудь действия, то мы можем добавить в конструкцию **switch** необязательный блок **default**. Например:



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

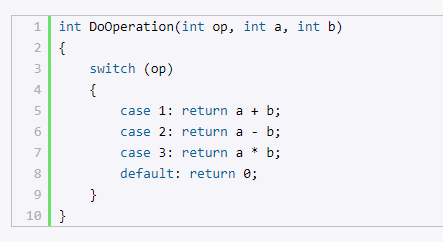


Однако если мы хотим, чтобы, наоборот, после выполнения текущего блока case выполнялся другой блок case, то мы можем использовать вместо break оператор goto case:



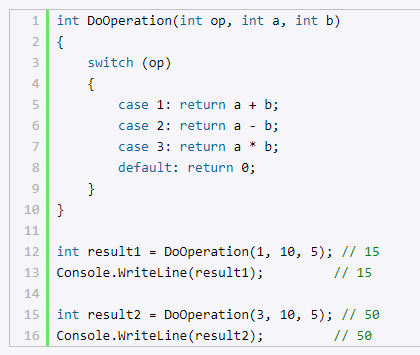
**Возвращение значения из switch**

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



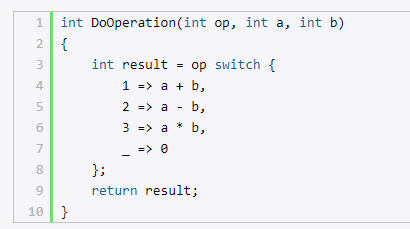
В метод DoOperation() передается числовой код операции и два операнда. В зависимости от кода операции над операндами выполнется определенная операция и ее результат возвращается из метода. Для примера при по умолчанию из метода возвращается 0, если код операции не равен 1, 2 или 3.

Затем мы можем вызвать этот метод:



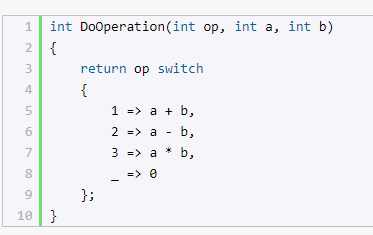
**Получение результата из switch**

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

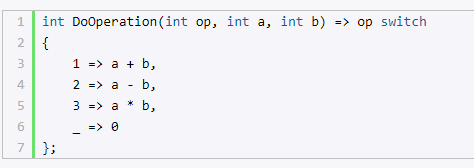


Теперь не требуется оператор **case**, а после сравниваемого значения ставится оператор стрелка **=>**. Значение справа от стрелки выступает в качестве возвращаемоего значения. Кроме того, вместо оператора **default** используется почерк \_. В итоге результат конструкции switch будет присвиваиваться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

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

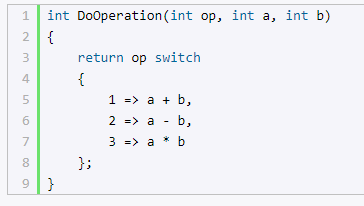


Или сделать метод еще короче:



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

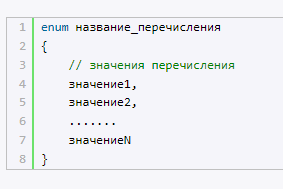


Эта версия метода возвращает значение, если код операции равен 1, 2 или 3. Но что, если будет передано значение 4 или какое-то другое? Поэтому данная версия метода даже не скомпилируется. Поэтому нам надо предусмотреть возвращение значения из метода при всех возможных вариантах. То есть, мы можем, как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

## **Перечисления enum**

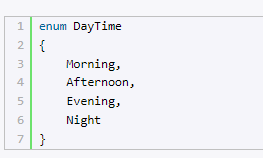
Кроме примитивных типов данных в языке программирования C# есть такой тип как **enum** или **перечисление**. Перечисления представляют набор логически связанных констант.

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



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

Определим простейшее перечисление:



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:

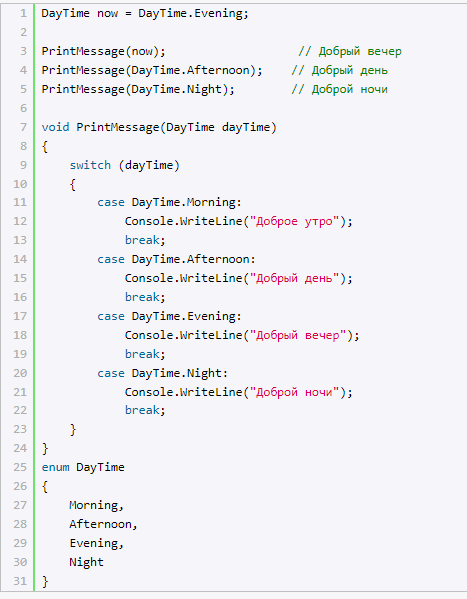


Далее в программе мы можем использовать подобные переменные/константы/параметры как и любые другие:



**Хранение состояния**

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



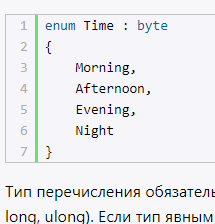
Здесь метод PrintMessage() в качестве параметра принимает значение типа перечисления DayTime и зависимости от этого значения выводит определенное приведение.

Другой пример: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.



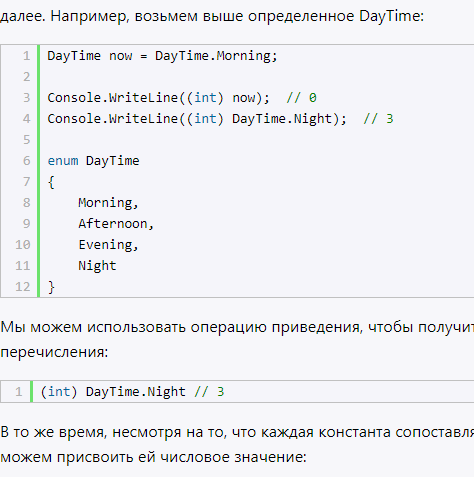
### **Тип и значения констант перечисления**

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

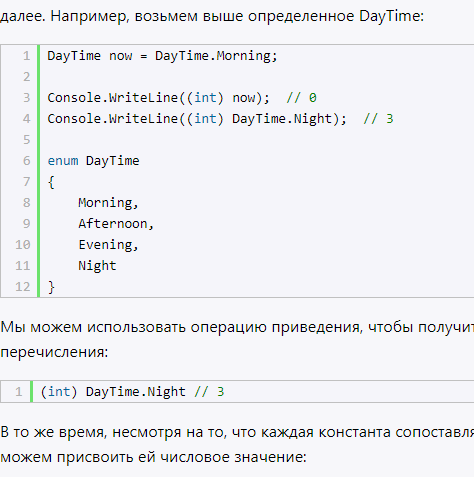


Тип перечисления обязательно должен представлять целочисленный тип (byte, sbyte, short, ushort, int, uint, long, ulong). Если тип явным образом не указан, то по умолчанию используется тип **int**.

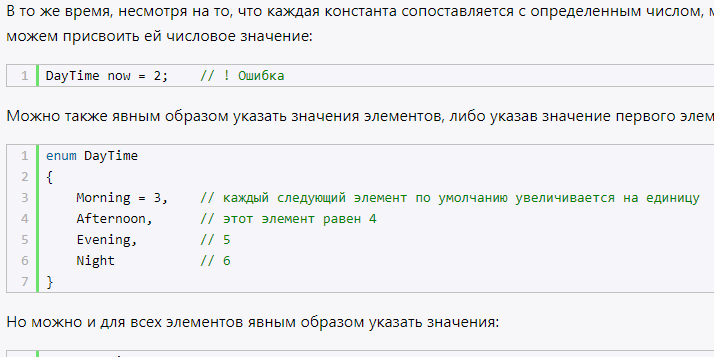
Тип влияет на значения, которые могут иметь константы. По умолчанию каждому элементу перечисления присваивается целочисленное значение, причем первый элемент будет иметь значение 0, второй - 1 и так далее. Например, возьмем выше определенное DayTime:



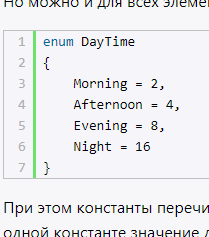
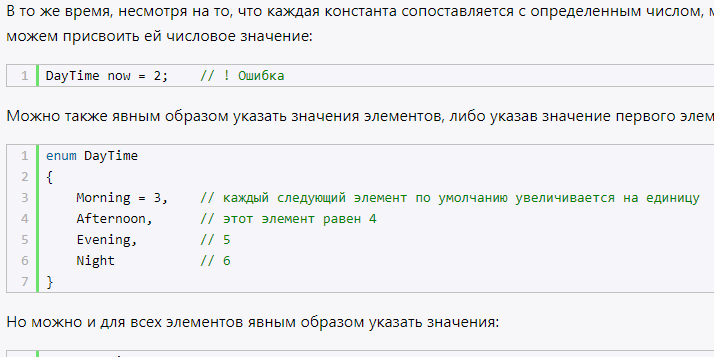
Мы можем использовать операцию приведения, чтобы получить целочисленное значение константы перечисления:



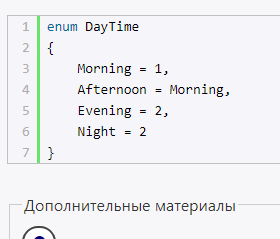
В то же время, несмотря на то, что каждая константа сопоставляется с определенным числом, мы НЕ можем присвоить ей числовое значение:



Можно также явным образом указать значения элементов, либо указав значение первого элемента:Но можно и для всех элементов явным образом указать значения:



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



# 

# **Классы, структуры и пространства имен**

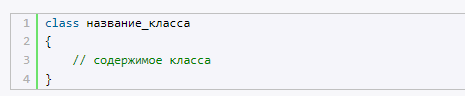
## **Классы и объекты**

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

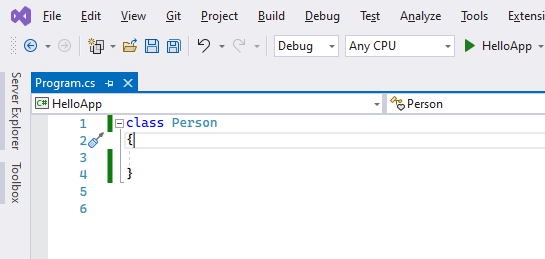
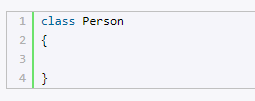
Описанием объекта является **класс**, а объект представляет экземпляр этого класса. Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке, у которого есть имя, возраст, какие-то другие характеристики. То есть некоторый шаблон - этот шаблон можно назвать классом. 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**:



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:



Начиная с версии C# 12, если класс имеет пустое определение, то фигурные скобки после названия типа можно не использовать:



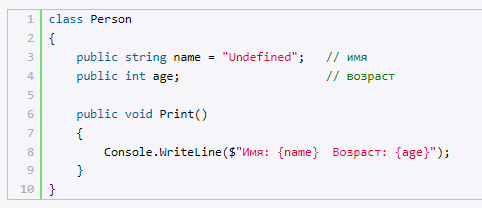
Однако такой класс не особо показателен, поэтому добавим в него некоторую функциональность.

### **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:



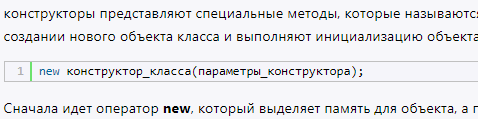
В данном случае в классе Person определено поле name, которое хранит имя, и поле age, которое хранит возраст человека. В отличие от переменных, определенных в методах, поля класса могут иметь модификаторы, которые указываются перед полем. Так, в данном случае, чтобы все поля были доступны вне класса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

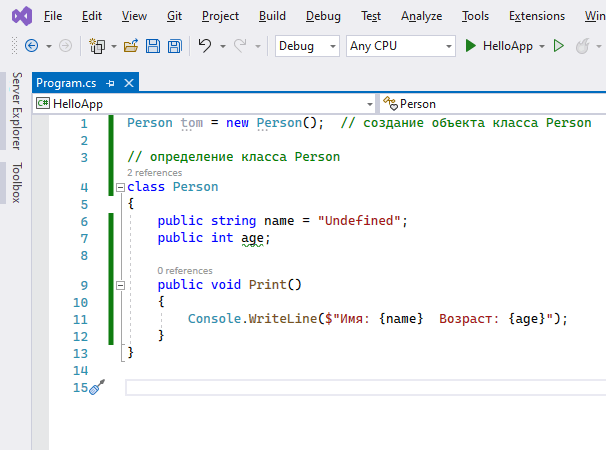


Сначала идет оператор **new**, который выделяет память для объекта, а после него идет вызов **конструктора**.

#### **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.

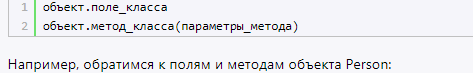
Теперь создадим объект класса Person:



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**

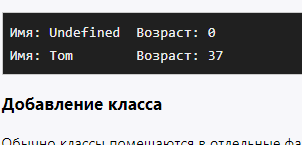
Для обращения к функциональности класса - полям, методам (а также другим элементам класса) применяется точечная нотация точки - после объекта класса ставится точка, а затем элемент класса:



Например, обратимся к полям и методам объекта Person:

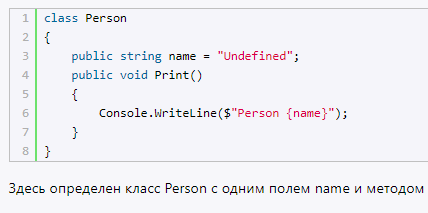


Консольный вывод данной программы:



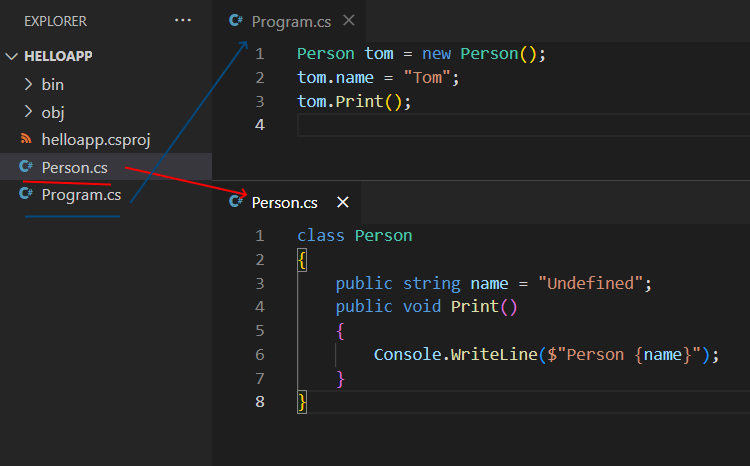
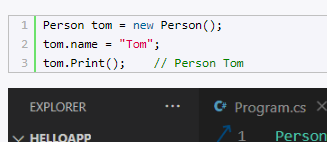
### **Добавление класса**

Обычно классы помещаются в отдельные файлы. Нередко для одного класса предназначен один файл. Если мы работаем над проектом вне среды Visual Studio, используя .NET CLI, то нам достаточно добавить новый файл класса в папку проекта. Например, добавим новый файл, который назовем **Person.cs** и в котором определим следующий код:



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

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

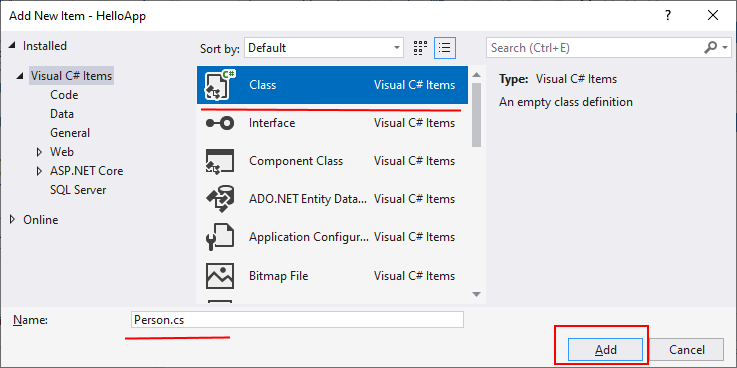


Visual Studio предоставляет по умолчанию встроенные шаблоны для добвления класса. Для добавления класса нажмем в Visual Studio правой кнопкой мыши на название проекта:



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 котором можно определить тот же код и также использовать в файле Program.cs.

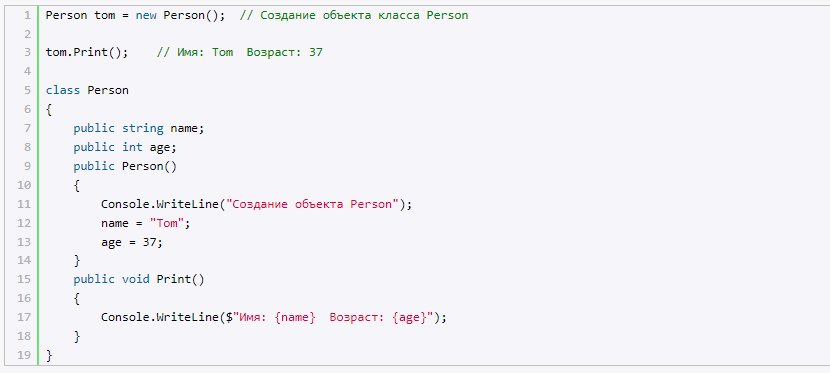
Таким образом, мы можем определять классы в отдельных файлах и использовать их в программе.

## **Конструкторы, инициализаторы и деконструкторы**

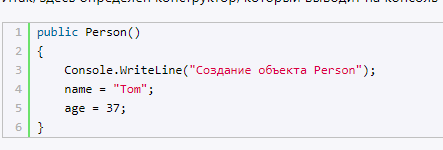
### **Создание конструкторов**

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:



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



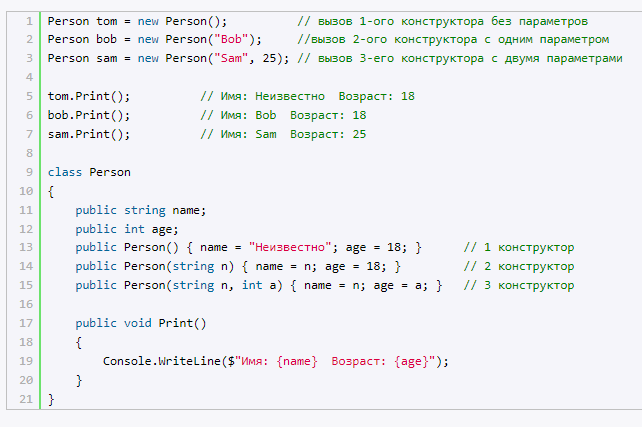
Конструкторы могут иметь модификаторы, которые указываются перед именем конструктора. Так, в данном случае, чтобы конструктор был доступен вне класса Person, он определен с модификатором **public**.

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

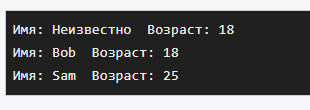


В данном случае выражение Person() как раз представляет вызов определенного в классе конструктора (это больше не автоматический конструктор по умолчанию, которого у класса теперь нет). Соответственно при его выполнении на консоли будет выводиться строка "Создание объекта Person"

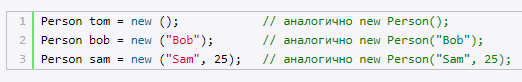
Подобным образом мы можем определять и другие конструкторы в классе. Например, изменим класс Person следующим образом:Теперь в классе определено три конструктора, каждый из которых принимает различное количество параметров и устанавливает значения полей класса. И мы можем вызвать один из этих конструкторов для создания объекта класса.



Консольный вывод данной программы:

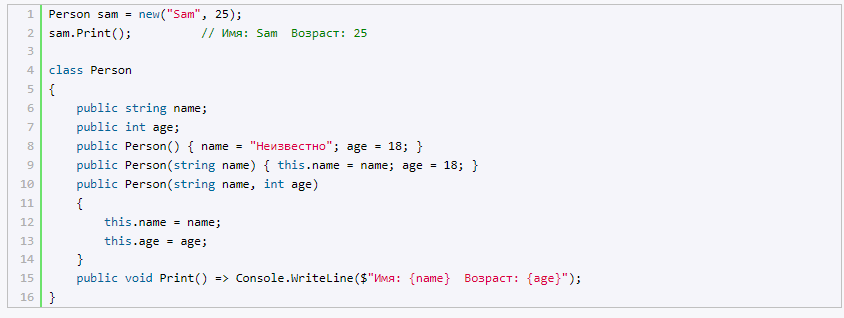


Стоит отметить, что начиная с версии C# 9 мы можем сократить вызов конструктора, убрав из него название типа:



**Ключевое слово this**

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

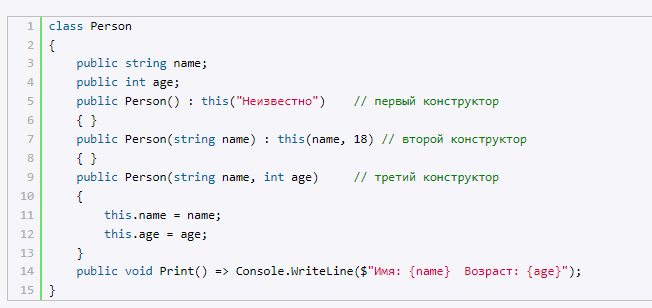


**this.name = name;**

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: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:



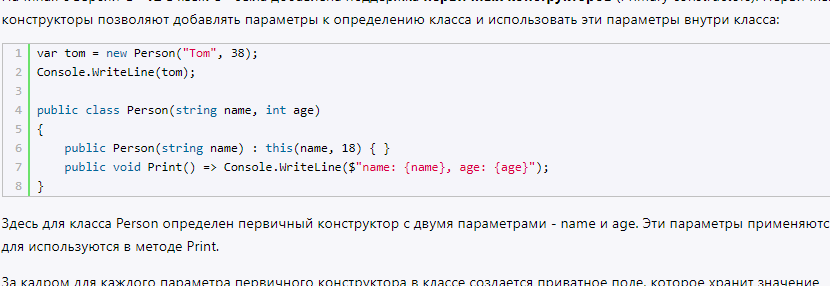
идет обращение к третьему конструктору, которому передаются два значения. Причем в начале будет выполняться именно третий конструктор, и только потом код второго конструктора.

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:И если при вызове конструктора мы не передаем значение для какого-то параметра, то применяется значение по умолчанию.



### **Первичные конструкторы**

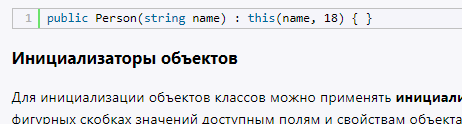
Начиная с версии **C# 12** в язык C# была добавлена поддержка **первичных конструкторов** (Primary constructors). Первичные конструкторы позволяют добавлять параметры к определению класса и использовать эти параметры внутри класса:



Здесь для класса Person определен первичный конструктор с двумя параметрами - name и age. Эти параметры применяются для используются в методе Print.

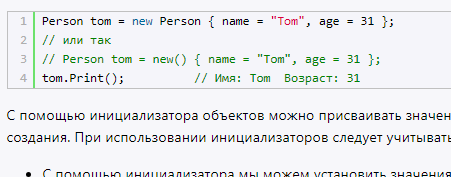
За кадром для каждого параметра первичного конструктора в классе создается приватное поле, которое хранит значение параметра. Благодаря этому они могут использоваться в теле класса.

Кроме первичных конструкторов класс может определять дополнительные конструкторы, как примере выше. Но эти дополнительные конструкторы должны вызывать первичный конструктор:



**Инициализаторы объектов**

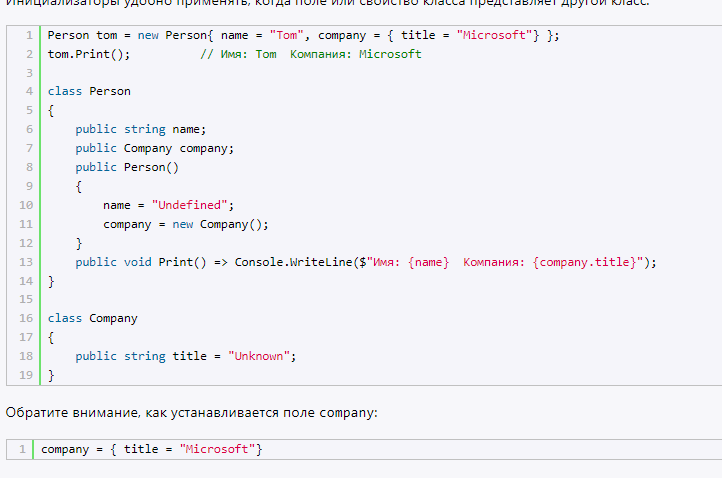
Для инициализации объектов классов можно применять **инициализаторы**. Инициализаторы представляют передачу в фигурных скобках значений доступным полям и свойствам объекта:



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:

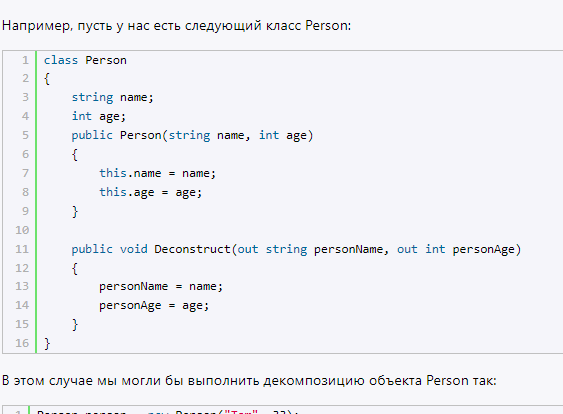


Обратите внимание, как устанавливается поле company:**Деконструкторы**

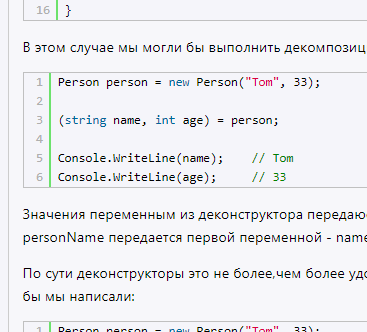


Деконструкторы (не путать с деструкторами) позволяют выполнить декомпозицию объекта на отдельные части.

Например, пусть у нас есть следующий класс Person:

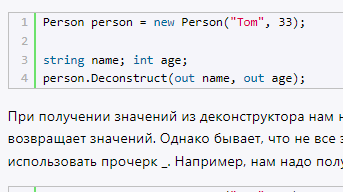


В этом случае мы могли бы выполнить декомпозицию объекта Person так:

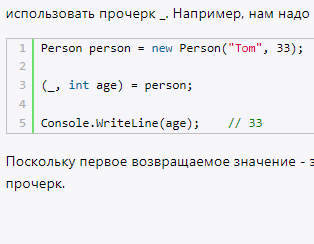


Значения переменным из деконструктора передаюся по позиции. То есть первое возвращаемое значение в виде параметра personName передается первой переменной - name, второе возващаемое значение - переменной age.

По сути деконструкторы это не более,чем более удобный способ разложения объекта на компоненты. Это все равно, что если бы мы написали:



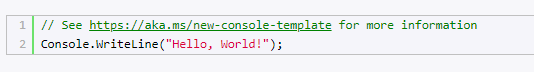
При получении значений из деконструктора нам необходимо предоставить столько переменных, сколько деконструктор возвращает значений. Однако бывает, что не все эти значения нужны. И вместо возвращаемых значений мы можм использовать прочерк **\_**. Например, нам надо получить только возраст пользователя:



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:



## 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:Определение метода **Main** обязательно начинается с модификатора static, которое указывает, что метод Main - статический. Позже мы подробнее разберем, что все это значит.



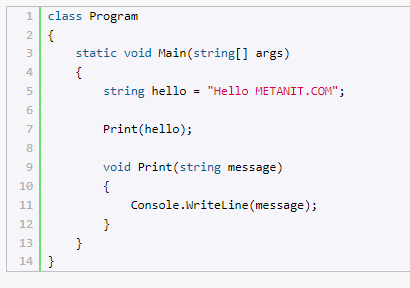
Возвращаемым типом метода **Main** обязательно является тип **void**. Кроме того, в качестве параметра он принимает массив строк - string[] args - в реальной программе это те параметры, через которые при запуске программы из консоли мы можем передать ей некоторые значения. Внутри метода располагаются 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.

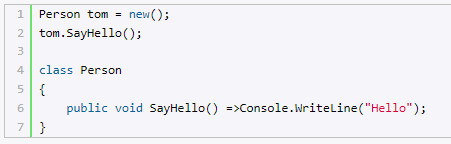
Если мы определяем какие-то переменные, константы, методы и обращаемся к ним, они помещаются в метод Main. Например, следующая программа верхнего уровня



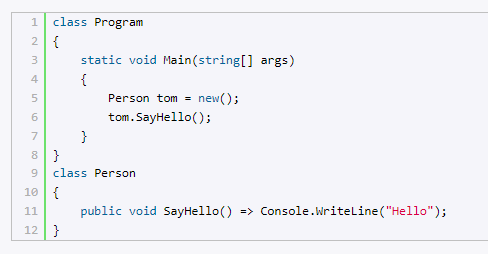
будет аналогична следующей программе:



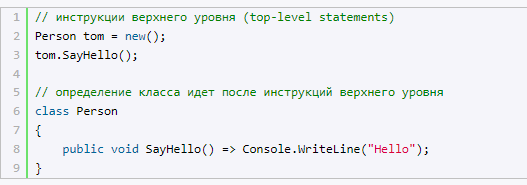
Если определяются новые типы, например, классы, то они помещаются вне класса Program. Например, код:



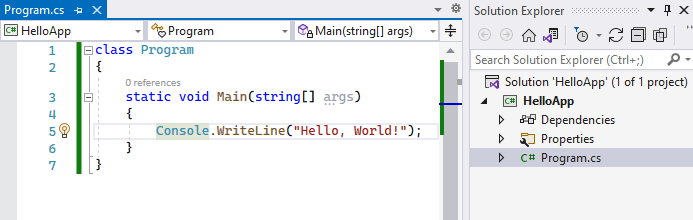
будет аналогичен следующему



Однако надо учитывать, что опредления типов (в частности классов) должны идти в конце файла после инструкций верхнего уровня. То есть:



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.

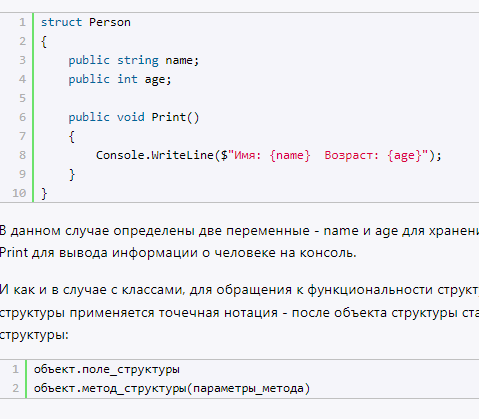
Например, определим структуру, которая будет называться Person и которая будет представлять человека:

|  |  |
| --- | --- |
| 1  2  3 | struct Person  {  } |

Начиная с версии C# 12, если структура имеет пустое определение (не содержат полей, свойств, методов), то фигурные скобки после названия типа можно не использовать:

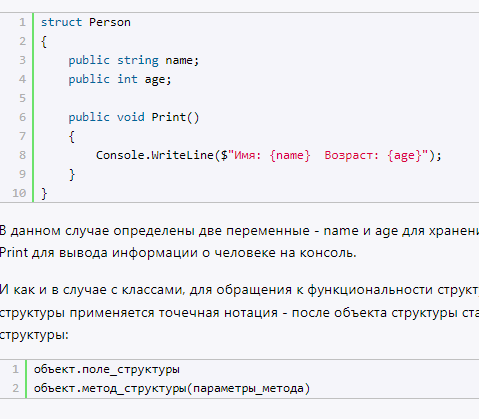
|  |  |
| --- | --- |
| 1 | struct Person; |

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:



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.

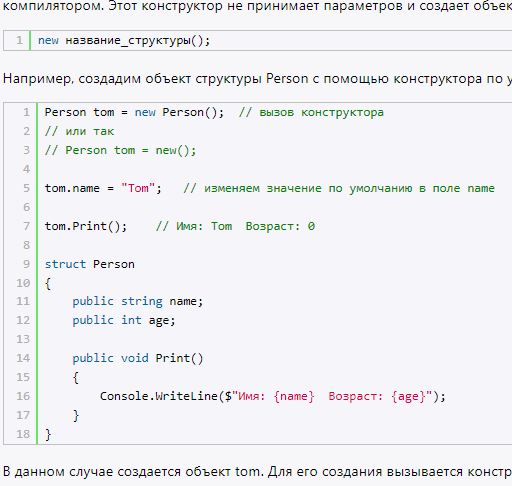
И как и в случае с классами, для обращения к функциональности структуры - полям, методам и другим компонентам структуры применяется точечная нотация - после объекта структуры ставится точка, а затем указывается компонент структуры:



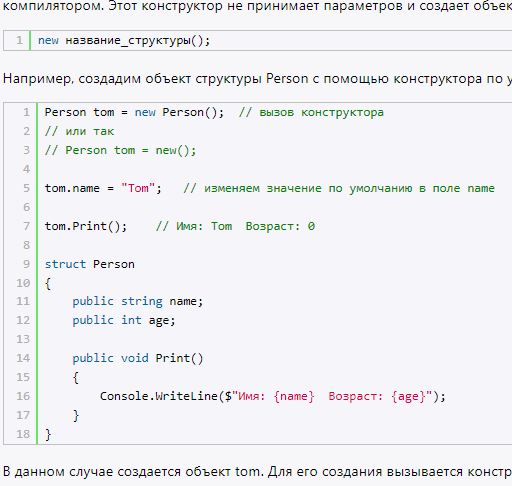
**Создание объекта структуры**

#### **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.



Например, создадим объект структуры Person с помощью конструктора по умолчанию:

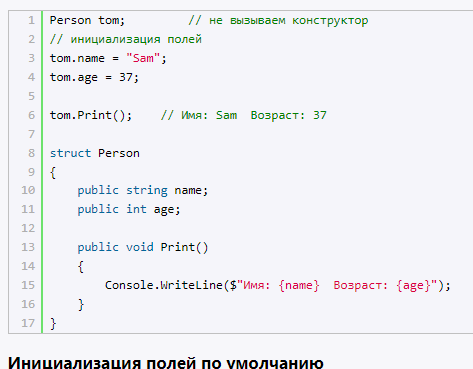


В данном случае создается объект tom. Для его создания вызывается конструктор по умолчанию, который устанавливает значения по умолчанию для его полей. Для числовых данных это значение 0, поэтому поле age будет иметь значение 0. Для строк это значение null, которое указывает на отсутствие значения. Но далее, если поля доступны (а в данном случае поскольку они имеют модификатор public они доступны), мы можем изменить их значения. Так, здесь полю name присваивается строка "Tom". Соответственно при выполнении метода Print() мы получим следующий консольный вывод:

Имя: Tom Возраст: 0

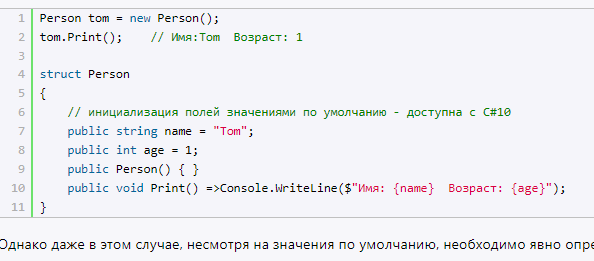
#### **Непосредственная иницилизация полей**

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:



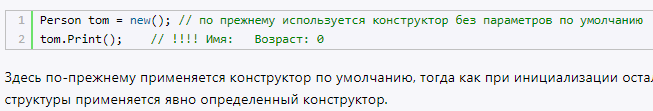
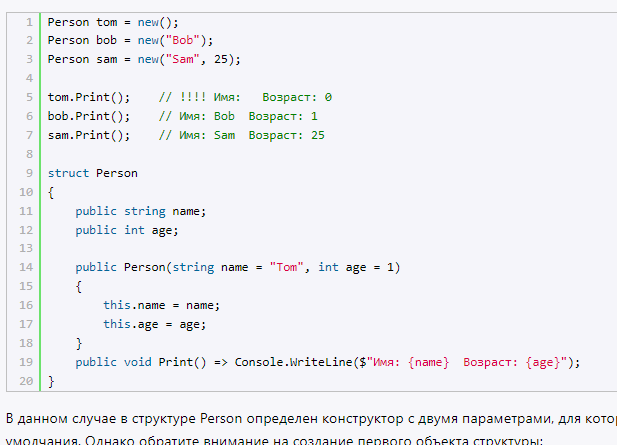
**Инициализация полей по умолчанию**

Стоит отметить, что начиная с версии C# 10, мы можем напрямую инициализировать поля структуры при их определении (до C# 10 это делать было нельзя):Однако даже в этом случае, несмотря на значения по умолчанию, необходимо явно определить и вызывать конструктор, если мы хотим использовать эти значения.

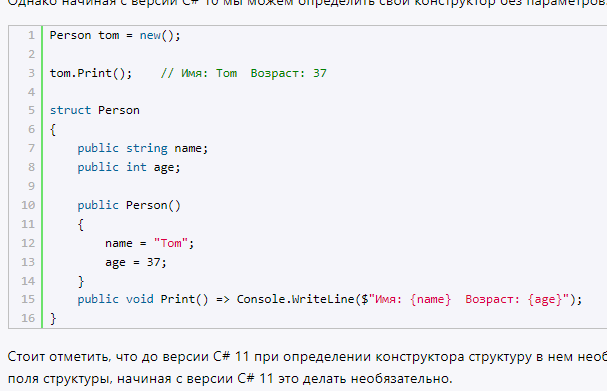


### **Конструкторы структуры**

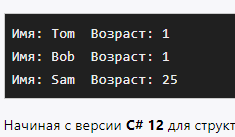
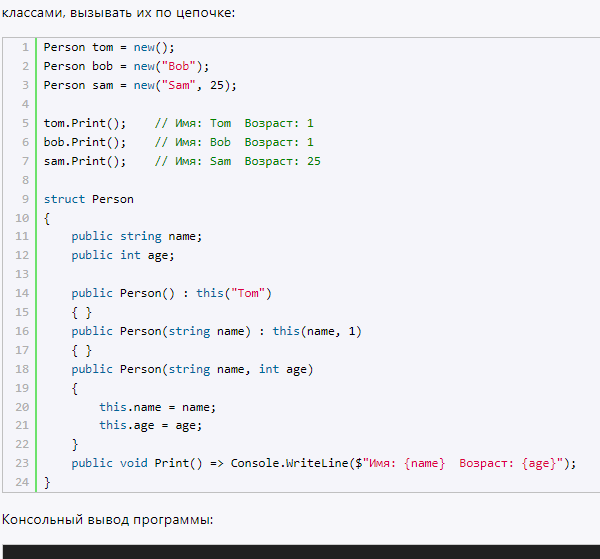
Как и класс, структура может определять конструкторы. Например, добавим в структуру Person конструктор:В данном случае в структуре Person определен конструктор с двумя параметрами, для которых предоставлены значения по умолчания. Однако обратите внимание на создание первого объекта структуры:Здесь по-прежнему применяется конструктор по умолчанию, тогда как при инициализации остальных двух переменных структуры применяется явно определенный конструктор.



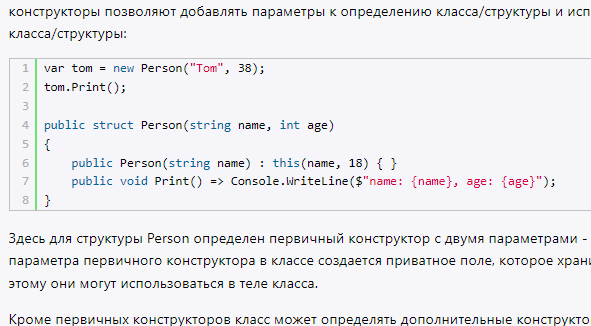
Однако начиная с версии C# 10 мы можем определить свой конструктор без параметров:Стоит отметить, что до версии C# 11 при определении конструктора структуру в нем необходимо было инициализировать все поля структуры, начиная с версии C# 11 это делать необязательно.



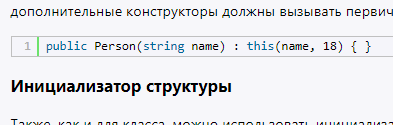
В случае если нам необходимо вызывать конструкторы с различным количеством параметров, то мы можем, как и в случае с классами, вызывать их по цепочке:Консольный вывод программы:



Начиная с версии **C# 12** для структур, как и для классов, можно определять первичные конструкторы. Первичные конструкторы позволяют добавлять параметры к определению класса/структуры и использовать эти параметры внутри класса/структуры:Здесь для структуры Person определен первичный конструктор с двумя параметрами - name и age. За кадром для каждого параметра первичного конструктора в классе создается приватное поле, которое хранит значение параметра. Благодаря этому они могут использоваться в теле класса.

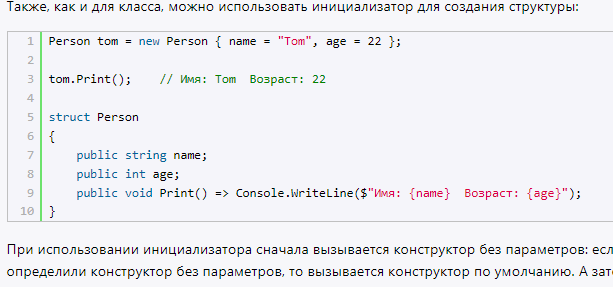


Кроме первичных конструкторов класс может определять дополнительные конструкторы, как примере выше. Но эти дополнительные конструкторы должны вызывать первичный конструктор:



**Инициализатор структуры**

Также, как и для класса, можно использовать инициализатор для создания структуры: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**:



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)

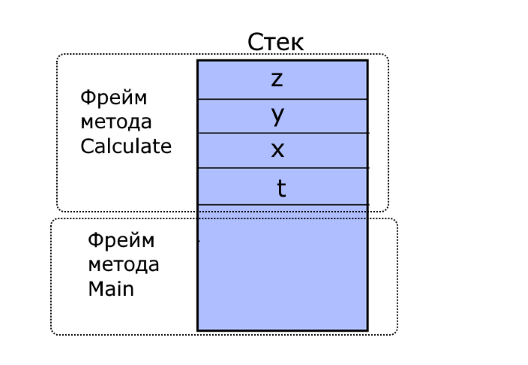
В чем же между ними различия? Для этого надо понять организацию памяти в .NET. Здесь память делится на два типа: стек и куча (heap). Параметры и переменные метода, которые представляют типы значений, размещают свое значение в стеке. Стек представляет собой структуру данных, которая растет снизу вверх: 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:



При запуске такой программы в стеке будут определяться два фрейма - для метода Main (так как он вызывается при запуске программы) и для метода Calculate:



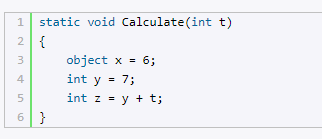
При вызове этого метода Calculate в его фрейм в стеке будут помещаться значения t, x, y и z. Они определяются в контексте данного метода. Когда метод отработает, область памяти, которая выделялась под стек, впоследствии может быть использована другими методами.

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.

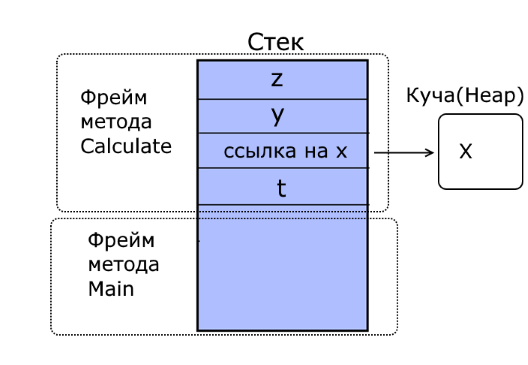
Ссылочные типы хранятся в куче или хипе, которую можно представить как неупорядоченный набор разнородных объектов. Физически это остальная часть памяти, которая доступна процессу.

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:

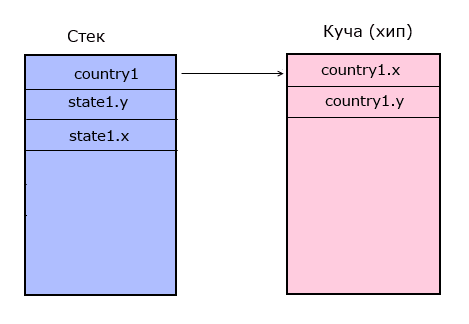
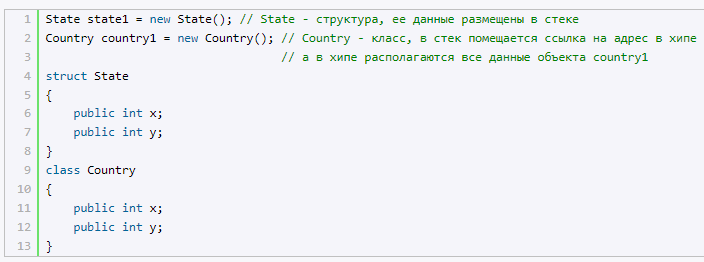


То теперь значение переменной x будет храниться в куче, так как она представляет ссылочный тип object, а в стеке будет храниться ссылка на объект в куче.



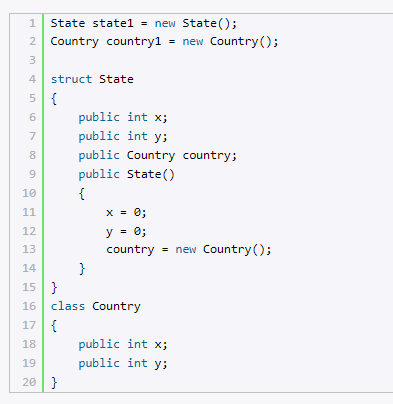
### **Составные типы**

Теперь рассмотим ситуацию, когда тип значений и ссылочный тип представляют составные типы - структуру и класс: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..

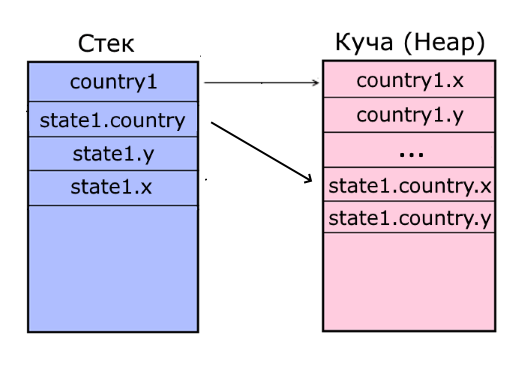


Таким образом, в стеке окажутся все поля структуры state1 и ссылка на объект country1 в хипе.

Но, допустим, в структуре State также определена переменная ссылочного типа Country. Где она будет хранить свое значение, если она определена в типе значений?



Значение переменной state1.country также будет храниться в куче, так как эта переменная представляет ссылочный тип:



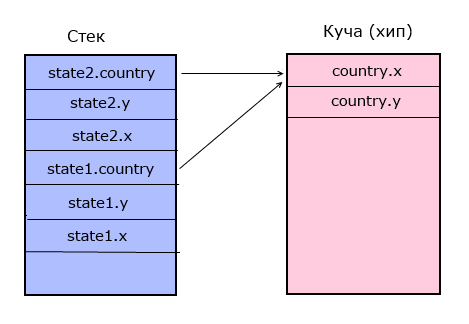
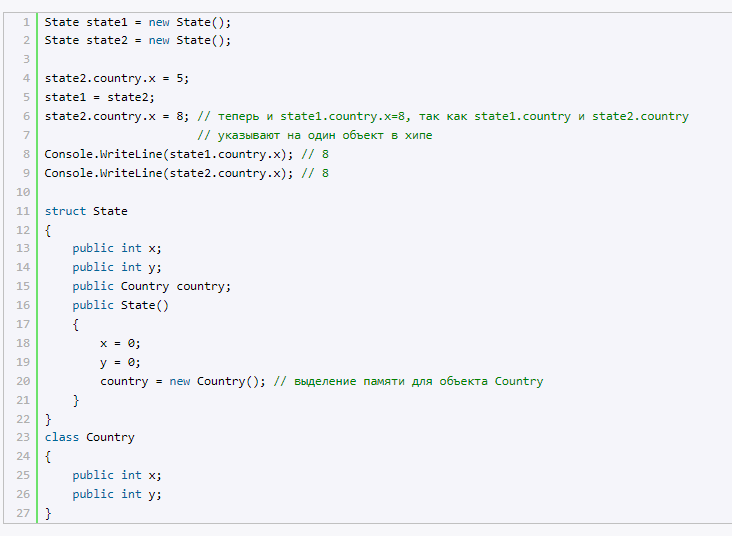
### **Копирование значений**

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: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: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:



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:



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.

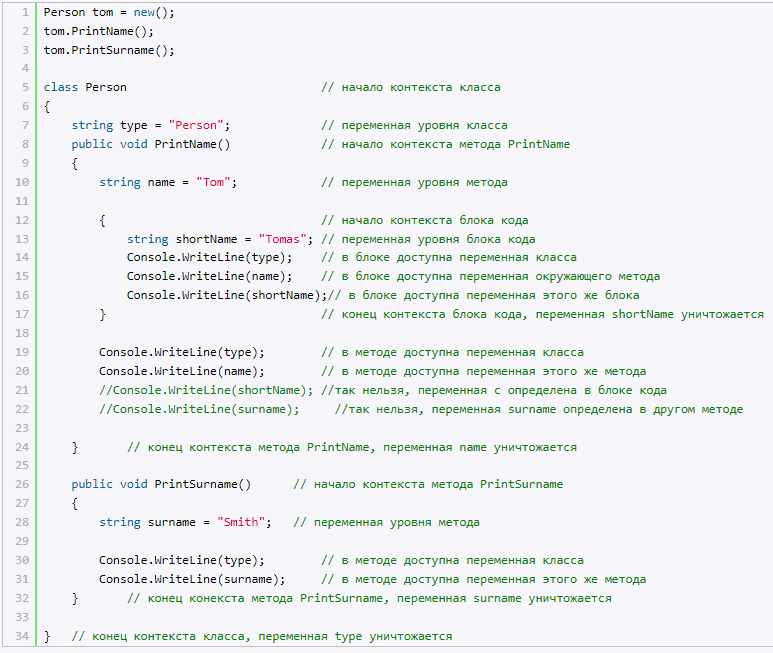
## **Область видимости (контекст) переменных и констант**

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 код программы определен следующим образом: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.

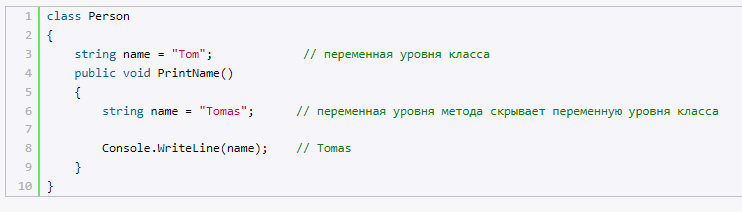


Переменная name существует только в рамках метода PrintName. Также как и переменная surname существует в рамках метода PrintSurname. В методе PrintName мы не можем обратиться к переменной surname, так как она в другом контексте.

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.

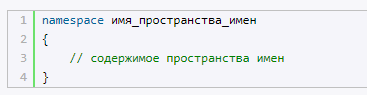
При работе с переменными надо учитывать, что локальные переменные, определенные в методе или в блоке кода, скрывают переменные уровня класса, если их имена совпадают:При объявлении переменных также надо учитывать, что в одном контексте нельзя определить несколько переменных с одним и тем же именем.



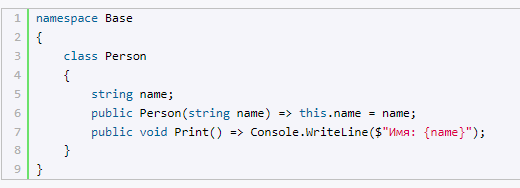
## **Пространства имен**

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.

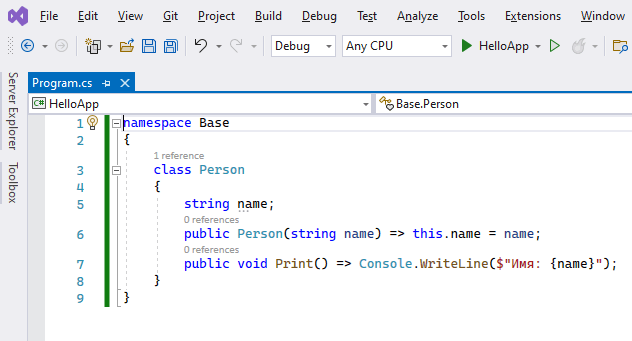
Для определения пространства имен применяется ключевое слово namespace, после которого идет название название пространства имен:



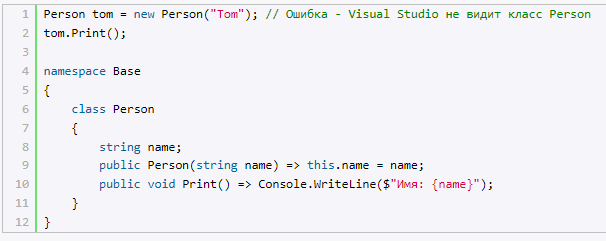
Например, определим в файле **Program.cs** пространство имен, которое будет называться **Base**



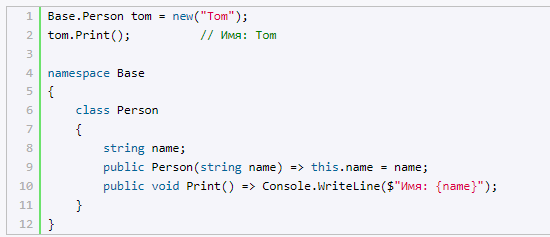
Здесь пространство имен содержит класс Person, которой имеет одну переменную - name, конструктор и метод Print.



Теперь попробуем использовать класс Person: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:

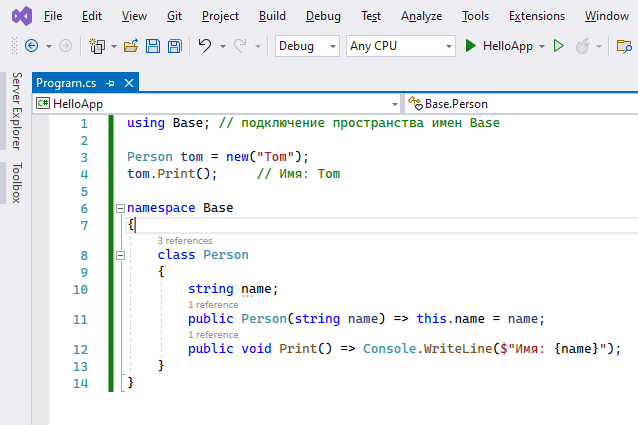
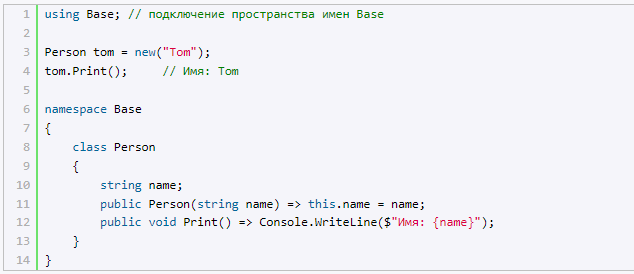


Поскольку класс Person располагается в пространстве имен Base, то его полное имя - **Base.Person**. Соответственно, используя это имя, мы можем обращаться к данному классу вне его пространства имен.



### **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**:

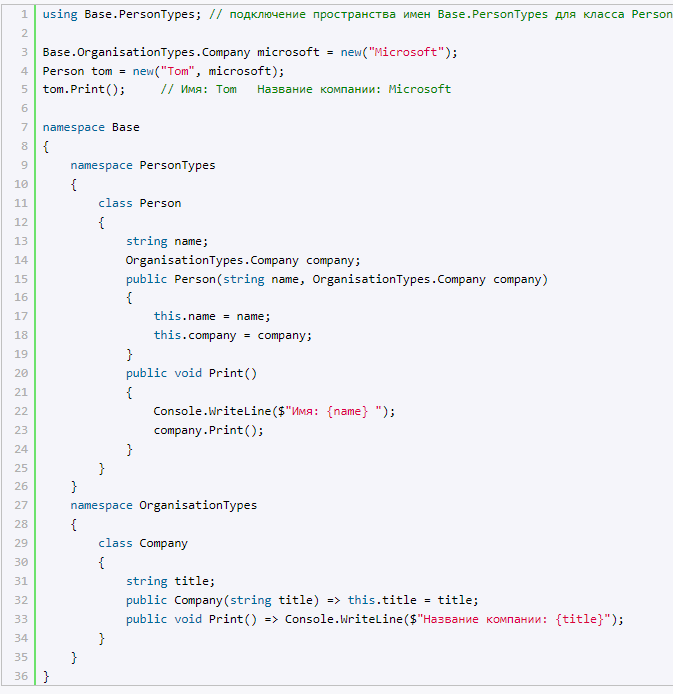


### **Вложенные пространства имен**

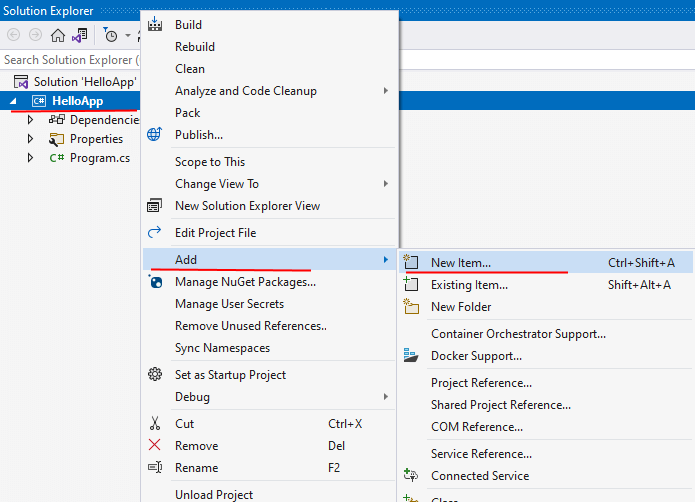
Одни пространства имен могут содержать другие. Например:В данном случае классы Person и Company находятся в разных вложенных пространствах имен, поэтому чтобы в классе Person использовать класс Company, надо прописывать имя класса с учетом его пространства имен: OrganisationTypes.Company (так как оба 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.



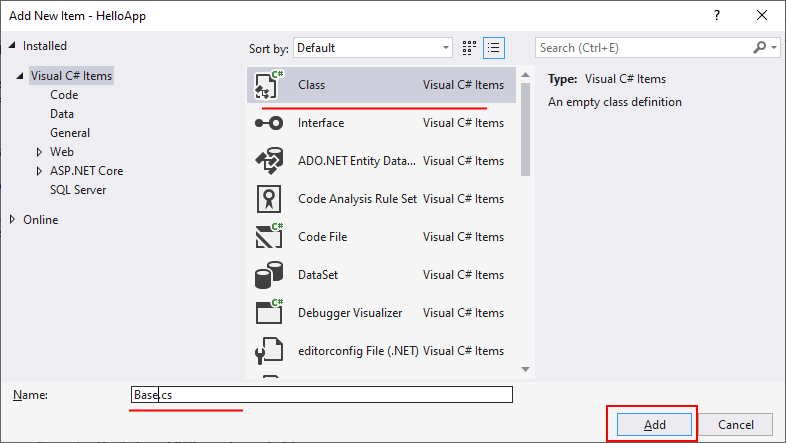
Для обращения к этим классам вне пространства Base необходимо использовать всю цепочку пространств имен:**Пространства имен уровня файла**



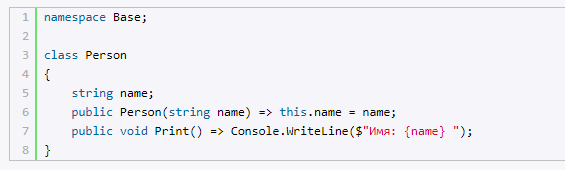
Начиная с .NET 6 и C# 10 можно определять пространства имен на уровне файла. Например, добавим в проект новый файл с кодом c#. Для этого в Visual Studio нажмем на название проекта правой кнопкой мыши и в появившемся меню выберем пункт **Add -> New Item...**



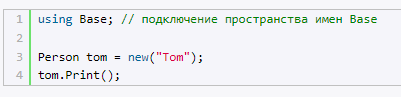
В окне добавления нового элемента выберем пункт **Class** и поле **Name** в качестве имени файла укажем **Base.cs**



После этого в проект будет добавлен файл **Base.cs**. Удалим из него все содержимое по умолчанию и определим в нем следующий код:Директива namespace Base в начале файла указывает, что содержимое файла будет представлять пространство имен Base.



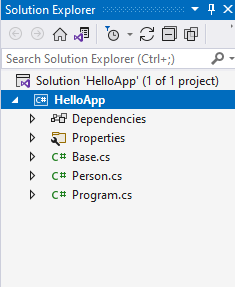
Теперь подключим это пространство имен в файле **Program.cs**:



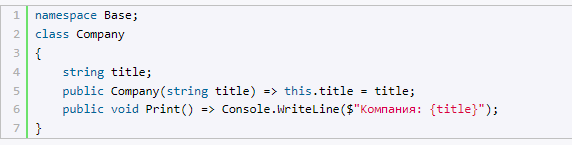
## **Глобальные пространства имен**

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.

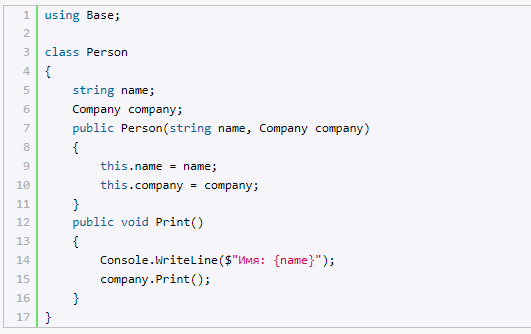
Например, пусть у нас в проекте есть три файла с кодом:



В файле **Base.cs** определяется класс Company в пространстве **Base**:

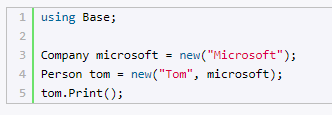


В файле **Person.cs** определен класс Person, который использует класс Company:



Чтобы использовать класс Company мы подключаем его пространство имен: using Base;

В классе **Program.cs** используются классы Person и Company:

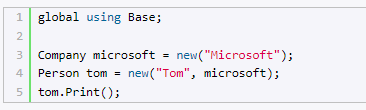


Опять же чтобы использовать класс Company, подключается его пространство имен: using Base;

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:



Например, изменим файл **Program.cs** следующим образом:

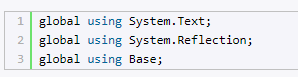


Теперь пространство Base подключается во все файлы кода в проекте. И из файла **Person.cs** мы можем убрать строку

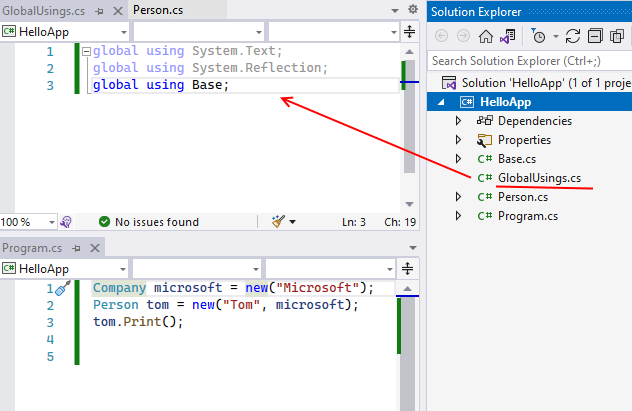


**Определение глобальных пространств в отдельном файле**

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:



И этот набор пространств имен будет автоматически подключаться во все файлы кода в проекте.

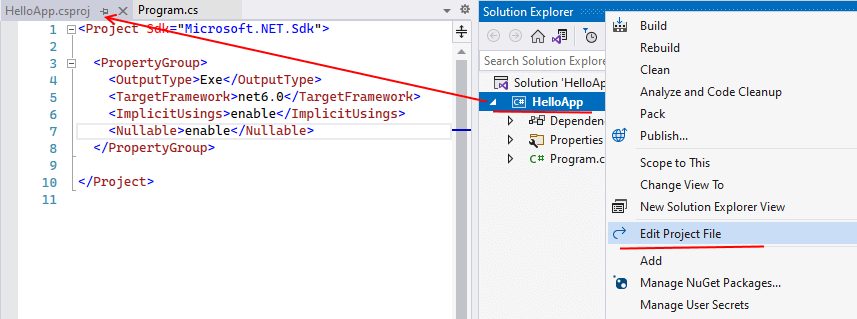


## **Подключение пространств имен по умолчанию**

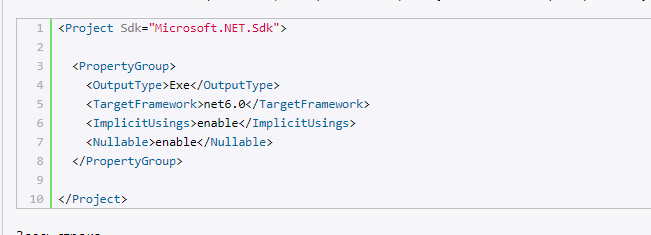
Все классы существуют в некотором пространстве имен, и чтобы эти классы использовать, необходимо подключить их пространства имен, либо использовать полное название класса с указанием его пространства имен. Однако начиная с Visual Studio 2022 и .NET 6 и C# 10 мы можем просто в файле программы написать:

|  |  |
| --- | --- |
| 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**

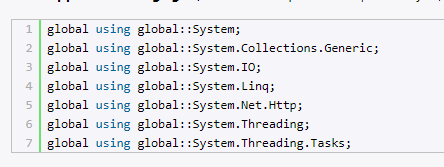


После этого Visual Studio откроет нам файл проекта, который будет выглядеть примерно следующим образом:Здесь строка

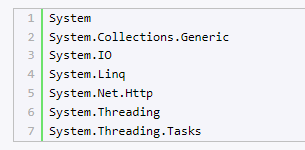


точнее элемент <ImplicitUsings> задает подключение некоторого набора пространств имен по умолчанию. Значение **enable** указывает, что эта настройка будет применяться.

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:



Здесь перечислены все те пространства имен, которые подключаются в проект по умолчанию:



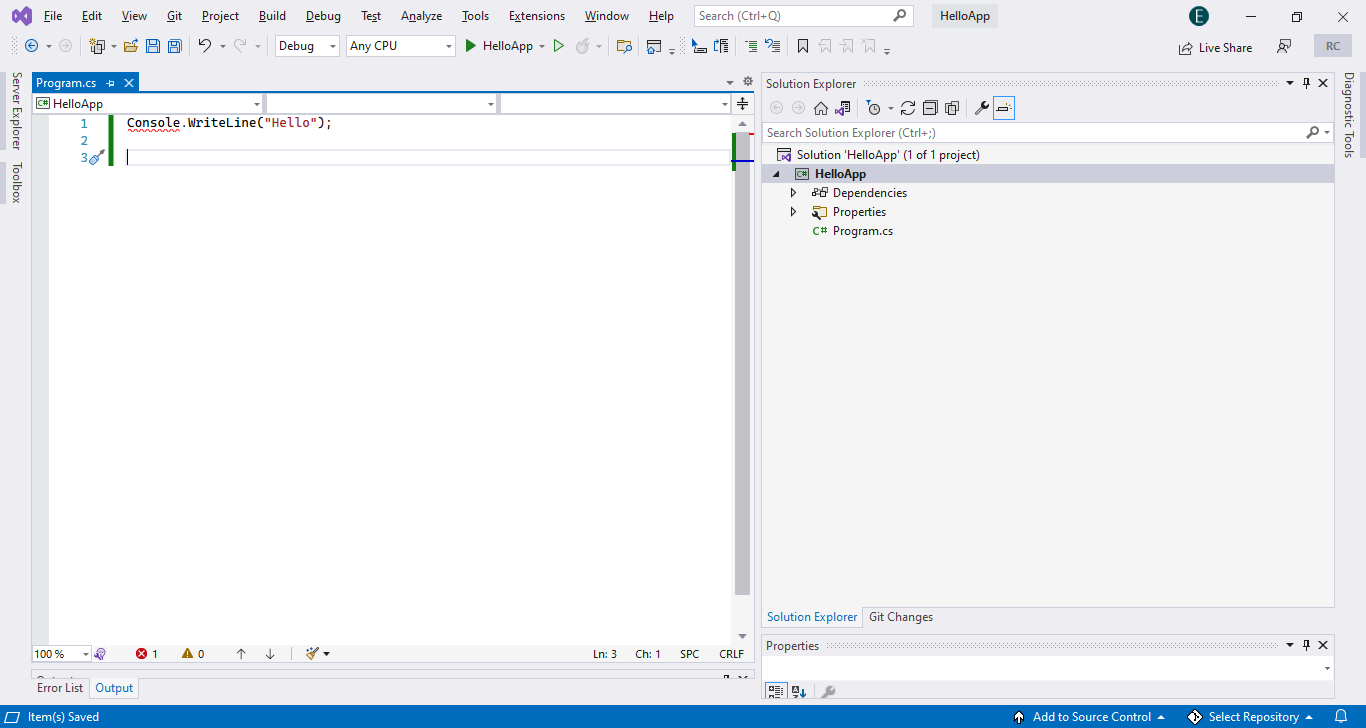
Стоит отметить, что это набор пространств имен, которые подключаются в проект консольного приложения по умолчанию, у других типов проектов набор подключаемых пространств может отличаться.

### **Отключение пространств по умолчанию**

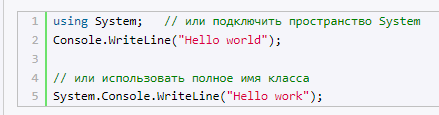
В проектах, создаваемых в Visual Studio 2022 для .NET 6 и C# 10 эта возможность включена по умолчанию. Тем не менее при необходимости мы ее можем отключить. Для этого изменим эту настройку следующим образом:



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:



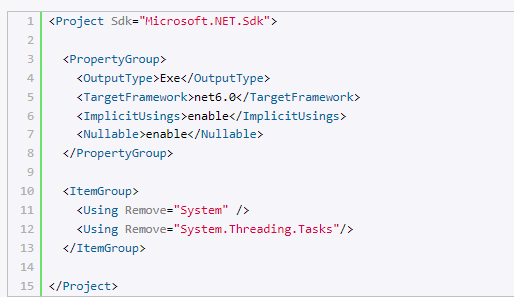
**Подключение и отключение пространств по отдельности**

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:Здесь атрибут **Include** элемента **Using** подключает пространства имен глобально по всему проекту. Например:



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**:



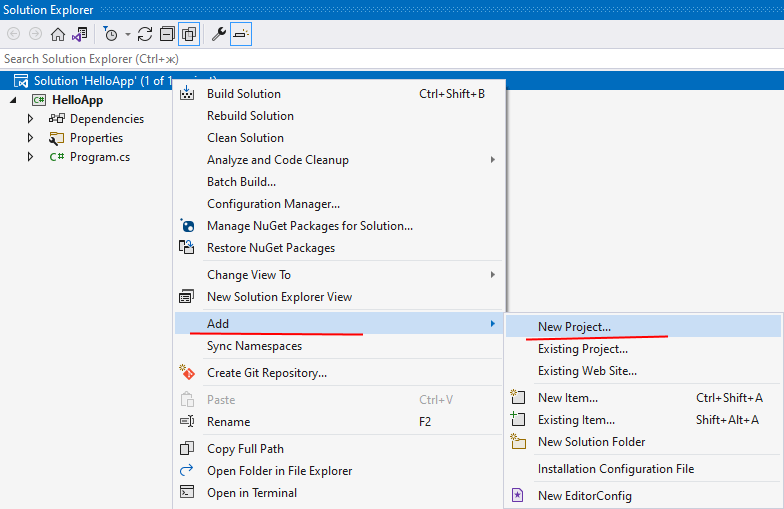
В данном случае отключаются пространства "System" и "System.Threading.Tasks"

## **Создание библиотеки классов в Visual Studio**

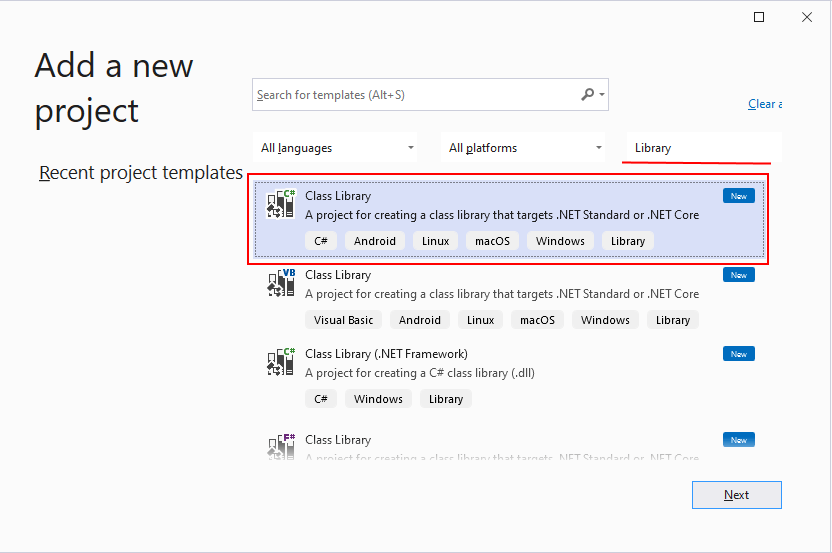
Нередко различные классы и структуры оформляются в виде отдельных библиотек, которые компилируются в файлы dll и затем могут подключаться в другие проекты. Благодаря этому мы можем определить один и тот же функционал в виде библиотеки классов и подключать в различные проекты или передавать на использование другим разработчикам.

Создадим и подключим библиотеку классов.

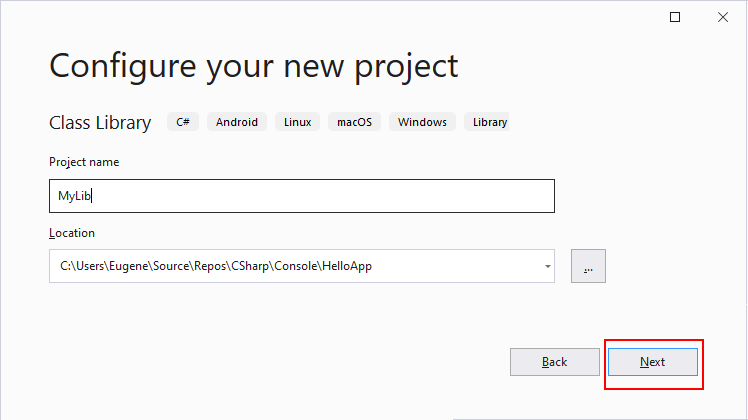
Возьмем имеющийся проект консольного приложения C#, например, созданный в прошлых темах. В структуре проекта нажмем правой кнопкой на название решения и далее в появившемся контекстном меню выберем **Add -> New Project...** (Добавить новый проект):



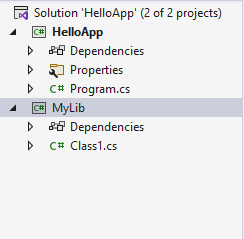
Далее в списке шаблонов проекта найдем пункт **Class Library**:



Затем дадим новому проекту какое-нибудь название, например, MyLib:

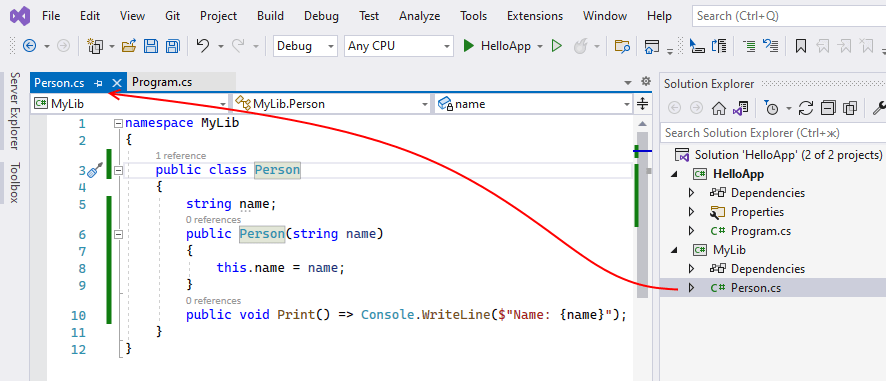
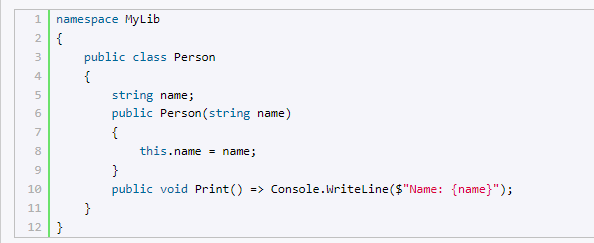


После создания этого проекта в решение будет добавлен новый проект, в моем случае с названием MyLib:

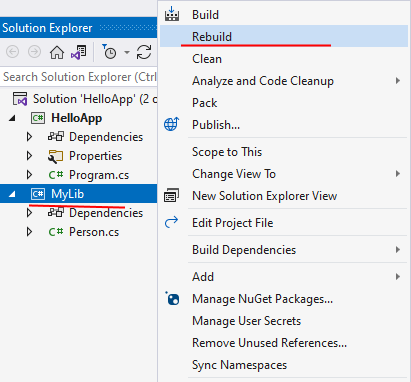


По умолчанию новый проект имеет один пустой класс Class1 в файле Class1.cs. Мы можем этот файл удалить или переименовать, как нам больше нравится.

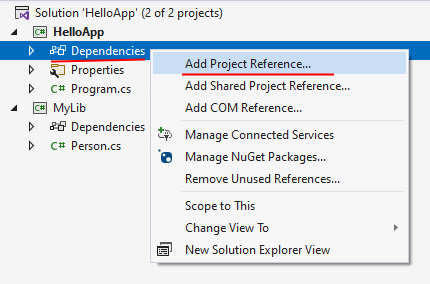
Например, переименуем файл Class1.cs в Person.cs, а класс Class1 в Person. Определим в классе Person простейший код:



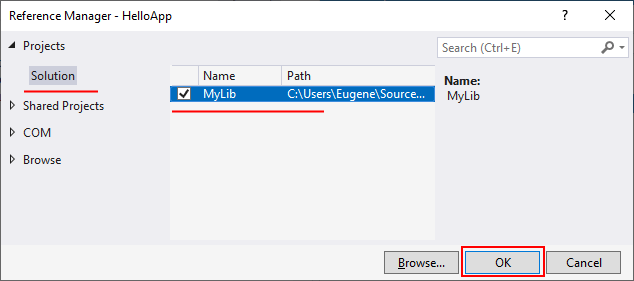
Теперь скомпилируем библиотеку классов. Для этого нажмем правой кнопкой на проект библиотеки классов и в контекстном меню выберем пункт **Rebuild**:



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...**:

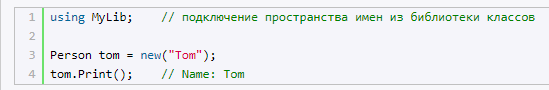


Далее нам откроется окно для добавления библиотек. В этом окне выберем пункт Solution,который позволяет увидеть все библиотеки классов из проектов текущего решения, поставим отметку рядом с нашей библиотекой и нажмем на кнопку OK:



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:



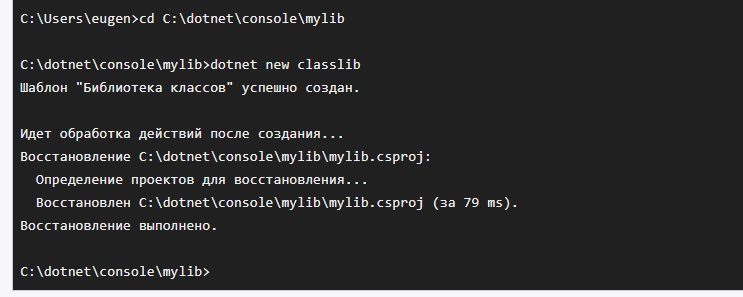
## **Создание библиотеки классов с помощью .NET CLI**

## В прошлой теме рассматривалось создание библиотеки классов в Visual Studio и ее подключение в другой проект решения. Рассмотрим, как сделать то же самое, только без Visual Studio, используя .NET CLI.

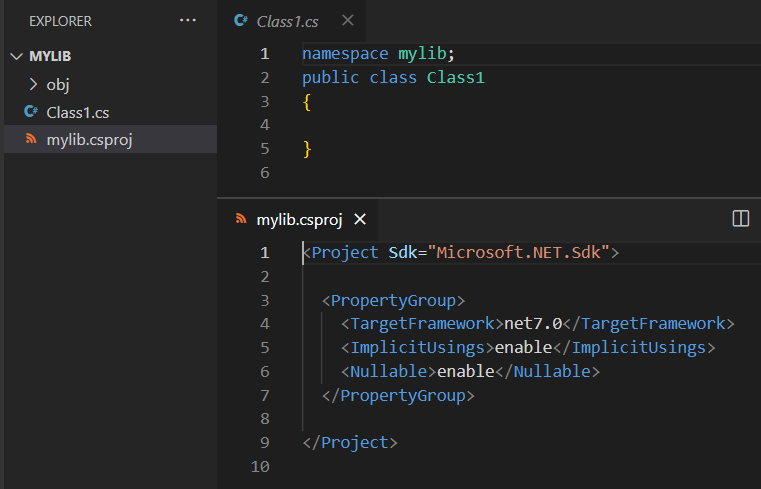
Прежде всего для создания проекта библиотеки классов в .NET CLI команде dotnet new надо передать шаблон **classlib**



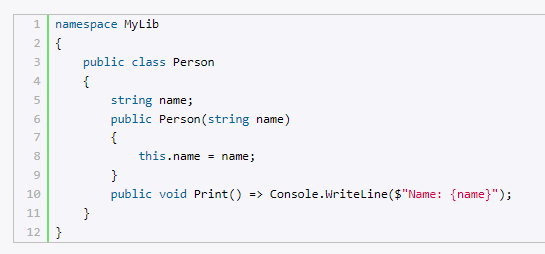
Например, пусть у нас проект библиотеки классов будет располагаться в папке **C:\dotnet\console\mylib**. Создадим проект:



После выполнения этой команды будет создан проект с файлом кода Class1.cs:



Это тот же проект, что создается в Visual Studio. Например, переименуем файл Class1.cs в Person.cs, а класс Class1 в Person. Определим в классе Person простейший код:

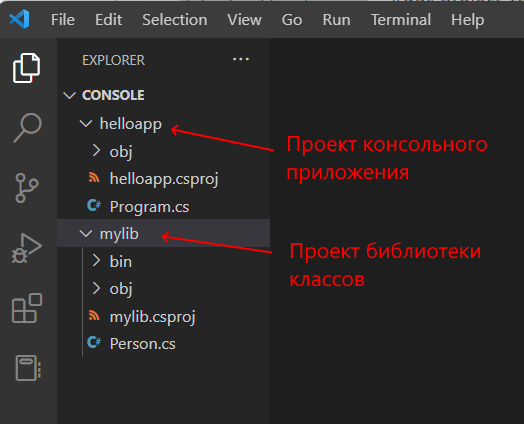


Для построения библиотеки классов выполним команду



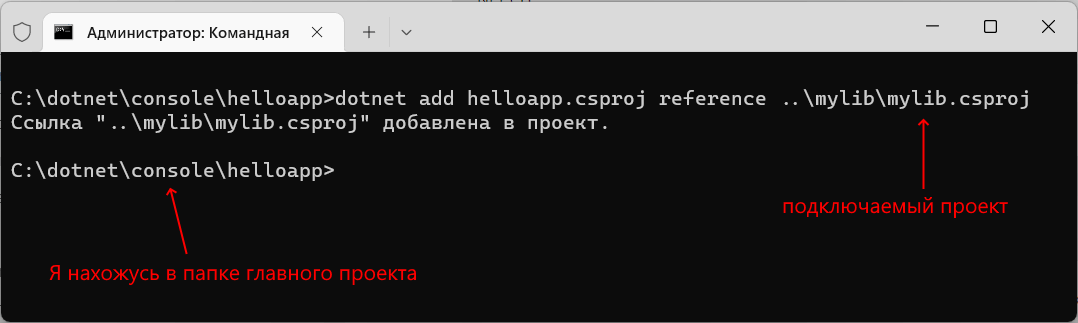
И после построения в проекте в папке \bin\Debug\net7.0\ можно будет найти файл mylib.dll.

Теперь подключим эту библиотеку в какой-нибудь проект. Для теста создадим новый консольный проект в папке C:\dotnet\console\helloapp (то есть в соседней папке по отношению к проекту mylib):

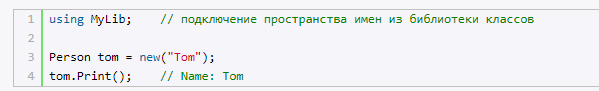


Для добавления ссылки одного проекта в другой применяется команда **dotnet add reference**, которая имеет следующий общий формат:

То есть в данном случае, если я нахожусь в папке проекта helloapp, то подключение библиотеки классов будет выглядеть следующим образом:



После успешного подключения библиотеки в главном проекте изменим файл **Program.cs**, чтобы он использовал класс Person из библиотеки классов:**Удаление ссылки на проект**



Если надо удалить ссылку, то аналогичным образом применяется команда**Создание решения**

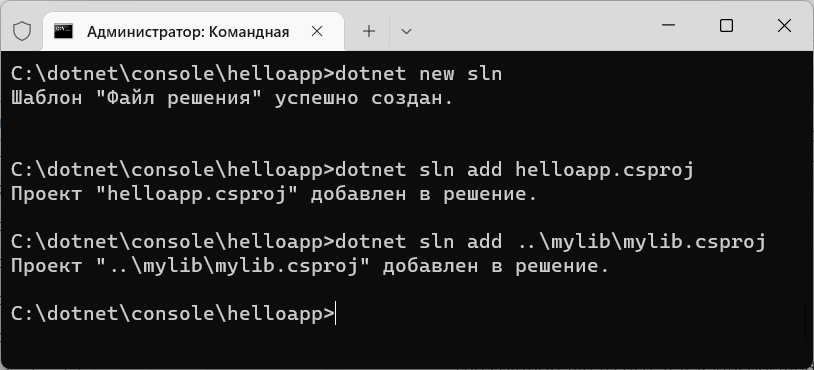
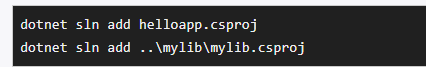


Для создания решения применяется команда **dotnet new sln**, которая по умолчанию создает файл решения в текущей папке. Так, перейдем в терминале в папку консольного проекта и выполним команду



После этого в текущей папке будет создан файл с расширением **.sln**, который по умолчанию называется по имени текущей папки.

Добавим ранее созданные проекты в решение. Для этого применяется команда **dotnet sln add**, которая имеет следующий обобщенный формат:Если файл решения расположен в текущей папке, то путь к файлу решения можно не указывать. Так, если мы по прежнему находимся в терминале в папке консольного проекта helloapp, выполним последовательно следующие команды



Если потом потребуется, наоборот, удалить проект из решения, то применяется команда



## **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:

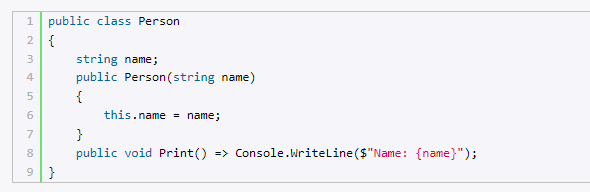
* **private**: закрытый или приватный компонент класса или структуры. Приватный компонент доступен только в рамках своего класса или структуры.
* **private protected**: компонент класса доступен из любого места в своем классе или в производных классах, которые определены в той же сборке.
* **file**: добавлен в версии C# 11 и применяется к типам, например, классам и структурам. Класс или структура с такми модификатором доступны только из текущего файла кода.
* **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.



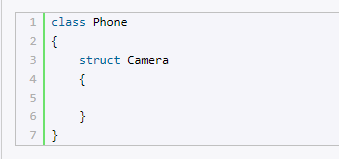
Стоит отметить, что эти модификаторы могут применяться как к компонентам класса, так и к компонентам структуры за тем исключением, что структуры не могут использовать модификаторы private protected, protected и protected internal, поскольку структуры не могут быть унаследованы.

Все классы и структуры, определенные напрямую вне других типов (классов и структур) могут иметь только модификаторы public, file или internal.

Мы можем явно задать модификатор доступа, а можем его и не указывать: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**.



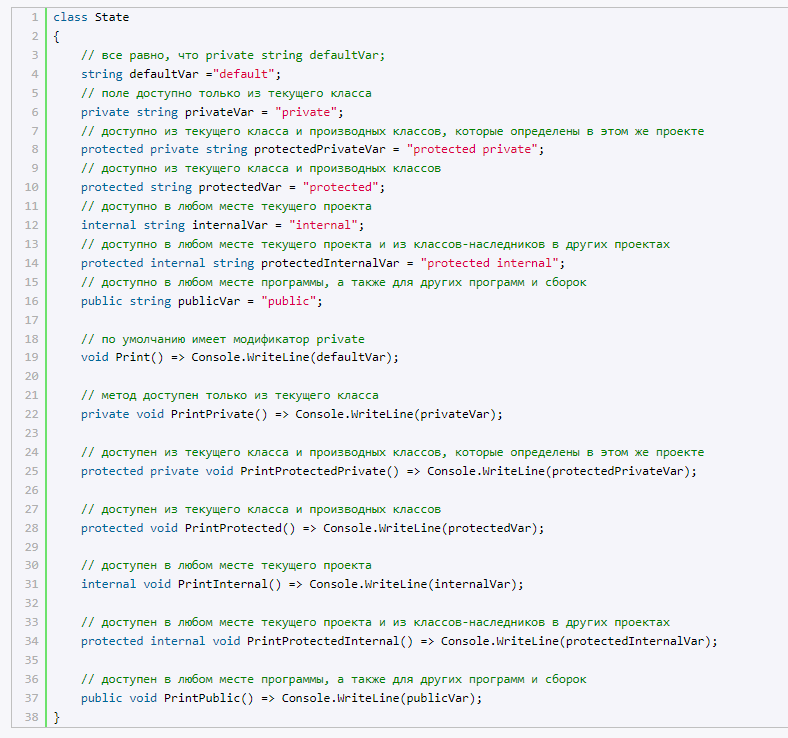
Классы и структуры, которые объявлены без модификатора и которые расположены вне других типов, по умолчанию имеют доступ **internal**, а вложенные классы и структуры, как и остальные компоненты классов/структур имеют модификатор **private**. Например:



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**

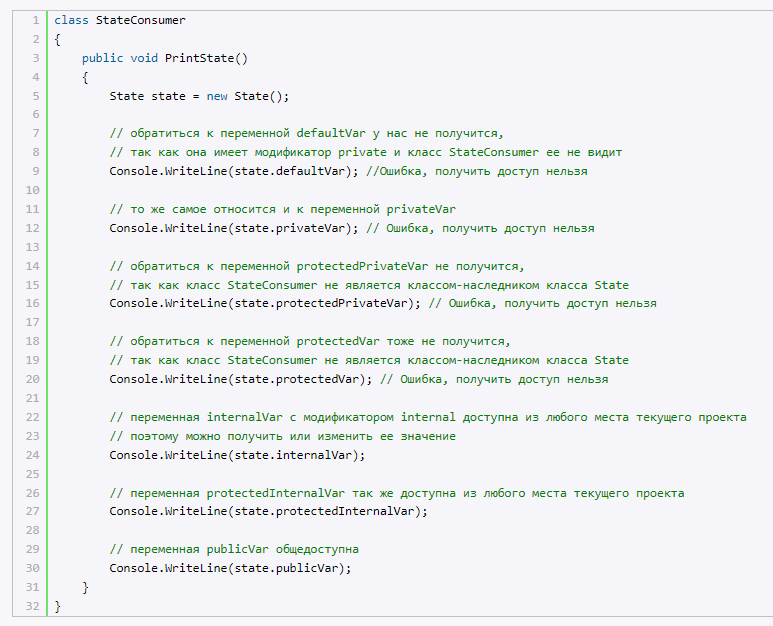
### **Модификаторы в рамках текущего проекта**

Посмотрим на примере и создадим следующий класс State: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**:Таким образом, в классе StateConsumer мы смогли только обратиться к переменным internalVar, protectedInternalVar и publicVar, так как их модификаторы позволяют использовать в данном контексте.

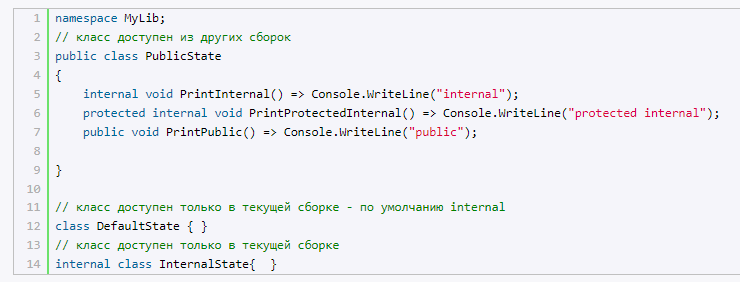


Аналогично дело обстоит и с методами:Здесь нам оказались доступны только три метода: PrintInternal, PrintProtectedInternal, PrintPublic, которые имееют соответственно модификаторы internal, protected internal, public.



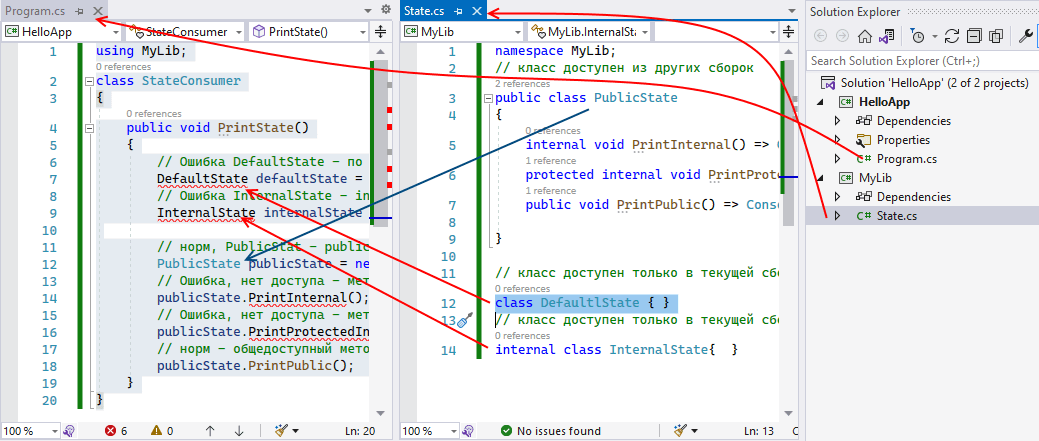
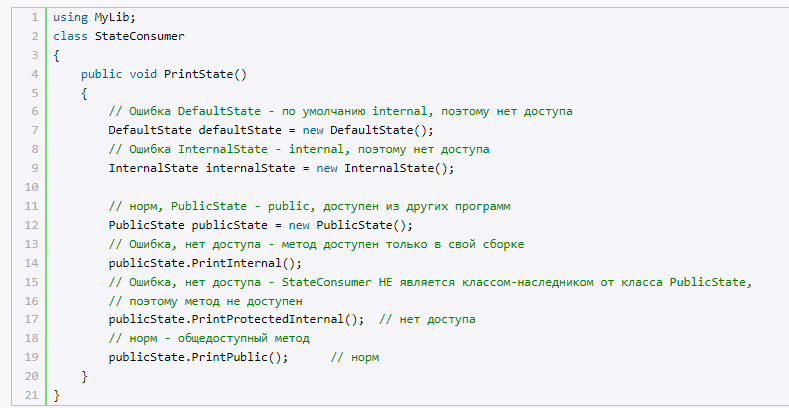
### **Модификаторы в рамках сборок**

Допустим, у нас есть проект (и соответственно сборка) MyLib, в которой определены три класса:Здесь классы DefaultState и InternalState имеют модификатор **internal**, поэтому доступны только в текущем проекте.



Класс PublicState модификатором **public** доступен из других проектов. Однако его метод PrintInternal() доступен только в текущем проекте. Вне текущего проекта доступен только его метод PrintPublic и PrintProtectedInternal() (доступен в другом проекте только в классах-наследниках).

Допустим, мы подключаем сборку этого проекта MyLib в другой проект, где есть класс StateConsumer:

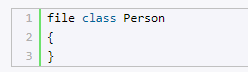


В классе StateConsumer есть доступ только к классу PublicState и его методу PrintPublic, потому что они имеют модификатор **public**. К остальной функциональности подключенной сборки StateConsumer доступа не имеет.

Благодаря такой системе модификаторов доступа можно скрывать некоторые моменты реализации класса от других частей программы.

### **Файл как область видимости**

C# 11 был добавлен еще один модификатор видимости - **file**, который применяется к классам, структурам, делегатам, перечислениям, интерфейсам. Типы с этим модификатором могут использоваться только внутри текущего файла кода.



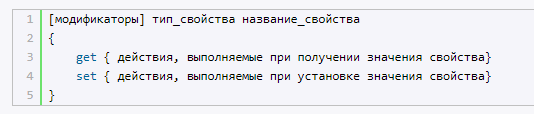
Данный модификатор не может использоваться в паре с другими модификаторами.

## **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:



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.

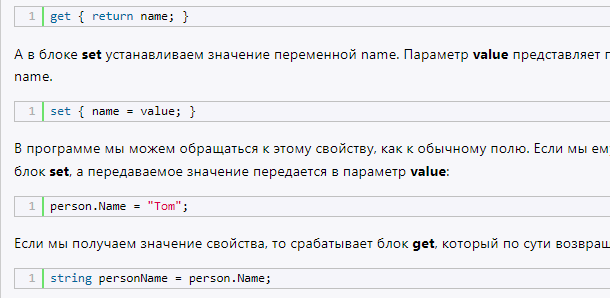
В блоке **set** устанавливается значение свойства. В этом блоке с помощью параметра **value** мы можем получить значение, которое передано свойству.

Блоки **get** и **set** еще называются акссесорами или методами доступа (к значению свойства), а также геттером и сеттером.

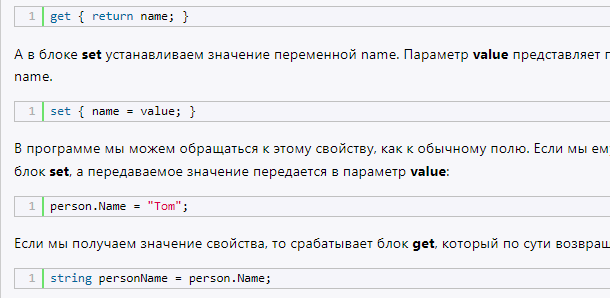
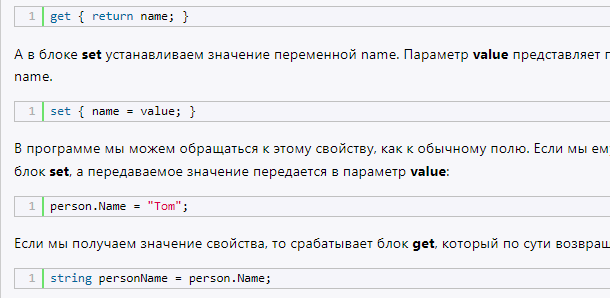
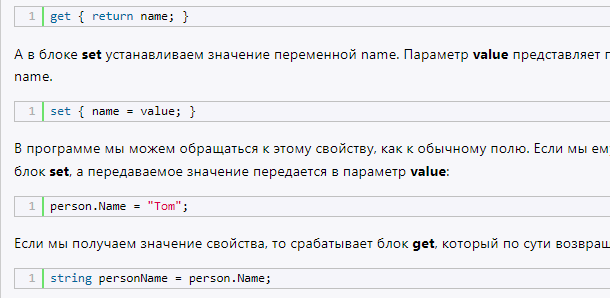
Рассмотрим пример:Здесь в классе Person определено приватное поле name, которая хранит имя пользователя, и есть общедоступное свойство Name. Хотя они имеют практически одинаковое название за исключением регистра, но это не более чем стиль, названия у них могут быть произвольные и не обязательно должны совпадать.



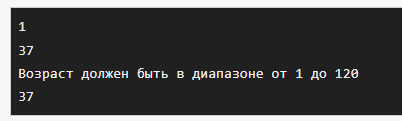
Через это свойство мы можем управлять доступом к переменной name. В свойстве в блоке **get** возвращаем значение поля:



А в блоке **set** устанавливаем значение переменной name. Параметр **value** представляет передаваемое значение, которое передается переменной name.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**:Если мы получаем значение свойства, то срабатывает блок **get**, который по сути возвращает значение переменной name:То есть по сути свойство Name ничего не хранит, оно выступает в роли посредника между внешним кодом и переменной 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: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:

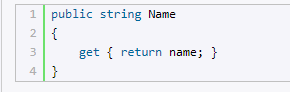


Таким образом, свойство позволяет опосредовать и контролировать доступ к данным объекта.

### **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.

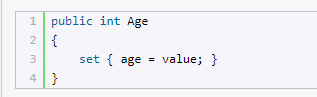
И, наоборот, если свойство имеет только блок **set**, тогда это свойство доступно только для записи - можно только установить значение, но нельзя получить:Здесь свойство Name доступно только для чтения, поскольку оно имеет только блок **get**:



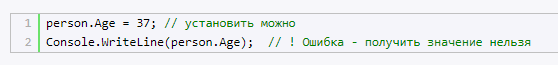
Мы можем получить его значение, но НЕ можем установить:



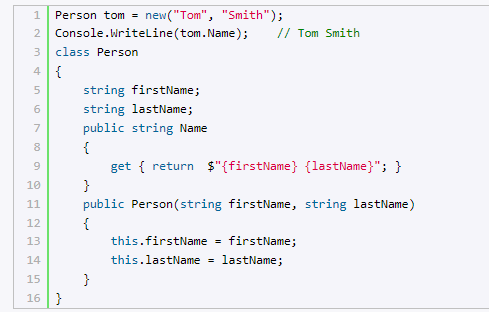
А свойство Age, наоборот, доступно только для записи, поскольку оно имеет только блок **set**:



Можно установить его значение, но нельзя получить:**Вычисляемые свойства**



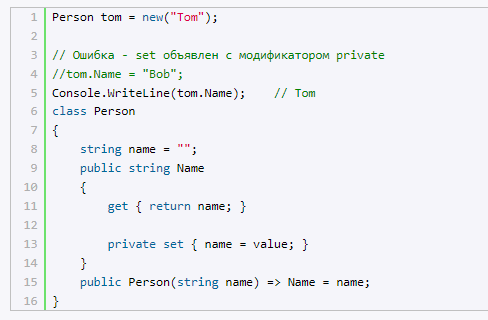
Свойства необязательно связаны с определенной переменной. Они могут вычисляться на основе различных выражений



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:



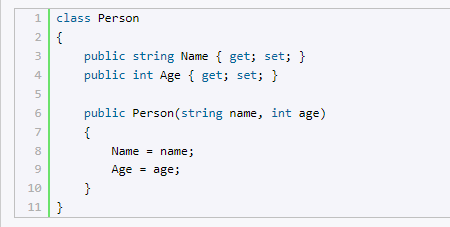
Теперь закрытый блок set мы сможем использовать только в данном классе - в его методах, свойствах, конструкторе, но никак не в другом классе:

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:

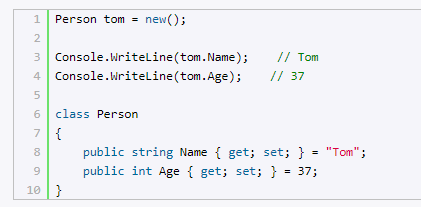


На самом деле тут также создаются поля для свойств, только их создает не программист в коде, а компилятор автоматически генерирует при компиляции.

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.

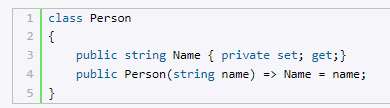
Стоит учитывать, что нельзя создать автоматическое свойство только для записи, как в случае со стандартными свойствами.

Автосвойствам можно присвоить значения по умолчанию (инициализация автосвойств):

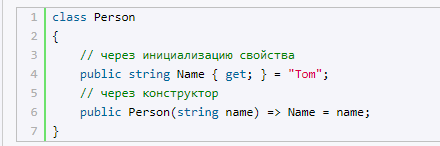


И если мы не укажем для объекта Person значения свойств Name и Age, то будут действовать значения по умолчанию.

Автосвойства также могут иметь модификаторы доступа:

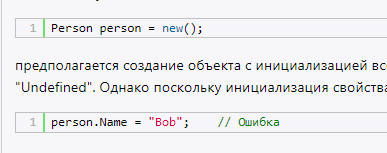
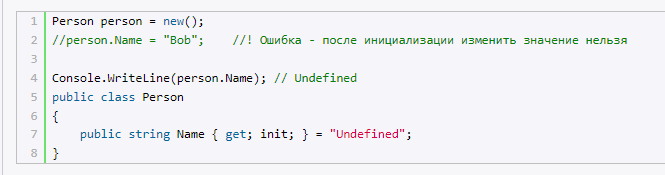


Мы можем убрать блок set и сделать автосвойство доступным только для чтения. В этом случае для хранения значения этого свойства для него неявно будет создаваться поле с модификатором readonly, поэтому следует учитывать, что подобные get-свойства можно установить либо из конструктора класса, как в примере выше, либо при инициализации свойства:

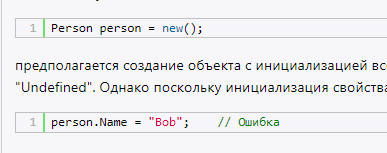


**Блок init**

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:В данном случае класс Person для свойства Name вместо сеттера использует оператор **init**. В итоге на строке

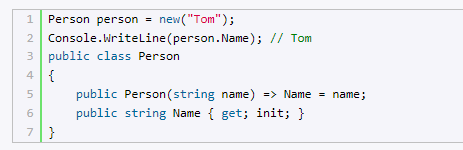


предполагается создание объекта с инициализацией всех его свойств. В данном случае свойство Name получит в качестве значения строку "Undefined". Однако поскольку инициализация свойства уже произошла, то на строке

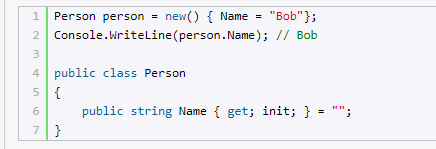


мы получим ошибку.

Как можно установить подобное свойство? Выше продемонстрирован один из способов - установка значения при определении свойства. Второй способ - через конструктор:



Третий способ - через инициализатор:



В принцпе есть еще четвертый способ - установка через другое свойство с модификатором **init**:

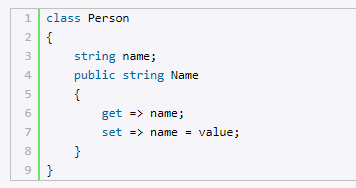


В данном случае свойство Name управляет полем для чтения name. Благодаря этому перед установкой значения свойства мы можем произвести некоторую предобработку. Кроме того, в выражении init устанавливается другое init-свойство - Email, которое для установки значения использует значение свойства Name - из имени получаем значение для электронного адреса.

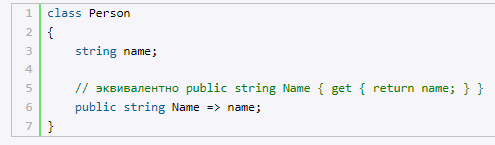
Причем если при объявлении свойства указано значение, то в конструкторе мы можем его изменить. Значение, установленное в конструкторе, можно изменить в инициализаторе. Однако дальше процесс инициализации заканчивается. И значение не может быть изменено.

### **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 **=>**:

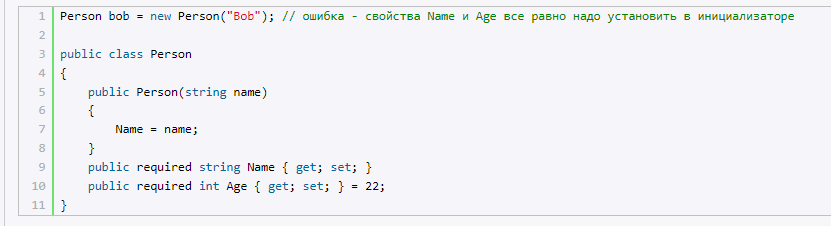
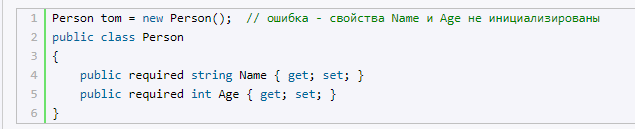


Также можно сокращать все свойство в целом:



**модификатор required**

Модификатор **required** (добавлен в C# 11) указывает, что поле или свойства с этим модификатором обязательно должны быть инициализированы. Например, в следующем примере мы получим ошибку:Здесь свойства Name и Age отмечены как обязательные для инициализации с помощью модификатора **required**, поэтому необходимо использовать инициализатор для их инициализации:Причем не важно, устанавливаем эти свойства в конструкторе или инициализируем при определении, все равно надо использовать инициализатор для установки их значений. Например, в следующем примере мы получим ошибку:



## **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:

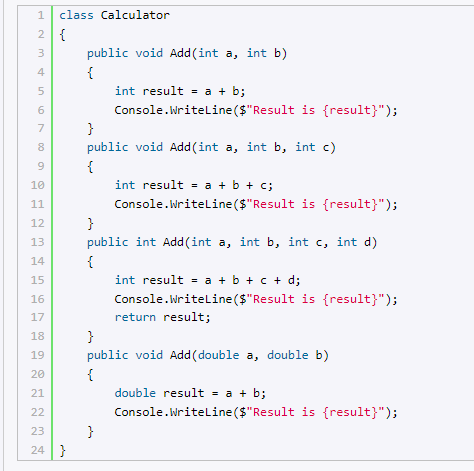


У данного метода сигнатура будет выглядеть так: Sum(int, int)

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

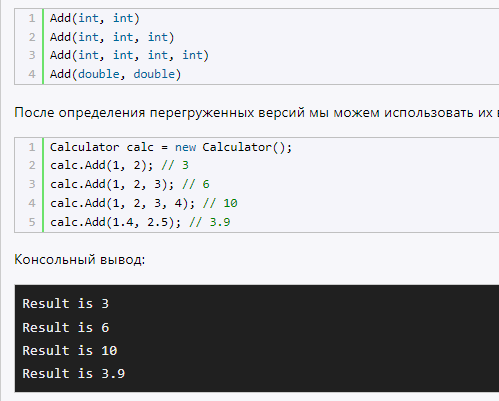
Например, пусть у нас есть следующий класс:



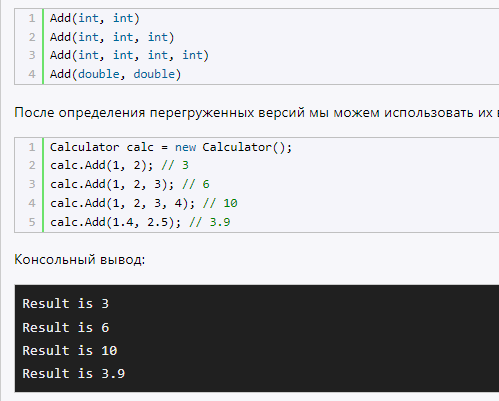
Здесь представлены четыре разных версии метода Add, то есть определены четыре перегрузки данного метода.

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.

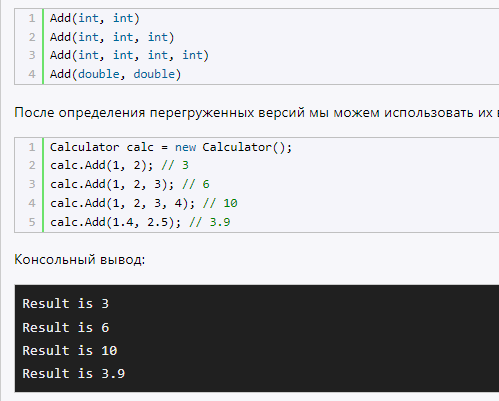
То есть мы можем представить сигнатуры данных методов следующим образом:



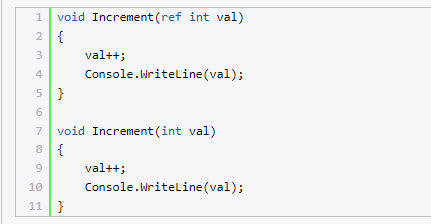
После определения перегруженных версий мы можем использовать их в программе:



Консольный вывод:

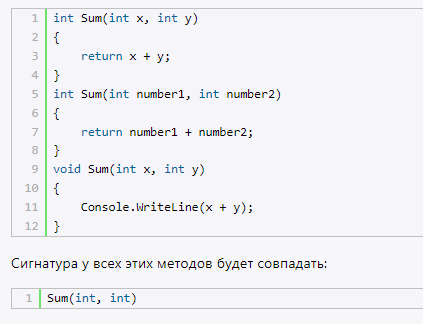


Также перегружаемые методы могут отличаться по используемым модификаторам. Например:

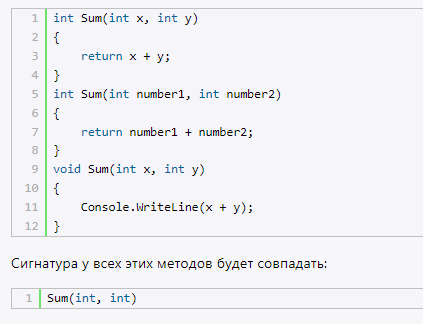


В данном случае обе версии метода Increment имеют одинаковый набор параметров одинакового типа, однако в первом случае параметр имеет модификатор ref. Поэтому обе версии метода будут корректными перегрузками метода Increment.

А отличие методов по возвращаемому типу или по имени параметров не является основанием для перегрузки. Например, возьмем следующий набор методов:



Сигнатура у всех этих методов будет совпадать:



Поэтому данный набор методов не представляет корректные перегрузки метода Sum и **работать не будет**.

# Лекции

## Бой матов

ООП

Методология программирования основанная на представлении программы в виде совокупности объектов. Каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

Наследование- концепция согласно которой одни классы называемые родительскими могут лежать на основе других дочерних. При этом дочернии классы перенимают свойства и поведения своего родителя.

Идеологически ооп подход к программированию как к моделированию информационных объектов. Решающий на новом уровне основную задачу структурного программирования: структурирование информации с точки зрения управляемости что существенно улучшает управляемость самим процессом моделирования что в свою очередь особенно важно при реализации крупных объектов.

(Структура в ооп - составной тип данных который может содержать в себе несколько переменных членов)

Структуры и строй блоки в ооп включают: 1. Классы - опред пользователем типы данных. Который действует как шаблон для отдельных атрибутов и методов. (Объект экземпляр класса созданный с использованием спец опред данных. )

Управляемость для иерархичности предполагает минимизацию избыточности данных. (Аналогично нормализации ) и их целостности. При этом созданным удобно управляемый будет удобно управляться. Таким образом через тактичичную задачу управляемости решается стратегическая задача транслировать понимание задачи программистом наиболее удобно для дальнейшего удобства.

Основные

Связаны с различными аспектами базового понимания предметной задачи которой требуется для оптимального управления соответствующими объектами:

1. Абстракция для выделения моделирования важного для решения конкретной задачи по предмету в конечном счете контекстное понимание объектов формализованный в виде класса
2. Инкапсуляция для быстрой и безопасной организации собственной иерархической управляемости: что бы было достаточно понятной команды что делать без одновременного уточнения как делать. Так как это уже другой уровень управления
3. Наследование для быстрой и безопасной организации родственных понятий. Что бы было достаточно на каждом иерархическом шаге только изменения не дублируя все остальное.
4. Полиморфизм для определения точки которые единое управление лучше распоралелить или запаралелить ( собрать в едино)

То есть реч идет о прогрессирующей организации информации согласно первичным сигматический критериям:

Важное / неважное ключевое подробности родительское дочернее,

Единое множественное.

Программирование в частности на последнем этапе дает возможность перевода на следующий уровень детализации что замыкает общий процесс.

## Бойматов лекция 2. Событийно управляемое программирование. Основные принципы соп

Соп - от англ event - driving programming - парадигма программирования в которой выполнение программ выполнение программ определяется событием - действиями пользователя( клава, мышь, сенсор экрана ) сообщениями других программных потоков, событиями операционной системы. пример : поступление сетевого пакета.

Соп можно также можно определить как способ построения компьютерной программы. При котором в ходе ( как правило головной функции программы ) явным образом выделяется главный цикл приложений тело которого состоит из двух частей: выборки события и обработки события. Как правило в реальной программе не допустимым длительное использование обработчика события поскольку при этом программа не может реагировать на другие события.

В связи с этим при написании соп часто применяют автоматное программирование.

Под событиями в яп обычно понимается способ внедрения того или иного фрагмента программного кода с целью изменения поведения программы. Как только происходит изменение среды вычисления из числа представляющих интерес для разработчика или пользователя ПО активизируется событие и выполняется соответствующий фрагмент кода.

В целом с точки зрения практического программирования обработка события подобна вызову процедур. Причем в роли параметра выступает те или иные характеристики среды вычисления.

Любой современный интерфейс пользователя (среда вычисления) построена на основе обработки событий ( on Clic On Mouse move MouseOver ) событие осуществляет взаимодействие локальных сетей, ос , сторонними приложениями и т.д. Могут также активизироваться по времени. Соответствие со схемой двух уровневой концептуализации первый уровень может означать например инициализацией пользователем события щелчек правой кнопкой мыше а второй изменения состояния объекта в меню при выборе соответсвующего пункта меню.

Как видим возможный индивид становится действительным ( активация события) а затем осуществляется подначивание объекта. Изменяеться текущие позиция в меню.

Фрагментом кода программы в таком случае является метод изменяющий текущую позицию в меню. Которая активизируется исходя из значений первого соотнесения. Таким образом на основе механизма события осуществляется управление программ.

Элементы управления

Почти все современные графические интерфейсы общего назначения строиться по модели wimp(Window Iсon menu point) ( окно иконка меню указатель) внутри окон рисуются элементы графического интерфейса, Которые для краткости будут называться виджетами (widget) (штучка). Меню могут располагаться в различных частях окна но их поведение достаточно однотипно. Они служат для выбора действия из набора пред определенных действий. Программа реализующая графический интерфейс событийно ориентирована. Она ждет от интерфейса событий которые обрабатывают и обрабатывается сообразно согласно своему внутреннему состояния. Эти события возникают в элементах графического интерфейса виджета и обрабатывается прикрепленного к этим виджетам обработчикам. Сами виджеты имеют многочисленные свойства ( цвет, размер, расположение). Выстраиваются в иерархию принадлежности. 1 виджет может быть хозяином другого,Имеет методы для доступа к своему состоянию.

Расположение виджетов внутри других виджетов ведует менеджеры расположения. Виджет устанавливается на место по правилам менеджера расположения. В vs имеется три менеджера расположения

1. Простой упоковщик pack
2. Серп greet
3. И произвольное расположение

Рассмотрим некоторые классы виджеты библиотеки informs

Button- кнопка для вызова некоторых действий.

Canvas( рисунок) основа для вывода графических примитивов

Check button ( флажок) кнопка которая умеет переключаться между двумя состояниями при ножатии на неё

Поле ввода поле в которое можно внести строку текста.

Freim( рамка) - виджет которой содержит в себе другие визуальные компоненты.

Label ( надпись) виджет может показывать текст или изображение.

ListBox( список) прямоугольная рамка со списком из которого пользователь один или несколько элементов.

Menu элемент с помощью которого можно создавать всплывающие и ниспадающие меню.

RadioButton селекторная кнопка - кнопка для предоставления одного из альтернативных значений такие кнопки как правило действует в группе

Scroll bar полоса прокрутки для отображения величины прокрутки в других виджетах.

Text форматированный текст этот прямоугольный виджет позволяет редактировать и форматировать текст с использованием различных стилей.

Top label окно верхнего уровня показывается как отдельное окно и содержит другие виджеты.

Структурное программирование- парадигма программирование которая подчеркивает использование четко определенных структур для организации и представления программного кода.

Основная структурного программирования сделать программу более понятными надежными легкими в сопровождении. Основные принципы структурного программирования

1. Модульность: деление программы на независимые модули каждый из которых выполняет определенную функцию.
2. Иерархия: организация модулей в иерархическую структуру где каждый модуль имеет определенный уровень абстракции.
3. Локализация: концентрация данных и функций связанных с конкретной задачей в одном месте.
4. Стандартизация: использование стандартизированных конструкций и соглашений для написания кода.

Структурные элементы программы.

1. Модули: независимые части программы которые выполняют определенную функцию
2. Функции: подпрограммы которые выполняют определеную задачу и возвращается результат
3. Цикл: конструкция которая позволяет повторять выполнения определенного кода.
4. Условные операторы: конструкции которые позволяют выполнить определеный код в зависимости от условий

Преимущества структурного программирования:

1. Улучшенная читаемость: программа написанная в структурном стиле легче понять и сопровождать.
2. Повышенная надежность: структурное программирование помогает избежать ошибок и исключений.
3. Легче модифицировать: программа написанная в структурном стиле легче модифицировать и расширять.
4. Легче повторно использовать: структурное повторение позволяет повторно использовать код в других программах.

ТЗ: банковская система- создание системы для создания пополнения и опустошения банковских счета и проверять их баланс.

Класс , создание класса BankAccount

public class BankAccount

{

private decimal balance;

public BankAccount(decimal initialBalance)

{

balance = initialBalance;

}

public void Deposit(decimal amount)

{

balance += amount;

}

public void Withdraw(decimal amount)

{

if (amount > balance)

{

throw new InvalidOperationException("Недостаточно средств на счете");

}

balance -= amount;

}

public decimal GetBalance()

{

return balance;

}

}

Класс BankSystem

public class BankSystem

{

private List<BankAccount> accounts = new List<BankAccount>();

public void CreateAccount(decimal initialBalance)

{

var account = new BankAccount(initialBalance);

accounts.Add(account);

}

public void Deposit(int accountIndex, decimal amount)

{

if (accountIndex < 0 || accountIndex >= accounts.Count)

{

throw new IndexOutOfRangeException("Счет не найден");

}

accounts[accountIndex].Deposit(amount);

}

public void Withdraw(int accountIndex, decimal amount)

{

if (accountIndex < 0 || accountIndex >= accounts.Count)

{

throw new IndexOutOfRangeException("Счет не найден");

}

accounts[accountIndex].Withdraw(amount);

}

public decimal GetBalance(int accountIndex)

{

if (accountIndex < 0 || accountIndex >= accounts.Count)

{

throw new IndexOutOfRangeException("Счет не найден");

}

return accounts[accountIndex].GetBalance();

}

}

Пример использования

class Program

{

static void Main(string[] args)

{

var bankSystem = new BankSystem();

bankSystem.CreateAccount(1000);

bankSystem.Deposit(0, 500);

Console.WriteLine($"Баланс: {bankSystem.GetBalance(0)}"); // 1500

bankSystem.Withdraw(0, 200);

Console.WriteLine($"Баланс: {bankSystem.GetBalance(0)}"); // 1300

}

}

В этом примере мы создали простую банковскую систему которая демонстрирует принципы структурного программирования. Мы использовали классы и структуры данных для решения задачи. Мы также использовали методы и свойства для взаимодействия между классами.

Отличие структурного программирования от ооп

Два различных подхода к разработке по

Структурное программирование подходит для небольших программ с простой логикой в то время как ооп подходит для более сложных программ с множеством взаимодействующих элементов выбор подхода завит от конкретных потребностей проекта.

ООП- объекты классы наследование полиморфизм

СТ- модульность иерархия поток управление.

## Бойматов 3 основы паттернов проектирования. Введение.

Паттерн представляет собой определенный способ построения программного кода для решения часто встречающихся проблем. В данном случае предполагается что есть набор общих формализованных проблем которые довольно часто встречаются и паттерн предоставляет ряд принципов для решения этих проблем. Хотя идея паттерна как способов решения распоранненых проблем в области проектирования появилась довольно давно но их популярность стала расти благодаря известной работе 4 авторов. Ричард хилман Ральф Дженсона джона Блиесиса Эрик гармон которая называлась: "приемы объектно ориентированного проектирования, паттерны проектирования" Которая вышла в свет в 1994 году. А сам коллектив авторов не редко называют бандой 4х. Данная книга является первым масштабной попыткой описать распостранненые способы проектирования программ. И со временем применение паттернов стала считаться хорошей практикой в программировании.

Что нам дает применение паттернов :

При написании программ мы можем мы можем формализировать проблему в виде классов, объектов и связи между ними. И приминить один из существующих паттернов для её решения. Нам нечего придумать у нас есть готовый шаблон. Нам надо применить её в конкретной программе.

Причем паттерны не зависит от ЯП. Их принципы программирования будут аналогичны и в c# и в Java и в других языках. Также мышление паттерными упрощает групповую разработку программ. Зная применяемый паттерн проектирования, его основные принципы другому программисту будет проще понять его реализацию и использовать её. В тоже время не стоит применять паттерн ради самих паттернов. Хорошая программа предполагает использование паттернов однако не всегда паттерны упрощает и улучшает программу. Неоправдонное их использование может привести к усложнению и уменьшение качества кода. Паттерн должен быть оправдано эффективным способом решения проблем. Существует множество различных паттернов которые решают разные проблемы и выполняют различные задачи. Но по своему действию их можно объединить в ряд групп.

Рассмотрим некоторые группы паттернов. Основы классификации основных паттернов положенны цель или задача которые определенный паттерн выполняет.

1. Порождающий паттерны- паттерны которые абстрактируют процесс инсцанцирования/ процесс порождения классов и объектов. Среди них выделяются следующие - абстрактная фабрика. - строитель. - фабричный метод. - прототип. - одиночка.
2. Структурные паттерны- расмотривает как классы и объекты образуют более крупные структуры. Более сложные по характеру классы и объекты: - адаптер. - мост. - компоновщик. - декоратор. - фасад. - приспособлениц. - заместитель.
3. Поведенчиские- паттерны определяющие алгоритмы взаимодействия между классами и объектами. - цепочка обязоностей. - команда. - интерпретатор. - итератор. -посредник. - хранитель. - наблюдатель. - состояние. - шаблонный метод. - стратегия. - посетитель.

Существуют и другие классификации паттернов относиться к классам или объектам. Паттерны классов описывают отношение между классами помредстаом наследования. Отношение между классами определяются на стадии кампиляции. К таким паттернам относятся: фабричный метод, интерпритатор, шаблонный метод , адаптер.

Другая часть паттернов объектов. Описывают отношение между объектами. Эти отношения возникают на этапе выполнения поэтому обладают большой гибкостью:

1. Абстакторная фабрика
2. Строитель
3. Прототип
4. Одиночка
5. Мост
6. Компановщик
7. Декоратор
8. Фасад
9. Приспособлениц
10. Заместитель
11. Цепочка обязоностей
12. Команда
13. Итератор
14. Посредник
15. Хранитель
16. Наблюдатель
17. Состояние
18. Стратегия
19. Посетитель.

И это только некоторые основные паттерны. А вообще из намного больше.

Одни из них только начинают применяться другие уже менее распространены чем раньше.

Как выбрать нужный патерн

Прежде при решение какой-нибудь проблемы нужно выделить все используемые сущности и связи между ними и абстрагировать их от конкретных ситуаций. Затем надо посмотреть вписываешься абстрактная форма решения задачи в определенный патерн.

Причем не сразу взять первый попавшиеся патерн который показался нужным а посмотреть на несколько родственных паттернов из одной группы который решает одну и туже задачу. При этом важно понимать смысл и назначение паттернов. Явно представлять его абстрактную организацию и его реализацию.

Один патерн может иметь различные реализации и чем чаще вы сталкиваетесь с этими реализациями тем лучше вы будете понимать смысл паттерна. Но не стоит использовать патерн если вы его не понимаете, даже если на первый взгляд он поможет в решении задачи. В конечном счете надо придерживаться принципа Kiss ( keep it simple stupid). Ведь смысл паттернов не усложнение кода программа , на оборот его упрощение.

Отношение между классами и объектами.

Прежде чем приступить к изучению основных паттернов рассмотрим основные отношения между объектами которые помогут помогут нам понять связи между сущностями при их использовании в паттернах. Мы можем выделить несколько основных отношений

1. Наследование- является базовым принципом ооп и позволяет одному классу наследнику унаследывать функционал другого класса ( родительского). Нередко отношение наследование еще называют генерализацией и обобщением. Определяет отношение из a ( является )

class User

{

public int Id { get; set; }

public string Name { get; set; }

}

class Manager : User

{

public string Company{ get; set; }

}

В данном случае используется наследование а объекты класса менеджер являются объектами класса юзер. С помощью диаграм UML. Отношение между классами выражается в незакрашенной стрелочке к классу родитель.

￼

1. Реализация- предполагает определение интерфейса и его реализация в классах. На припер имееться интерфейс Iammuvable с методом move который реализуется в классе k.

public interface IMovable

{

void Move();

}

public class Car : IMovable

{

public void Move()

{

Console.WriteLine("Машина едет");

}

}

С помощью диаграммы uml отношение реализации также выражается в незакрашенной стрелкой только пунктирной.

￼

1. Ассоциация- отношение при котором объекты одного типа неким образом связаны с объектам другого типа. Например объект одного типа содержит и использует объект другого типа. Например игрок играет в определенной команде.

class Team

{

}

class Player

{

public Team Team { get; set; }

}

Класс плеер связан асоцией с классом Тим. На классах uml обозначается в виде обычной стрелки.

￼

Не редко при ассоциации указывается кратность связи. В данном случае единица у Тим и звездочка у плеер отражает связь один ко многим. То есть одна команда будет будет соответствовать многим игрокам. Агрегация и композиция являются частными случаями ассоциациями.

1. Композиция - определяет отношение has a (отношение имеет). Например в класс автомобиля содержит класс электро двигателя.

public class ElectricEngine

{

}

public class Car

{

ElectricEngine engine;

public Car()

{

engine = new ElectricEngine();

}

}

При этом класс автомобиля полностью управляет жизненным циклом класса

Пр уничтожении объекта автомобиля в области памяти вместе с ним будет уничтожен объект двигателя. В этом плане объекты автомобиля главные объект двигателя зависимы.

На диограме uml обозначается в обычной стрелке от главной сущности к зависимой при этом со стороны главной сущности которой содержит объект второй сущьности распологаеться закрашенный ромбик.

￼

1. Агрегация - от композиции следует отличать агрегацию. Она также предполагает отношение has a но реализуется она иначе.

public abstract class Engine

{ }

public class Car

{

Engine engine;

public Car(Engine eng)

{

engine = eng;

}

}

Объекты к и инжаир будут равноправны. В конструктор к передаеться ссылка на уже имеющийся объект инжаер и ссылка на не конкретный класс а на абстрактный класс или интерфейс. Отношение агрегации на диограмах uml отображается также как и отношение композиции только ромбик будет не закрашен.

￼

При проектировании отношения между классами надо учитывать некоторые общие рекомендации в место наследования следует предпочитать композицию. При наследовании весь функционал класса наследника жестко определен на этапе компиляции. И во время выполнения программы мы не можем его динамически переопределять. А класс наследник не всегда модет переопределять код. Который определен в родительском классе. Композиция позволяет динамически определять поведение объекта во время выполнения и по этому является более гибко. Вместо композиции следует предпочитать агригацию как более гибкий способ связи компонентов. В тоже время не всегда агрегация уместна например у нас есть класс человека который содержит объект нервной системы понятно что в реальности на текущий момент невозможно во вне определить нервную систему и внедрить её в человека. В данном случае человек будет главным компонентом а нервная система зависимым. Подчиненым и их создание и жизненный цик будет происходить совместно. По этому лучше выбрать композицию.

## Бойматов 7 октября Интерфейсы и абстрактные классы

Один из принципов проектирования гласит что при создании системы классов надо программировать на уровне интерфейсов а не их конкретно реализаций. Под интерфейсами в данном случае понимаются не только типы C# определенно при помощь ключевого слова interference а определение функционала без его конкретной реализации. То есть под данное определение попадает как собственно интерфейсы так и абстрактные классы. Которые могут иметь абстрактные методы без конкретной реалиазии. В этом плане у абстрактный классов и интерфейсов много общего нередко при проектировании программ паттернов мы можем заменять абстрактные классы на интерфейсы или наоборот. Однако же все они имеют некоторые различия.

Когда следует использовать абстрактные классы:

* Если надо определить общий функционал для родственных объектов
* Если мы проектируем довольно большую функциональную единицу которая содержит много базовых функционалов.
* Если нужно чтобы все производные класса на всех уровнях наследования имели некую общую реализацию.

При использовании абстрактных классов мы

* Если надо будет поменять название или параметр метода интерфейса то придется вносить изменения и также во всех классах который данный интерфейс реализует.

Когда следует использовать интерфейс

* если нам надо определить функционал для группы разрозненных объектов которые могут быть не как ни связаны между собой.
* Если мы проектируем небольшой функциональный тип.

Ключевым здесь является первые пункты которые можно свести к следующиму принципу: если классы относиться к единой системе классификации то выбирается абстрактные коассы иначе выбирается интерфейс

Допустим если у нас система транспортных средств: легковых автомобилей, автобус, трамвай, поезд и так далее. Поскольку данные объекты являются родственными мы можем выделить у них общие признаки то в данном случае можно использовать абстрактные классы.

public abstract class Vehicle

{

public abstract void Move();

}

public class Car : Vehicle

{

public override void Move()

{

Console.WriteLine("Машина едет");

}

}

public class Bus : Vehicle

{

public override void Move()

{

Console.WriteLine("Автобус едет");

}

}

public class Tram : Vehicle

{

public override void Move()

{

Console.WriteLine("Трамвай едет");

}

}

Абстрактный класс Vehicle определяет абстрактный метод перемещения Move. А классы наследники его реализуют. Но предположим наша система транспорта не ограничивается выше перечисли нами транспортными средствами пример самолет лодки возможна лошадь животное которое также может быть транспортом. Также можно добавить дирижабль. В общем получаеться довольно широкий круг объектов которые связаны только тем что являются транспортным средством и должны реализовать некоторый метод Мove. Так как объекты мало связаны между собой то для определения общего функционала лучше определить интерфейс.

Тем более некоторые из этих объектов могут существовать параллельных системах класификаци. Лошадь может быть классом в структурной системе классов животного мира. Возможная реализация интерфейса могла выглядеть след образом.

public interface IMovable

{

void Move();

}

public abstract class Vehicle : IMovable

{

public abstract void Move();

}

public class Car : Vehicle

{

public override void Move() => Console.WriteLine("Машина едет");

}

public class Bus : Vehicle

{

public override void Move() => Console.WriteLine("Автобус едет");

}

public class Hourse : IMovable

{

public void Move() => Console.WriteLine("Лошадь скачет");

}

public class Aircraft : IMovable

{

public void Move() => Console.WriteLine("Самолет летит");

}

Теперь метод Move определяется в интерфейсе IMovable а конкретный классы его реализует.

Говоря об использовании абстрактных классов и интерфейсов можно привести аналогию как состояние и действие.

Как правило абстрактные классы фокусируются на общем состоянии классов наследников. В то время как интерфейсы строиться вокруг одного общего действия.

Пример солнце, костер , батарея отопления выполняют функцию нагревания или излучения тепла. По большому счету выделение тепла это единственный общий между ними признак. Можно ли для них общий абстрактный класс. Можно но это не будет оптимальным решением. Тем более у нас могут быть родственные сущности которые мы возможно тоже захотим использовать. Поэтому для каждой выше перечисленной сущности мы можем определить свою систему класификации. Например в одной системе классов которые наследуются от общего абстрактного класса были бы звезды в том числе солнце планеты астероида то есть все те объекты которые могут иметь которое общее с солнцем состояние. В рамках другой системы классов мы могли бы мы могли определить электро приборы в том числе электоро нагреватель и так для разноплановой сущности можно было бы составить свою систему классов. Исходя от определенного абстрактного класса. А для общего действия определить интерфейс. И этот интерфейс реализовать во всех необходимых классах. Таким образом если разно плановые классы обладают общим действием. То его лучше выносить в интерфейс. А для одно плановых классов которые имеют схожие состояние лучше использовать абстрактные классы.

Порождающий патерн

Порождающие патроны это паттерны проектирования которые занимаются созданием объектов они предоставляют способ создания объектов скрывая логику создания. Что делает код более читаемым эти паттерны помогают

* разъединить создание объекта от реализации
* Улучшить гибкость и подерживаемость кода
* Снизить связанность между классами.

Типы порождающих паттернов в C#

* Singlron - патерн обеспечитывпет создание только одного экземпляра класса. И предоставляет глобальную точку доступа к этому экземпляру

Пример singilton.

public class Singleton

{

private static Singleton \_instance;

private static object \_lock = new object();

private Singleton() { }

public static Singleton Instance

{

get

{

if (\_instance == null)

{

lock (\_lock)

{

if (\_instance == null)

{

\_instance = new Singleton();

}

}

}

return \_instance;

}

}

}

* Патерн factory создание объекта без указания класса которого будут созданы.

Пример

public abstract class Animal { }

public class Dog : Animal { }

public class Cat : Animal { }

public class AnimalFactory

{

public static Animal CreateAnimal(string type)

{

switch (type)

{

case "Dog":

return new Dog();

case "Cat":

return new Cat();

default:

throw new ArgumentException("Invalid animal type");

}

}

}

* Abstract factory предоставляет способ создавать семейство связанных объектов без указания конкретных классов.

puу blic abstract class Animal { }

public class Dog : Animal { }

public class Cat : Animal { }

public abstract class AnimalFactory

{

public abstract Animal CreateAnimal();

}

public class DogFactory : AnimalFactory

{

public override Animal CreateAnimal()

{

return new Dog();

}

}

public class CatFactory : AnimalFactory

{

public override Animal CreateAnimal()

{

return new Cat();

}

}

* патерн билдер отделяет строитель о оььекта от его представления что позволяет дочь с большой гибкости и повторного использования.

public class Car

{

public string Make { get; set; }

public string Model { get; set; }

public int Year { get; set; }

public Car(string make, string model, int year)

{

Make = make;

Model = model;

Year = year;

}

}

public class CarBuilder

{

private string \_make;

private string \_model;

private int \_year;

public CarBuilder WithMake(string make)

{

\_make = make;

return this;

}

public CarBuilder WithModel(string model)

{

\_model = model;

return this;

}

public CarBuilder WithYear(int year)

{

\_year = year;

return this;

}

public Car Build()

{

return new Car(\_make, \_model, \_year);

}

}

* патерн prototype создает новый объект копируя существующий объект, что позволяет достич большой гибкости и повторного использования

public class Car

{

public string Make { get; set; }

public string Model { get; set; }

public int Year { get; set; }

public Car(string make, string model, int year)

{

Make = make;

Model = model;

Year = year;

}

public Car Clone()

{

return (Car)this.MemberwiseClone();

}

}

* Object pool это патерн позволяющий повторно использовать объект вместо создания нового это помоч уменьшить затраты на создание объектов и уменьшить количество мусора в памяти.

public class ObjectPool<T>

{

private List<T> \_objects;

private Func<T> \_factory;

public ObjectPool(Func<T> factory)

{

\_objects = new List<T>();

\_factory = factory;

}

public T GetObject()

{

lock (\_objects)

{

if (\_objects.Count > 0)

{

T obj = \_objects[0];

\_objects.RemoveAt(0);

return obj;

}

else

{

return \_factory();

}

}

}

public void ReturnObject(T obj)

{

lock (\_objects)

{

\_objects.Add(obj);

}

}

}

* флюент интерфейс. Паттерн который позволяет создавать объект с помощью цепочки методов. Каждый из которых возвращает объект.

public class FluentCar

{

private string \_make;

private string \_model;

private int \_year;

public FluentCar WithMake(string make)

{

\_make = make;

return this;

}

public FluentCar WithModel(string model)

{

\_model = model;

return this;

}

public FluentCar WithYear(int year)

{

\_year = year;

return this;

}

public Car Build()

{

return new Car(\_make, \_model, \_year);

}

}

* патерн Лейзи инитилиазийшен. Паттерн который позволяет отложить создание объекта до тех пор пока он не будет необходим.

public class LazyInitialization

{

private Lazy<Car> \_car;

public LazyInitialization()

{

\_car = new Lazy<Car>(() => new Car("Toyota", "Corolla", 2015));

}

public Car GetCar()

{

return \_car.Value;

}

}

## Потоки

Потоки это способ выполнения не кольких задач одновременно в одной программе в с# реализуется convert. Trsystemtrekihh создание потоков что бы создать поток н8. Ттббудет выполню в этом потоке. Делегат может быть любым метод (voidMethod()

using System.Threading;

class MyThread

{

public void Run()

{

for (int i = 0; i < 10; i++)

{

Console.WriteLine("Hello from thread!");

Thread.Sleep(1000);

}

}

}

class Program

{

static void Main()

{

MyThread thread = new MyThread();

Thread t = new Thread(thread.Run);

t.Start();

}

}

В этом примере мы создаем класса и передаем ему метод а затем запускаем поток методом старт.

**Синхронизация потоков.**

когда несколько потоков выполняют несколько потоков задачи одновременно может возникнуть проблема синхронизации. Синхронизация потоков это процесс координации доступа к общим ресурсам что бы предотвратить ошибки и обеспечить корректное выполнение программы. В C# есть несколько способов синхронизации потоков.

* Lock: блокирует доступ к ресурсу пока поток не освободит его.

class MyThread

{

private object \_lock = new object();

private int \_counter = 0;

public void Run()

{

for (int i = 0; i < 10; i++)

{

lock (\_lock)

{

\_counter++;

Console.WriteLine("Counter: " + \_counter);

}

Thread.Sleep(1000);

}

}

}

В этом примере мы используем блокировку для ограничения доступа к переменной.

* Mytinhts позволяет нескольким потоком доступ к ресурсу но только одному потоку в каждый момент времени.
* Semaphore позволяет ограничить количество потоков которые имеют доступ к ресурсу.

Асинхронное программирование

Это способы выполнения задач которые не блокируют основной поток программ. В c# есть несколько способов асинхронного программирования.

1. Asynk/away- позволяет писать асинхронный код который выглядит как синхронный.
2. task - позволяет создавать задачи которые могут быть выполнены асинхронно.

class MyAsyncMethod

{

public async Task Run()

{

await Task.Delay(1000);

Console.WriteLine("Hello from async method!");

}

}

class Program

{

static async Task Main()

{

MyAsyncMethod asyncMethod = new MyAsyncMethod();

await asyncMethod.Run();

}

}

Созданием асинхронный метод.

Потоки и UI когда мы работаем с пользовательскими интерфейсом мы должны быть осторожны с потоками. UI компоненты только из основного потока программ. В шарпах есть несколько способов обновление Уайт из потоков

* Invoke позволяет вызвать метод из основного потока программ.
* Begin invoke позволяет вызвать метод из основного потока программы асинхронно.

class MyForm : Form

{

private Label \_label;

public MyForm()

{

\_label = new Label();

this.Controls.Add(\_label);

}

public void UpdateLabel(string text)

{

if (\_label.InvokeRequired)

{

\_label.Invoke((Action<string>)UpdateLabel, text);

}

else

{

\_label.Text = text;

}

}

}

В этом примере мы использовали метод инвок для обновление текста из потока.

# Листинг Модуль.

#### Класс Пользователь

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace WindowsFormsApp2

{

public class User

{

public string Username { get; set; }

public string Password { get; set; }

public string Role { get; set; } // "Admin" или "User"

public override string ToString()

{

return Username;

}

}

}

#### Класс книга

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using Newtonsoft.Json;

using System.IO;

namespace WindowsFormsApp1

{

public class Book

{

public Guid Id { get; set; } // Идентификатор книги (UUID)

public string Title { get; set; } // Название книги

public string Author { get; set; } // Автор книги

public int Year { get; set; } // Год издания книги

// Конструктор для инициализации книги с передачей всех данных, включая ID

public Book(Guid id, string title, string author, int year)

{

Id = id; // Устанавливаем Id

Title = title;

Author = author;

Year = year;

}

// Конструктор без Id, для случаев, когда Id присваивается после создания

public Book(string title, string author, int year)

{

Title = title;

Author = author;

Year = year;

}

// Переопределенный метод ToString для удобного вывода информации о книге

public override string ToString()

{

return $"ID: {Id}, Название: {Title}, Автор: {Author}, Год: {Year}";

}

}

}

#### Класс книжный менеджер

using System;

using System.Windows.Forms;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using Newtonsoft.Json;

using System.IO;

using iText.Kernel.Pdf;

using iText.Layout;

using iText.Layout.Element; // Для Text в iTextSharp

using iText.Kernel.Font;

using iText.Kernel.Pdf.Canvas.Parser;

using DocumentFormat.OpenXml.Packaging;

using DocumentFormat.OpenXml.Wordprocessing; // Для Text в Open XML SDK

using DocumentFormat.OpenXml.ExtendedProperties;

using Npgsql;

using System.Drawing;

using ZXing;

namespace WindowsFormsApp1

{

public class BookManager

{

private readonly string \_connectionString;

// Конструктор с передачей строки подключения

public BookManager(string connectionString)

{

\_connectionString = connectionString;

}

// Метод для получения всех книг из базы данных

public List<Book> GetAllBooks()

{

// Создаем список для хранения книг

var books = new List<Book>();

// Создаем подключение к базе данных

using (var conn = new NpgsqlConnection(\_connectionString))

{

conn.Open();

// Пример SQL-запроса для получения всех книг

var cmd = new NpgsqlCommand("SELECT \* FROM books", conn);

var reader = cmd.ExecuteReader();

// Чтение данных из базы данных и добавление в список

while (reader.Read())

{

var book = new Book(

reader.GetString(1), // Title

reader.GetString(2), // Author

reader.GetInt32(3) // Year

)

{

Id = reader.GetGuid(0)

};

books.Add(book); // Добавляем книгу в список

}

}

return books;

}

public void AddBook(Book book)

{

using (var conn = new NpgsqlConnection(\_connectionString))

{

conn.Open();

// SQL-запрос для вставки новой книги в базу данных

var cmd = new NpgsqlCommand("INSERT INTO books (title, author, year) VALUES (@title, @author, @year) RETURNING id", conn);

// Добавление параметров в запрос для предотвращения SQL-инъекций

cmd.Parameters.AddWithValue("@title", book.Title);

cmd.Parameters.AddWithValue("@author", book.Author);

cmd.Parameters.AddWithValue("@year", book.Year);

// Извлечение сгенерированного id из базы данных

var id = cmd.ExecuteScalar(); // Получаем результат выполнения запроса, который будет UUID

// Присваиваем сгенерированный Id объекту book

book.Id = (Guid)id; // Преобразуем результат в тип Guid

// Теперь у вас есть объект book с заполненным Id

}

}

public List<Book> FindBookByName(string title)

{

var books = new List<Book>();

using (var connection = new NpgsqlConnection(\_connectionString))

{

connection.Open();

// Запрос для поиска книг по названию

var command = new NpgsqlCommand("SELECT \* FROM books WHERE title ILIKE @Title", connection);

command.Parameters.AddWithValue("@Title", "%" + title + "%");

using (var reader = command.ExecuteReader())

{

while (reader.Read())

{

var book = new Book(

reader.GetString(1), // Title

reader.GetString(2), // Author

reader.GetInt32(3) // Year

)

{

Id = reader.GetGuid(0) // Id устанавливаем отдельно

};

books.Add(book); // Добавляем книгу в список

}

}

}

return books;

}

// Метод для поиска книги по автору

public List<Book> FindBookByAuthor(string author)

{

var books = new List<Book>();

using (var connection = new NpgsqlConnection(\_connectionString))

{

connection.Open();

// Запрос для поиска книг по автору

var command = new NpgsqlCommand("SELECT \* FROM books WHERE author ILIKE @Author", connection);

command.Parameters.AddWithValue("@Author", "%" + author + "%");

using (var reader = command.ExecuteReader())

{

while (reader.Read())

{

var book = new Book(

reader.GetString(1), // Title

reader.GetString(2), // Author

reader.GetInt32(3) // Year

)

{

Id = reader.GetGuid(0) // Id устанавливаем отдельно

};

books.Add(book); // Добавляем книгу в список

}

}

}

return books;

}

public void RemoveBook(Guid bookId)

{

using (var connection = new NpgsqlConnection(\_connectionString))

{

connection.Open();

// Запрос для удаления книги по Id

var command = new NpgsqlCommand("DELETE FROM books WHERE id = @BookId", connection);

command.Parameters.AddWithValue("@BookId", bookId);

command.ExecuteNonQuery(); // Выполнение запроса

}

}

public void ExportBooksToJson(string filePath)

{

var books = new List<Book>();

string json = JsonConvert.SerializeObject(books, Formatting.Indented); // Преобразуем список в JSON

File.WriteAllText(filePath, json); // Записываем в файл

}

public void ImportBooksFromJson(string filePath)

{

if (File.Exists(filePath))

{

var books = new List<Book>();

string json = File.ReadAllText(filePath); // Читаем JSON из файла

var importedBooks = JsonConvert.DeserializeObject<List<Book>>(json); // Преобразуем JSON в список книг

if (importedBooks != null)

{

books.AddRange(importedBooks); // Добавляем книги в существующий список

}

}

}

// Экспорт в PDF

public void ExportBooksToPdf(string filePath, List<Book> books)

{

using (var writer = new PdfWriter(filePath))

{

using (var pdf = new PdfDocument(writer))

{

// Получаем доступ к метаданным документа

var docInfo = pdf.GetDocumentInfo();

// Устанавливаем метаданные

docInfo.SetTitle("Список книг");

docInfo.SetAuthor("Your Author Name");

docInfo.SetSubject("Список книг в библиотеке");

docInfo.SetCreator("Book Manager Application");

docInfo.SetKeywords("Books, Library, Export, PDF");

// Создаем Document для добавления содержимого

var document = new iText.Layout.Document(pdf);

// Заголовок PDF документа

// Используем шрифт, поддерживающий кириллицу

var font = PdfFontFactory.CreateFont("C:/Windows/Fonts/arial.ttf", PdfFontFactory.EmbeddingStrategy.PREFER\_EMBEDDED);

document.Add(new iText.Layout.Element.Paragraph("Список книг").SetFont(font));

// Добавление каждой книги в PDF

foreach (var book in books)

{

document.Add(new iText.Layout.Element.Paragraph($"ID: {book.Id}").SetFont(font));

document.Add(new iText.Layout.Element.Paragraph($"Название: {book.Title}").SetFont(font));

document.Add(new iText.Layout.Element.Paragraph($"Автор: {book.Author}").SetFont(font));

document.Add(new iText.Layout.Element.Paragraph($"Год: {book.Year}").SetFont(font));

document.Add(new iText.Layout.Element.Paragraph("\n").SetFont(font));

}

// Закрытие документа

document.Close();

}

}

}

// Экспорт в DOCX

public void ExportBooksToDocx(string filePath, List<Book> books)

{

using (var wordDoc = WordprocessingDocument.Create(filePath, DocumentFormat.OpenXml.WordprocessingDocumentType.Document))

{

var mainPart = wordDoc.AddMainDocumentPart();

mainPart.Document = new DocumentFormat.OpenXml.Wordprocessing.Document(new Body());

var body = mainPart.Document.Body;

// Заголовок Word документа

body.AppendChild(new DocumentFormat.OpenXml.Wordprocessing.Paragraph(new Run(new DocumentFormat.OpenXml.Wordprocessing.Text("Список книг")))); // Явно указываем Open XML для Text

// Добавление каждой книги в Word документ

foreach (var book in books)

{

body.AppendChild(new DocumentFormat.OpenXml.Wordprocessing.Paragraph(new Run(new DocumentFormat.OpenXml.Wordprocessing.Text($"ID: {book.Id}"))));

body.AppendChild(new DocumentFormat.OpenXml.Wordprocessing.Paragraph(new Run(new DocumentFormat.OpenXml.Wordprocessing.Text($"Название: {book.Title}"))));

body.AppendChild(new DocumentFormat.OpenXml.Wordprocessing.Paragraph(new Run(new DocumentFormat.OpenXml.Wordprocessing.Text($"Автор: {book.Author}"))));

body.AppendChild(new DocumentFormat.OpenXml.Wordprocessing.Paragraph(new Run(new DocumentFormat.OpenXml.Wordprocessing.Text($"Год: {book.Year}"))));

body.AppendChild(new DocumentFormat.OpenXml.Wordprocessing.Paragraph(new Run(new DocumentFormat.OpenXml.Wordprocessing.Text("\n"))));

}

wordDoc.Save();

}

}

public List<Book> ImportBooksFromPdf(string filePath)

{

List<Book> books = new List<Book>();

using (PdfReader reader = new PdfReader(filePath))

{

using (PdfDocument pdfDoc = new PdfDocument(reader))

{

// Прочитаем все страницы PDF

int numPages = pdfDoc.GetNumberOfPages();

for (int i = 1; i <= numPages; i++)

{

var page = pdfDoc.GetPage(i);

var text = PdfTextExtractor.GetTextFromPage(page);

// Разделяем текст на строки

var lines = text.Split('\n');

Book book = null;

// Допустим, что формат текста всегда одинаковый

foreach (var line in lines)

{

if (line.Contains("ID:"))

{

book = new Book("", "", 0); // Инициализация с пустыми значениями

// Преобразуем строку в Guid, используя Guid.Parse

var idString = line.Replace("ID:", "").Trim();

if (Guid.TryParse(idString, out Guid id))

{

book.Id = id;

}

else

{

// Если преобразование не удалось, обрабатываем ошибку (например, можно записать в лог)

book.Id = Guid.Empty; // Можно присвоить пустой Guid

}

}

else if (line.Contains("Название:") && book != null)

{

book.Title = line.Replace("Название:", "").Trim();

}

else if (line.Contains("Автор:") && book != null)

{

book.Author = line.Replace("Автор:", "").Trim();

}

else if (line.Contains("Год:") && book != null)

{

if (int.TryParse(line.Replace("Год:", "").Trim(), out int year))

{

book.Year = year;

}

books.Add(book);

book = null; // Сброс после добавления книги

}

}

}

}

}

return books;

}

public List<Book> ImportBooksFromDocx(string filePath)

{

List<Book> books = new List<Book>();

// Открываем документ DOCX

using (WordprocessingDocument wordDocument = WordprocessingDocument.Open(filePath, false))

{

// Получаем тело документа

Body body = wordDocument.MainDocumentPart.Document.Body;

Book book = null;

// Проходим по каждому параграфу в документе

foreach (var paragraph in body.Elements<DocumentFormat.OpenXml.Wordprocessing.Paragraph>())

{

// Получаем текст из параграфа

string text = paragraph.InnerText.Trim();

// Проверка на наличие ключевых слов для поиска

if (text.Contains("ID:"))

{

book = new Book("", "", 0); // Инициализация книги с пустыми значениями

// Извлекаем ID

var idString = text.Replace("ID:", "").Trim();

if (Guid.TryParse(idString, out Guid id))

{

book.Id = id;

}

else

{

// Если ID не удалось разобрать, записываем пустой Guid

book.Id = Guid.Empty;

}

}

else if (text.Contains("Название:") && book != null)

{

// Извлечение названия книги

book.Title = text.Replace("Название:", "").Trim();

}

else if (text.Contains("Автор:") && book != null)

{

// Извлечение автора книги

book.Author = text.Replace("Автор:", "").Trim();

}

else if (text.Contains("Год:") && book != null)

{

// Извлечение года

if (int.TryParse(text.Replace("Год:", "").Trim(), out int year))

{

book.Year = year;

}

// Добавляем книгу в список

books.Add(book);

book = null; // Сброс книги после добавления

}

}

}

// Возвращаем список книг

return books;

}

public void ConvertFile(string inputFilePath, string outputFilePath)

{

if (Path.GetExtension(inputFilePath).ToLower() == ".pdf")

{

var books = ImportBooksFromPdf(inputFilePath);

ExportBooksToDocx(outputFilePath, books);

}

else if (Path.GetExtension(inputFilePath).ToLower() == ".docx")

{

var books = ImportBooksFromDocx(inputFilePath);

ExportBooksToPdf(outputFilePath, books);

}

else

{

throw new NotSupportedException("Формат файла не поддерживается для конвертации.");

}

}

private void ConvertDocument()

{

using (OpenFileDialog openFileDialog = new OpenFileDialog())

{

openFileDialog.Filter = "Supported Files|\*.pdf;\*.docx";

if (openFileDialog.ShowDialog() == DialogResult.OK)

{

string inputFilePath = openFileDialog.FileName;

using (SaveFileDialog saveFileDialog = new SaveFileDialog())

{

saveFileDialog.Filter = inputFilePath.EndsWith(".pdf")

? "Word Document|\*.docx"

: "PDF File|\*.pdf";

if (saveFileDialog.ShowDialog() == DialogResult.OK)

{

string outputFilePath = saveFileDialog.FileName;

try

{

ConvertFile(inputFilePath, outputFilePath);

MessageBox.Show("Конвертация завершена!");

}

catch (Exception ex)

{

MessageBox.Show($"Ошибка конвертации: {ex.Message}");

}

}

}

}

}

}

}

}

#### Код авторизации

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using BCrypt.Net;

namespace WindowsFormsApp2

{

public partial class LoginForm : Form

{

private readonly UserManager \_userManager;

public User AuthenticatedUser { get; private set; }

public LoginForm(UserManager userManager)

{

\_userManager = userManager;

InitializeComponent();

}

private void btnLogin\_Click(object sender, EventArgs e)

{

var username = txtUsername.Text;

var password = txtPassword.Text;

var user = \_userManager.Authenticate(username, password);

if (user != null)

{

// Если пользователь найден, сохраняем его и закрываем форму

AuthenticatedUser = user;

DialogResult = DialogResult.OK;

Close();

}

else

{

// Если логин или пароль неверные, показываем сообщение об ошибке

lblError.Visible = true;

}

}

}

}

Основная форма

using System;

using System.Linq;

using System.Windows.Forms;

using WindowsFormsApp1;

using Newtonsoft.Json;

using System.IO;

using iText.Kernel.Pdf;

using iText.Layout;

using iText.Layout.Element; // Для Text в iTextSharp

using iText.Kernel.Font;

using iText.Kernel.Pdf.Canvas.Parser;

using DocumentFormat.OpenXml.Packaging;

using DocumentFormat.OpenXml.Wordprocessing; // Для Text в Open XML SDK

using System.Collections.Generic;

using ZXing;

using WindowsFormsApp2;

using System.Drawing;

using Npgsql;

namespace WindowsFormsApp1

{

public partial class Form1 : Form

{

private BookManager \_bookManager;

private User \_currentUser;

private UserManager \_userManager;

// Строка подключения к базе данных

private string \_connectionString = "Host=localhost;Port=5432;Database=LibraryDB;Username=postgres;Password=danil2005";

public Form1(User currentUser)

{

InitializeComponent();

// Передаем строку подключения в BookManager и UserManager

\_bookManager = new BookManager(\_connectionString);

\_userManager = new UserManager(\_connectionString);

\_currentUser = currentUser;

openFileDialog = new System.Windows.Forms.OpenFileDialog();

ConfigureAccessBasedOnRole();

}

// Добавление книги

private void btnAdd\_Click(object sender, EventArgs e)

{

string title = txtTitle.Text;

string author = txtAuthor.Text;

int year;

if (string.IsNullOrEmpty(title) || string.IsNullOrEmpty(author) || !int.TryParse(txtYear.Text, out year))

{

MessageBox.Show("Пожалуйста, заполните все поля корректно.");

return;

}

// Создание новой книги

var book = new Book(title, author, year);

// Добавление книги в базу данных через BookManager

\_bookManager.AddBook(book);

// Обновление списка книг

UpdateBookList();

// Очистка текстовых полей

ClearInputFields();

}

// Удаление книги по ID

private void btnDelete\_Click(object sender, EventArgs e)

{

if (lstBooks.SelectedItem == null)

{

MessageBox.Show("Выберите книгу для удаления.");

return;

}

var selectedBook = (Book)lstBooks.SelectedItem;

\_bookManager.RemoveBook(selectedBook.Id);

UpdateBookList();

}

// Поиск книги по названию

private void btnSearchTitle\_Click(object sender, EventArgs e)

{

string title = txtTitle.Text;

if (!string.IsNullOrEmpty(title))

{

var books = \_bookManager.FindBookByName(title);

lstBooks.DataSource = books;

}

}

// Поиск книги по автору

private void btnSearchAuthor\_Click(object sender, EventArgs e)

{

string author = txtAuthor.Text;

if (!string.IsNullOrEmpty(author))

{

var books = \_bookManager.FindBookByAuthor(author);

lstBooks.DataSource = books;

}

}

// Обновление списка книг в ListBox

private void UpdateBookList()

{

lstBooks.DataSource = null; // Очистка DataSource

lstBooks.DataSource = \_bookManager.GetAllBooks();

}

// Метод для обработки нажатия на кнопку "Показать все книги"

private void btnShowAllBooks\_Click(object sender, EventArgs e)

{

// Получаем все книги из менеджера

var allBooks = \_bookManager.GetAllBooks();

// Обновляем список на форме, отображая все книги

lstBooks.DataSource = null; // Сначала очищаем текущий список

lstBooks.DataSource = allBooks; // Затем заполняем его заново

}

// Очистка текстовых полей

private void ClearInputFields()

{

txtTitle.Clear();

txtAuthor.Clear();

txtYear.Clear();

}

private void btnExport\_Click(object sender, EventArgs e)

{

using (SaveFileDialog saveFileDialog = new SaveFileDialog())

{

saveFileDialog.Filter = "JSON files (\*.json)|\*.json|All files (\*.\*)|\*.\*";

if (saveFileDialog.ShowDialog() == DialogResult.OK)

{

\_bookManager.ExportBooksToJson(saveFileDialog.FileName);

MessageBox.Show("Книги успешно экспортированы!");

}

}

}

private void btnImport\_Click(object sender, EventArgs e)

{

using (OpenFileDialog openFileDialog = new OpenFileDialog())

{

openFileDialog.Filter = "JSON files (\*.json)|\*.json|All files (\*.\*)|\*.\*";

if (openFileDialog.ShowDialog() == DialogResult.OK)

{

\_bookManager.ImportBooksFromJson(openFileDialog.FileName);

UpdateBookList();

MessageBox.Show("Книги успешно импортированы!");

}

}

}

// Обработчик кнопки для экспорта в DOCX

private void btnExportToDocx\_Click(object sender, EventArgs e)

{

var books = \_bookManager.GetAllBooks(); // Получаем все книги

var saveFileDialog = new SaveFileDialog

{

Filter = "Word Document|\*.docx",

FileName = "BooksList.docx"

};

if (saveFileDialog.ShowDialog() == DialogResult.OK)

{

string filePath = saveFileDialog.FileName;

\_bookManager.ExportBooksToDocx(filePath, books); // Экспортируем в DOCX

MessageBox.Show("Книги экспортированы в Word!");

}

}

// Обработчик кнопки для экспорта в PDF

private void btnExportToPdf\_Click(object sender, EventArgs e)

{

var books = \_bookManager.GetAllBooks(); // Получаем все книги

var saveFileDialog = new SaveFileDialog

{

Filter = "PDF File|\*.pdf",

FileName = "BooksList.pdf"

};

if (saveFileDialog.ShowDialog() == DialogResult.OK)

{

string filePath = saveFileDialog.FileName;

\_bookManager.ExportBooksToPdf(filePath, books); // Экспортируем в PDF

MessageBox.Show("Книги экспортированы в PDF!");

}

}

private void btnImportFromPdf\_Click(object sender, EventArgs e)

{

openFileDialog.Filter = "PDF Files|\*.pdf";

if (openFileDialog.ShowDialog() == DialogResult.OK)

{

string filePath = openFileDialog.FileName;

if (!File.Exists(filePath))

{

MessageBox.Show("Не удалось найти файл.");

return;

}

List<Book> books = \_bookManager.ImportBooksFromPdf(filePath);

lstBooks.DataSource = books;

MessageBox.Show($"Импортировано {books.Count} книг из PDF.");

}

}

// Импорт из DOCX

private void btnImportFromDocx\_Click(object sender, EventArgs e)

{

// Настройка фильтра для диалогового окна

openFileDialog.Filter = "Word Files|\*.docx";

// Открытие диалогового окна для выбора файла

if (openFileDialog.ShowDialog() == DialogResult.OK)

{

string filePath = openFileDialog.FileName;

// Проверка на существование файла

if (!File.Exists(filePath))

{

MessageBox.Show("Не удалось найти файл.");

return;

}

// Вызов метода импорта для DOCX

List<Book> books = \_bookManager.ImportBooksFromDocx(filePath);

// Установка источника данных для DataGridView (предполагается, что lstBooks это DataGridView)

lstBooks.DataSource = books;

// Показ сообщения о количестве импортированных книг

MessageBox.Show($"Импортировано {books.Count} книг из DOCX.");

}

}

private void ConvertDocument(string inputFilePath)

{

try

{

// Определение выходного файла на основе расширения входного

string outputFilePath = Path.ChangeExtension(inputFilePath,

Path.GetExtension(inputFilePath).ToLower() == ".pdf" ? ".docx" : ".pdf");

\_bookManager.ConvertFile(inputFilePath, outputFilePath);

MessageBox.Show($"Файл успешно конвертирован: {outputFilePath}");

}

catch (Exception ex)

{

MessageBox.Show($"Ошибка конвертации: {ex.Message}");

}

}

private void btnConvert\_Click(object sender, EventArgs e)

{

using (OpenFileDialog openFileDialog = new OpenFileDialog())

{

openFileDialog.Filter = "PDF and DOCX Files|\*.pdf;\*.docx";

if (openFileDialog.ShowDialog() == DialogResult.OK)

{

ConvertDocument(openFileDialog.FileName);

}

}

}

private void btnOpenBook\_Click(object sender, EventArgs e)

{

using (OpenFileDialog openFileDialog = new OpenFileDialog())

{

openFileDialog.Filter = "PDF and Word Files|\*.pdf;\*.docx|All files (\*.\*)|\*.\*";

if (openFileDialog.ShowDialog() == DialogResult.OK)

{

string filePath = openFileDialog.FileName;

// Открываем файл с помощью стандартной программы

try

{

System.Diagnostics.Process.Start(new System.Diagnostics.ProcessStartInfo

{

FileName = filePath,

UseShellExecute = true // Используем системное приложение для открытия

});

}

catch (Exception ex)

{

MessageBox.Show($"Ошибка при открытии файла: {ex.Message}");

}

}

}

}

private void ConfigureAccessBasedOnRole()

{

if (\_currentUser.Role == "User")

{

btnAdd.Enabled = false;

btnDelete.Enabled = false;

}

}

private void btnGenerateQRCode\_Click(object sender, EventArgs e)

{

if (lstBooks.SelectedItem is Book selectedBook)

{

GenerateQRCode(selectedBook.Title, selectedBook.Author, selectedBook.Year.ToString());

}

else

{

MessageBox.Show("Пожалуйста, выберите книгу из списка.");

}

}

private void GenerateQRCode(string title, string author, string year)

{

// Создаем текст для QR-кода

string qrText = $"Название: {title}\nАвтор: {author}\nГод: {year}";

// Создаем объект BarcodeWriter для генерации QR-кода

var qrCodeWriter = new ZXing.BarcodeWriter

{

Format = ZXing.BarcodeFormat.QR\_CODE,

Options = new ZXing.Common.EncodingOptions

{

Width = 250,

Height = 250,

Margin = 1

},

Renderer = new ZXing.Rendering.BitmapRenderer()

};

// Добавляем кодировку UTF-8 для поддержки кириллицы

qrCodeWriter.Options.Hints[ZXing.EncodeHintType.CHARACTER\_SET] = "UTF-8";

try

{

// Генерируем изображение QR-кода

var qrCodeImage = qrCodeWriter.Write(qrText);

// Отображаем изображение QR-кода в PictureBox

pictureBoxQRCode.Image = qrCodeImage;

}

catch (Exception ex)

{

MessageBox.Show($"Ошибка генерации QR-кода: {ex.Message}");

}

}

}

}

Program.cs  
  
  
using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using System.Windows.Forms;

using Newtonsoft.Json;

using System.IO;

using WindowsFormsApp2;

namespace WindowsFormsApp1

{

internal static class Program

{

[STAThread]

static void Main()

{

Application.EnableVisualStyles();

Application.SetCompatibleTextRenderingDefault(false);

// Ваша строка подключения к базе данных PostgreSQL

string connectionString = "Host=localhost;Port=5432;Database=LibraryDB;Username=postgres;Password=danil2005";

// Создайте экземпляры BookManager и UserManager с передачей строки подключения

var userManager = new UserManager(connectionString); // Передаем строку подключения

var loginForm = new LoginForm(userManager);

if (loginForm.ShowDialog() == DialogResult.OK)

{

var mainForm = new Form1(loginForm.AuthenticatedUser);

Application.Run(mainForm);

}

}

}

}

UserManager.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using Npgsql;

using BCrypt.Net;

namespace WindowsFormsApp2

{

public class UserManager

{

private readonly string \_connectionString;

// Конструктор с передачей строки подключения

public UserManager(string connectionString)

{

\_connectionString = connectionString;

}

// Метод для добавления пользователя

public User Authenticate(string username, string password)

{

using (var conn = new NpgsqlConnection(\_connectionString))

{

conn.Open();

// Запрос для получения пользователя по логину и паролю

var cmd = new NpgsqlCommand("SELECT \* FROM users WHERE username = @username AND password = @password", conn);

cmd.Parameters.AddWithValue("@username", username);

cmd.Parameters.AddWithValue("@password", password); // Сравниваем пароли как обычные строки

var reader = cmd.ExecuteReader();

if (reader.Read())

{

// Если нашли пользователя, возвращаем объект User

return new User

{

Username = reader.GetString(0),

Password = reader.GetString(1), // Пароль сохраняем в открытом виде (не рекомендуется)

Role = reader.GetString(2)

};

}

}

return null; // Если не нашли, возвращаем null

}

}

}

**Фото программы**

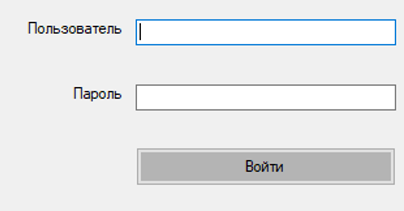


Рисунок 1



Рисунок 2

Задание по подготовке к демо-экзамену.

Databasehelper.cs  
  
  
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using Npgsql;

using System;

using System.Data;

using System.Windows.Forms;

namespace Разработка\_программного\_модуля

{

public class DatabaseHelper

{

private string connectionString = "Host=localhost;Username=postgres;Password=danil2005;Database=repairdb";

private NpgsqlConnection connection;

public DatabaseHelper()

{

connection = new NpgsqlConnection(connectionString);

}

public void ExecuteNonQuery(string query, NpgsqlParameter[] parameters = null)

{

try

{

connection.Open();

using (var cmd = new NpgsqlCommand(query, connection))

{

if (parameters != null)

{

cmd.Parameters.AddRange(parameters);

}

cmd.ExecuteNonQuery();

}

}

catch (Exception ex)

{

MessageBox.Show($"Ошибка выполнения запроса: {ex.Message}");

}

finally

{

connection.Close();

}

}

public DataTable ExecuteQuery(string query, NpgsqlParameter[] parameters = null)

{

var table = new DataTable();

try

{

connection.Open();

using (var cmd = new NpgsqlCommand(query, connection))

{

if (parameters != null)

{

cmd.Parameters.AddRange(parameters);

}

using (var reader = cmd.ExecuteReader())

{

table.Load(reader);

}

}

}

catch (Exception ex)

{

MessageBox.Show($"Ошибка выполнения запроса: {ex.Message}");

}

finally

{

connection.Close();

}

return table;

}

}

}

Форма ExtendDeadlineForm  
using Npgsql;

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace Разработка\_программного\_модуля

{

public partial class ExtendDeadlineForm : Form

{

private readonly int requestId;

private readonly DatabaseHelper dbHelper;

public ExtendDeadlineForm(int requestId)

{

InitializeComponent();

dbHelper = new DatabaseHelper(); // Убедитесь, что dbHelper инициализирован

this.requestId = requestId;

}

private void btnExtend\_Click(object sender, EventArgs e)

{

try

{

// Убедитесь, что requestId был передан и не равен 0

if (requestId == 0)

{

MessageBox.Show("Идентификатор заявки не задан.");

return;

}

// Проверка, что новая дата не пустая

if (dtpNewDeadline.Value == null)

{

MessageBox.Show("Пожалуйста, выберите новый срок выполнения.");

return;

}

// Убедитесь, что dbHelper инициализирован

if (dbHelper == null)

{

MessageBox.Show("Ошибка базы данных: не удалось инициализировать соединение.");

return;

}

// Подготовка SQL-запроса для обновления нового срока

string query = @"

UPDATE ЗаявкиНаРемонт

SET НовыйСрокВыполнения = @НовыйСрокВыполнения

WHERE Идентификатор = @Идентификатор";

var parameters = new List<NpgsqlParameter>

{

new NpgsqlParameter("@НовыйСрокВыполнения", NpgsqlTypes.NpgsqlDbType.Date) { Value = dtpNewDeadline.Value },

new NpgsqlParameter("@Идентификатор", NpgsqlTypes.NpgsqlDbType.Integer) { Value = requestId }

};

// Выполнение запроса с параметрами

dbHelper.ExecuteNonQuery(query, parameters.ToArray());

MessageBox.Show("Срок выполнения заявки продлен!");

}

catch (Exception ex)

{

// Обработка ошибок

MessageBox.Show($"Ошибка при обновлении срока заявки: {ex.Message}");

}

}

// Метод для получения заявки по ее идентификатору

private Request GetRequestById(int id)

{

try

{

string connectionString = "Host=localhost;Port=5432;Username=postgres;Password=danil2005;Database=repairdb";

using (var conn = new NpgsqlConnection(connectionString))

{

conn.Open();

string query = "SELECT \* FROM ЗаявкиНаРемонт WHERE Идентификатор = @Идентификатор";

using (var cmd = new NpgsqlCommand(query, conn))

{

cmd.Parameters.AddWithValue("@Идентификатор", id);

using (var reader = cmd.ExecuteReader())

{

if (reader.Read())

{

// Создаем объект заявки

var request = new Request

{

Id = Convert.ToInt32(reader["Идентификатор"]),

// Проверка на DBNull перед конвертацией

Deadline = reader["ДатаВыполнения"] != DBNull.Value ? Convert.ToDateTime(reader["ДатаВыполнения"]) : DateTime.MinValue // Если NULL, задаем значение по умолчанию

};

return request;

}

}

}

}

}

catch (Exception ex)

{

MessageBox.Show($"Ошибка при получении заявки: {ex.Message}");

}

return null;

}

// Метод для обновления заявки в базе данных

private void UpdateRequest(Request request)

{

try

{

using (var conn = new NpgsqlConnection("YourConnectionString"))

{

conn.Open();

string query = "UPDATE ЗаявкиНаРемонт SET ДатаВыполнения = @ДатаВыполнения WHERE Идентификатор = @Идентификатор";

using (var cmd = new NpgsqlCommand(query, conn))

{

cmd.Parameters.AddWithValue("@ДатаВыполнения", request.Deadline);

cmd.Parameters.AddWithValue("@Идентификатор", request.Id);

cmd.ExecuteNonQuery();

}

}

}

catch (Exception ex)

{

MessageBox.Show($"Ошибка при обновлении заявки: {ex.Message}");

}

}

private void btnCancel\_Click(object sender, EventArgs e)

{

this.Close(); // Закрытие формы при нажатии на кнопку

}

}

}

Форма LoginForm

using Npgsql;

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using System.Security.Cryptography;

namespace Разработка\_программного\_модуля

{

public partial class LoginForm : Form

{

private readonly string connectionString = "Host=localhost;Username=postgres;Password=danil2005;Database=repairdb";

private readonly DatabaseHelper dbHelper;

public string CurrentUserRole { get; private set; }

public LoginForm()

{

InitializeComponent();

dbHelper = new DatabaseHelper();

}

private void btnLogin\_Click(object sender, EventArgs e)

{

string username = txtUsername.Text.Trim();

string password = txtPassword.Text.Trim();

if (string.IsNullOrEmpty(username) || string.IsNullOrEmpty(password))

{

MessageBox.Show("Введите логин и пароль!");

return;

}

string query = "SELECT Роль FROM Пользователи WHERE Логин = @Логин AND Пароль = @Пароль";

var parameters = new[]

{

new NpgsqlParameter("@Логин", username),

new NpgsqlParameter("@Пароль", password)

};

DataTable result = dbHelper.ExecuteQuery(query, parameters);

if (result.Rows.Count > 0)

{

CurrentUserRole = result.Rows[0]["Роль"].ToString();

this.DialogResult = DialogResult.OK;

this.Close();

}

else

{

MessageBox.Show("Неверный логин или пароль!");

}

}

private void btnCancel\_Click(object sender, EventArgs e)

{

Application.Exit();

}

}

}

MailForm.cs  
  
  
using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using System;

using System.Data;

using System.Windows.Forms;

using Npgsql;

using QRCoder;

namespace Разработка\_программного\_модуля

{

public partial class MainForm : Form

{

// Строка подключения к базе данных PostgreSQL

private readonly string connectionString = "Host=localhost;Port=5432;Username=postgres;Password=danil2005;Database=repairdb"; // Укажите свой пароль

private readonly DatabaseHelper dbHelper; // Объявление dbHelper

public MainForm(string role)

{

InitializeComponent();

dbHelper = new DatabaseHelper(); // Инициализация dbHelper

LoadRequests();

if (role == "Пользователь")

{

btnAddRequest.Visible = false;

btnEditRequest.Visible = false;

btnDeleteRequest.Visible = false;

btnUpdateStats.Visible = false;

btnExtendDeadline.Visible = false; // Кнопка для продления срока

txtQRCodeData.Visible = false;

}

else if (role == "Менеджер")

{

btnAddRequest.Visible = false;

btnDeleteRequest.Visible = false;

btnGenerateQRCode.Visible = false;

txtQRCodeData.Visible = false;

}

if (role == "Менеджер")

{

btnExtendDeadline.Visible = true; // Кнопка для продления срока

}

}

/// <summary>

/// Загружает все заявки из базы данных и отображает их в DataGridView.

/// </summary>

private void LoadRequests()

{

try

{

var query = "SELECT \* FROM ЗаявкиНаРемонт"; // SQL-запрос для получения всех заявок

var dataTable = ExecuteQuery(query); // Выполнение запроса и получение данных

dgvRequests.DataSource = dataTable;

dgvRequests.CellFormatting += dgvRequests\_CellFormatting;

}

catch (Exception ex)

{

MessageBox.Show($"Ошибка загрузки заявок: {ex.Message}", "Ошибка", MessageBoxButtons.OK, MessageBoxIcon.Error);

}

}

/// <summary>

/// Выполнение SQL-запроса и возврат результатов в виде DataTable.

/// </summary>

private DataTable ExecuteQuery(string query)

{

using (var connection = new NpgsqlConnection(connectionString))

{

var dataTable = new DataTable();

try

{

connection.Open();

using (var command = new NpgsqlCommand(query, connection))

{

using (var adapter = new NpgsqlDataAdapter(command))

{

adapter.Fill(dataTable);

}

}

}

catch (Exception ex)

{

MessageBox.Show($"Ошибка выполнения запроса: {ex.Message}", "Ошибка", MessageBoxButtons.OK, MessageBoxIcon.Error);

}

return dataTable;

}

}

/// <summary>

/// Выполнение SQL-запроса без возврата данных (например, для удаления).

/// </summary>

private void ExecuteNonQuery(string query, NpgsqlParameter[] parameters)

{

using (var connection = new NpgsqlConnection(connectionString))

{

try

{

connection.Open();

using (var command = new NpgsqlCommand(query, connection))

{

command.Parameters.AddRange(parameters);

command.ExecuteNonQuery(); // Выполняем запрос без возврата данных

}

}

catch (Exception ex)

{

MessageBox.Show($"Ошибка выполнения запроса: {ex.Message}", "Ошибка", MessageBoxButtons.OK, MessageBoxIcon.Error);

}

}

}

/// <summary>

/// Обработчик кнопки "Добавить заявку".

/// </summary>

private void btnAddRequest\_Click(object sender, EventArgs e)

{

var addForm = new RequestForm(); // Создаем форму для добавления заявки

if (addForm.ShowDialog() == DialogResult.OK)

{

LoadRequests(); // Обновляем список заявок после добавления

}

}

/// <summary>

/// Обработчик кнопки "Редактировать заявку".

/// </summary>

private void btnEditRequest\_Click(object sender, EventArgs e)

{

if (dgvRequests.SelectedRows.Count > 0)

{

var selectedRow = dgvRequests.SelectedRows[0];

var requestId = (int)selectedRow.Cells["Идентификатор"].Value; // Получаем ID выбранной заявки

var editForm = new RequestForm(requestId); // Открываем форму для редактирования

if (editForm.ShowDialog() == DialogResult.OK)

{

LoadRequests(); // Обновляем список заявок после редактирования

}

}

else

{

MessageBox.Show("Выберите заявку для редактирования.", "Предупреждение", MessageBoxButtons.OK, MessageBoxIcon.Warning);

}

}

/// <summary>

/// Обработчик кнопки "Удалить заявку".

/// </summary>

private void btnDeleteRequest\_Click(object sender, EventArgs e)

{

if (dgvRequests.SelectedRows.Count > 0)

{

var row = dgvRequests.SelectedRows[0];

var requestId = (int)row.Cells["Идентификатор"].Value;

// Удаление записи

var deleteQuery = "DELETE FROM ЗаявкиНаРемонт WHERE Идентификатор = @Идентификатор";

var deleteParams = new[]

{

new NpgsqlParameter("@Идентификатор", requestId)

};

dbHelper.ExecuteNonQuery(deleteQuery, deleteParams);

// Синхронизация последовательностей

var resetIdSequenceQuery = "SELECT setval('ЗаявкиНаРемонт\_Идентификатор\_seq', (SELECT COALESCE(MAX(\"Идентификатор\"), 1) FROM \"ЗаявкиНаРемонт\"), false)";

var resetNumSequenceQuery = "SELECT setval('ЗаявкиНаРемонт\_НомерЗаявки\_seq', (SELECT COALESCE(MAX(\"НомерЗаявки\")::integer, 1) FROM \"ЗаявкиНаРемонт\"), false)";

dbHelper.ExecuteNonQuery(resetIdSequenceQuery);

dbHelper.ExecuteNonQuery(resetNumSequenceQuery);

// Перезагрузка списка заявок

LoadRequests();

}

else

{

MessageBox.Show("Выберите заявку для удаления.");

}

}

private void SearchRequests(string searchQuery)

{

var query = "SELECT \* FROM \"ЗаявкиНаРемонт\" WHERE \"НомерЗаявки\" LIKE @SearchQuery";

var parameters = new[]

{

new NpgsqlParameter("@SearchQuery", "%" + searchQuery + "%")

};

var table = dbHelper.ExecuteQuery(query, parameters);

dgvRequests.DataSource = table;

}

private void btnSearch\_Click(object sender, EventArgs e)

{

string searchText = txtStatusSearch.Text.Trim(); // Получаем текст из текстового поля

string query;

if (int.TryParse(searchText, out int номерЗаявки))

{

// Если введённое значение - это число, ищем по "НомерЗаявки"

query = "SELECT \* FROM \"ЗаявкиНаРемонт\" WHERE \"НомерЗаявки\" = @searchValue";

var parameters = new[]

{

new NpgsqlParameter("@searchValue", номерЗаявки) // Передаём как integer

};

var table = dbHelper.ExecuteQuery(query, parameters);

dgvRequests.DataSource = table;

if (table.Rows.Count == 0)

{

MessageBox.Show("Заявки с указанным номером не найдены.", "Результаты поиска", MessageBoxButtons.OK, MessageBoxIcon.Information);

}

}

else

{

// Если введённое значение - не число, ищем по другим параметрам

query = "SELECT \* FROM \"ЗаявкиНаРемонт\" WHERE \"Оборудование\" ILIKE @searchValue OR \"ТипНеисправности\" ILIKE @searchValue OR \"Клиент\" ILIKE @searchValue";

var parameters = new[]

{

new NpgsqlParameter("@searchValue", $"%{searchText}%") // Используем подстановки для текстового поиска

};

var table = dbHelper.ExecuteQuery(query, parameters);

dgvRequests.DataSource = table;

if (table.Rows.Count == 0)

{

MessageBox.Show("Заявки по указанным параметрам не найдены.", "Результаты поиска", MessageBoxButtons.OK, MessageBoxIcon.Information);

}

}

}

private void dgvRequests\_CellFormatting(object sender, DataGridViewCellFormattingEventArgs e)

{

// Проверяем, что это колонка со статусом

if (dgvRequests.Columns[e.ColumnIndex].Name == "Статус")

{

string статус = e.Value?.ToString();

if (статус == "В работе")

{

e.CellStyle.BackColor = Color.Yellow;

e.CellStyle.ForeColor = Color.Black;

}

else if (статус == "Выполнено")

{

e.CellStyle.BackColor = Color.LightGreen;

e.CellStyle.ForeColor = Color.Black;

}

}

}

private void btnShowAllRequests\_Click(object sender, EventArgs e)

{

LoadRequests(); // Метод для загрузки всех заявок

}

private void UpdateStatistics()

{

try

{

string updateCompletedQuery = @"

UPDATE ""Статистика""

SET ""КоличествоВыполненных"" = (

SELECT COUNT(\*) FROM ""ЗаявкиНаРемонт"" WHERE ""Статус"" = 'Выполнено'

);";

string updateAvgTimeQuery = @"

UPDATE ""Статистика""

SET ""СреднееВремяВыполнения"" = (

SELECT AVG(""ДатаВыполнения"" - ""ДатаДобавления"")

FROM ""ЗаявкиНаРемонт""

WHERE ""Статус"" = 'Выполнено'

);";

string updateFaultTypeQuery = @"

INSERT INTO ""Статистика"" (""ТипНеисправности"", ""КоличествоПоТипу"")

SELECT ""ТипНеисправности"", COUNT(\*)

FROM ""ЗаявкиНаРемонт""

GROUP BY ""ТипНеисправности""

ON CONFLICT (""ТипНеисправности"")

DO UPDATE SET ""КоличествоПоТипу"" = EXCLUDED.""КоличествоПоТипу"";";

dbHelper.ExecuteNonQuery(updateCompletedQuery);

dbHelper.ExecuteNonQuery(updateAvgTimeQuery);

dbHelper.ExecuteNonQuery(updateFaultTypeQuery);

MessageBox.Show("Статистика успешно обновлена!");

}

catch (Exception ex)

{

MessageBox.Show("Ошибка при обновлении статистики: " + ex.Message);

}

}

private void btnUpdateStats\_Click(object sender, EventArgs e)

{

UpdateStatistics();

}

private void btnExtendDeadline\_Click(object sender, EventArgs e)

{

if (dgvRequests.SelectedRows.Count > 0)

{

var selectedRow = dgvRequests.SelectedRows[0];

var requestId = (int)selectedRow.Cells["Идентификатор"].Value;

var extendForm = new ExtendDeadlineForm(requestId); // Форма для продления срока

extendForm.ShowDialog();

// После продления срока обновляем таблицу

LoadRequests();

}

}

private void BtnGenerateQRCode\_Click(object sender, EventArgs e)

{

// Получаем данные из текстового поля

string qrCodeText = this.txtQRCodeData.Text;

if (string.IsNullOrEmpty(qrCodeText))

{

MessageBox.Show("Введите данные для генерации QR-кода.");

return;

}

try

{

// Генерация QR-кода с помощью QRCoder

QRCodeGenerator qrGenerator = new QRCodeGenerator();

QRCodeData qrCodeData = qrGenerator.CreateQrCode(qrCodeText, QRCodeGenerator.ECCLevel.Q); // Создаем данные QR

// Генерация QR-кода как изображение

QRCode qrCode = new QRCode(qrCodeData);

Bitmap qrCodeImage = qrCode.GetGraphic(5); // 20 - это размер пикселей QR-кода

// Отображаем QR-код в PictureBox

pictureBoxQRCode.Image = qrCodeImage;

}

catch (Exception ex)

{

MessageBox.Show("Ошибка при генерации QR-кода: " + ex.Message);

}

}

}

}

Program.cs  
  
using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace Разработка\_программного\_модуля

{

internal static class Program

{

/// <summary>

/// Главная точка входа для приложения.

/// </summary>

[STAThread]

static void Main()

{

Application.EnableVisualStyles();

Application.SetCompatibleTextRenderingDefault(false);

var loginForm = new LoginForm();

if (loginForm.ShowDialog() == DialogResult.OK)

{

Application.Run(new MainForm(loginForm.CurrentUserRole));

}

}

}

}

Request.cs  
  
  
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Разработка\_программного\_модуля

{

public class Request

{

public int Id { get; set; }

public DateTime Deadline { get; set; }

}

}

RequesrForm.cs

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using Npgsql;

namespace Разработка\_программного\_модуля

{

public partial class RequestForm : Form

{

private readonly DatabaseHelper dbHelper;

private int? requestId = null;

// Конструктор для добавления новой заявки

public RequestForm()

{

InitializeComponent();

dbHelper = new DatabaseHelper();

}

// Конструктор для редактирования заявки

public RequestForm(int requestId)

{

InitializeComponent();

dbHelper = new DatabaseHelper();

this.requestId = requestId;

LoadRequestData();

}

/// <summary>

/// Загрузка данных заявки для редактирования

/// </summary>

private void LoadRequestData()

{

if (requestId.HasValue)

{

string query = "SELECT \* FROM ЗаявкиНаРемонт WHERE Идентификатор = @Идентификатор";

var parameters = new[]

{

new NpgsqlParameter("@Идентификатор", requestId.Value)

};

DataTable table = dbHelper.ExecuteQuery(query, parameters);

if (table.Rows.Count > 0)

{

var row = table.Rows[0];

// Заполняем поля формы данными заявки

txtEquipment.Text = row["Оборудование"].ToString();

cmbFaultType.SelectedItem = row["ТипНеисправности"].ToString();

txtDescription.Text = row["ОписаниеПроблемы"].ToString();

txtClient.Text = row["Клиент"].ToString();

cmbStatus.SelectedItem = row["Статус"].ToString();

dtpRequestDate.Value = Convert.ToDateTime(row["ДатаДобавления"]);

}

}

}

/// <summary>

/// Сохранение данных заявки в базу данных

/// </summary>

private void btnSaveRequest\_Click(object sender, EventArgs e)

{

// Проверка, что все поля заполнены

if (string.IsNullOrEmpty(txtEquipment.Text) ||

string.IsNullOrEmpty(txtDescription.Text) ||

string.IsNullOrEmpty(txtClient.Text) ||

string.IsNullOrEmpty(txtResponsible.Text) || // Проверка заполненности нового поля

string.IsNullOrEmpty(txtComments.Text) || // Проверка заполненности нового поля

cmbFaultType.SelectedIndex == -1 ||

cmbStatus.SelectedIndex == -1)

{

MessageBox.Show("Пожалуйста, заполните все поля!");

return;

}

// Собираем данные из полей

string equipment = txtEquipment.Text;

string faultType = cmbFaultType.SelectedItem.ToString();

string description = txtDescription.Text;

string client = txtClient.Text;

string status = cmbStatus.SelectedItem.ToString();

DateTime requestDate = dtpRequestDate.Value;

string responsible = txtResponsible.Text; // Новый параметр

string comments = txtComments.Text; // Новый параметр

string query;

NpgsqlParameter[] parameters;

// Если это новая заявка, формируем запрос на добавление

if (!requestId.HasValue)

{

query = @"

INSERT INTO ЗаявкиНаРемонт

(

Оборудование, ТипНеисправности, ОписаниеПроблемы, Клиент, Статус,

ДатаДобавления, Ответственный, Комментарии

)

VALUES

(

@Оборудование, @ТипНеисправности, @ОписаниеПроблемы, @Клиент, @Статус,

@ДатаДобавления, @Ответственный, @Комментарии

)";

parameters = new[]

{

new NpgsqlParameter("@Оборудование", equipment),

new NpgsqlParameter("@ТипНеисправности", faultType),

new NpgsqlParameter("@ОписаниеПроблемы", description),

new NpgsqlParameter("@Клиент", client),

new NpgsqlParameter("@Статус", status),

new NpgsqlParameter("@ДатаДобавления", requestDate),

new NpgsqlParameter("@Ответственный", responsible),

new NpgsqlParameter("@Комментарии", comments)

};

}

else

{

// Если это редактирование заявки, формируем запрос на обновление

query = @"

UPDATE ЗаявкиНаРемонт

SET

Оборудование = @Оборудование,

ТипНеисправности = @ТипНеисправности,

ОписаниеПроблемы = @ОписаниеПроблемы,

Клиент = @Клиент,

Статус = @Статус,

ДатаДобавления = @ДатаДобавления,

Ответственный = @Ответственный,

Комментарии = @Комментарии

WHERE Идентификатор = @Идентификатор";

parameters = new[]

{

new NpgsqlParameter("@Оборудование", equipment),

new NpgsqlParameter("@ТипНеисправности", faultType),

new NpgsqlParameter("@ОписаниеПроблемы", description),

new NpgsqlParameter("@Клиент", client),

new NpgsqlParameter("@Статус", status),

new NpgsqlParameter("@ДатаДобавления", requestDate),

new NpgsqlParameter("@Ответственный", responsible),

new NpgsqlParameter("@Комментарии", comments),

new NpgsqlParameter("@Идентификатор", requestId.Value)

};

}

// Выполняем запрос для сохранения данных в базе данных

try

{

dbHelper.ExecuteNonQuery(query, parameters); // Вставка или обновление данных в базе

MessageBox.Show("Заявка успешно сохранена!"); // Уведомление об успешном сохранении

this.DialogResult = DialogResult.OK; // Закрытие формы с возвратом результата

this.Close();

}

catch (Exception ex)

{

MessageBox.Show("Ошибка при сохранении заявки: " + ex.Message);

}

}

private void btnCancel\_Click(object sender, EventArgs e)

{

this.Close(); // Закрытие формы без сохранения

}

private void cmbStatus\_SelectedIndexChanged(object sender, EventArgs e)

{

}

}

}

**Фото программы**

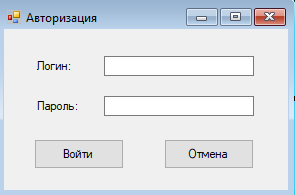


Рисунок 1

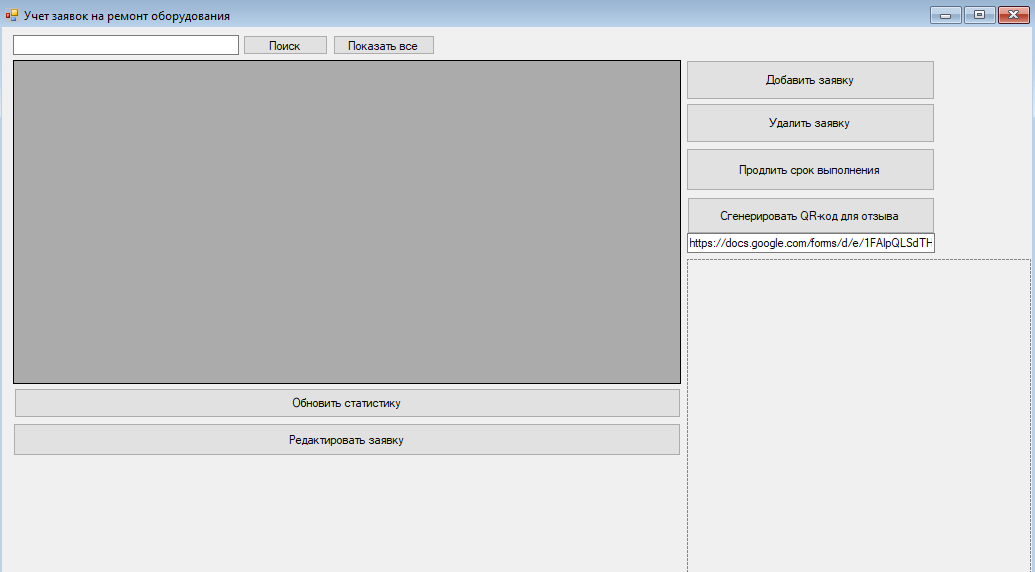


Рисунок 2

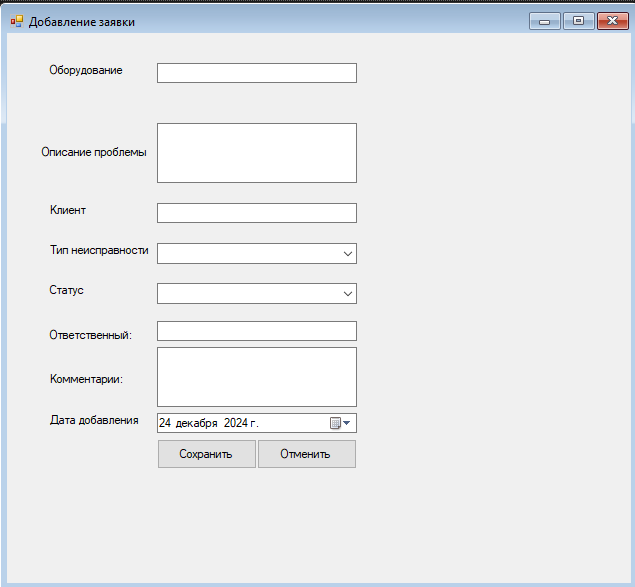


Рисунок 3

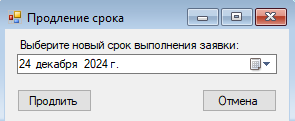
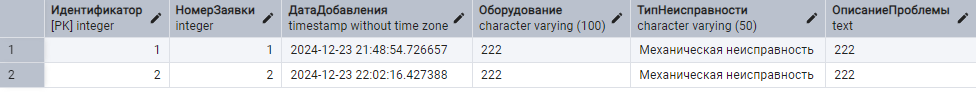


Рисунок 4

**Фото базы-данных**

Таблица ЗаявкиНаРемонт

****

Продолжение таблицы ****

Таблица Пользователи

****

Таблица Статистика



1. **Разработка модулей программного обеспечения для компьютерных систем**

**Техническое задание**   
 **Общие сведения**   
Наименование проекта: Разработка программного модуля для управления задачами внутри проектной команды.  
Заказчик: ООО "ПроектСервис".  
Исполнитель: Компания "IT Платформа".

* 1. **Возможность добавления задач с указанием параметров:**  
      - Номер задачи;  
      - Дата создания;  
      - Название проекта;  
      - Описание задачи;  
      - Приоритет задачи (низкий, средний, высокий);  
      - Исполнитель задачи;  
      - Статус выполнения (в ожидании, в работе, выполнено).

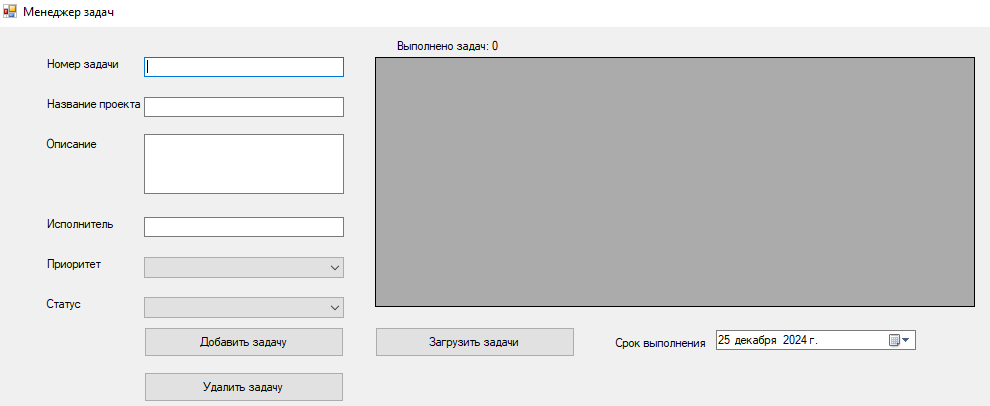


Рисунок 1 - добавления задач

**2. Возможность редактирования задач:**  
 - Изменение приоритета;  
 - Добавление/замена исполнителя;  
 - Изменение статуса выполнения;  
 - Корректировка описания задачи.



Рисунок 2 – Возможность редактирования задач

**private void btnUpdateTask\_Click(object sender, EventArgs e)**

**{**

**string taskNumber = txtTaskNumber.Text;**

**if (string.IsNullOrWhiteSpace(taskNumber))**

**{**

**MessageBox.Show("Пожалуйста, введите номер задачи для редактирования.");**

**return;**

**}**

**string projectName = txtProjectName.Text;**

**string description = txtDescription.Text;**

**string assignee = txtAssignee.Text;**

**string priority = cmbPriority.SelectedItem.ToString();**

**string status = cmbStatus.SelectedItem.ToString();**

**using (var conn = new NpgsqlConnection(connectionString))**

**{**

**try**

**{**

**conn.Open();**

**string query = "UPDATE tasks SET project\_name = @projectName, description = @description, assignee = @assignee, " +**

**"priority = @priority, status = @status WHERE task\_number = @taskNumber";**

**using (var cmd = new NpgsqlCommand(query, conn))**

**{**

**cmd.Parameters.AddWithValue("taskNumber", taskNumber);**

**cmd.Parameters.AddWithValue("projectName", projectName);**

**cmd.Parameters.AddWithValue("description", description);**

**cmd.Parameters.AddWithValue("assignee", assignee);**

**cmd.Parameters.AddWithValue("priority", priority);**

**cmd.Parameters.AddWithValue("status", status);**

**int rowsAffected = cmd.ExecuteNonQuery();**

**if (rowsAffected > 0)**

**{**

**MessageBox.Show("Задача успешно обновлена.");**

**LoadTasks();**

**}**

**else**

**{**

**MessageBox.Show("Задача с таким номером не найдена.");**

**}**

**}**

**}**

**catch (Exception ex)**

**{**

**MessageBox.Show($"Ошибка при редактировании задачи: {ex.Message}");**

**}**

**}**

**}**

**private void LoadTasks()**

**{**

**using (var conn = new NpgsqlConnection(connectionString))**

**{**

**try**

**{**

**conn.Open();**

**string query = "SELECT \* FROM tasks";**

**using (var cmd = new NpgsqlCommand(query, conn))**

**{**

**NpgsqlDataAdapter dataAdapter = new NpgsqlDataAdapter(cmd);**

**DataTable dataTable = new DataTable();**

**dataAdapter.Fill(dataTable);**

**dataGridViewTasks.DataSource = dataTable;**

**}**

**}**

**catch (Exception ex)**

**{**

**MessageBox.Show($"Ошибка при загрузке задач: {ex.Message}");**

**}**

**}**

**}**

**private void btnSearchTask\_Click(object sender, EventArgs e)**

**{**

**string searchQuery = txtSearch.Text;**

**using (var conn = new NpgsqlConnection(connectionString))**

**{**

**try**

**{**

**conn.Open();**

**string query = "SELECT \* FROM tasks WHERE task\_number ILIKE @searchQuery OR description ILIKE @searchQuery";**

**using (var cmd = new NpgsqlCommand(query, conn))**

**{**

**cmd.Parameters.AddWithValue("searchQuery", "%" + searchQuery + "%");**

**NpgsqlDataAdapter dataAdapter = new NpgsqlDataAdapter(cmd);**

**DataTable dataTable = new DataTable();**

**dataAdapter.Fill(dataTable);**

**dataGridViewTasks.DataSource = dataTable;**

**}**

**}**

**catch (Exception ex)**

**{**

**MessageBox.Show($"Ошибка при поиске задач: {ex.Message}");**

**}**

**}**

**}**

* 1. **Возможность отслеживания задач:**  
      - Просмотр списка задач с фильтрацией по статусу, исполнителю и приоритету;  
      - Поиск задачи по номеру или ключевым словам;  
      - Получение уведомлений о сроках выполнения и изменениях в задаче.

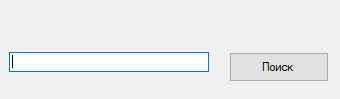


Рисунок 3 - Возможность отслеживания задач

private void btnSearchTask\_Click(object sender, EventArgs e)

{

string searchQuery = txtSearch.Text;

using (var conn = new NpgsqlConnection(connectionString))

{

try

{

conn.Open();

string query = "SELECT \* FROM tasks WHERE task\_number ILIKE @searchQuery OR description ILIKE @searchQuery";

using (var cmd = new NpgsqlCommand(query, conn))

{

cmd.Parameters.AddWithValue("searchQuery", "%" + searchQuery + "%");

NpgsqlDataAdapter dataAdapter = new NpgsqlDataAdapter(cmd);

DataTable dataTable = new DataTable();

dataAdapter.Fill(dataTable);

dataGridViewTasks.DataSource = dataTable;

}

}

catch (Exception ex)

{

MessageBox.Show($"Ошибка при поиске задач: {ex.Message}");

}

}

}

1. **Расчёт статистики работы:**  
    - Количество выполненных задач за указанный период;  
    - Среднее время выполнения задачи;  
    - Статистика по проектам и исполнителям.



Рисунок 4 - Расчёт статистики работы

**// 1. Статистика по выполненным задачам**

**private void btnStatCompletedTasks\_Click(object sender, EventArgs e)**

**{**

**try**

**{**

**string query = "SELECT COUNT(\*) FROM tasks WHERE status = 'Выполнено' AND due\_date BETWEEN @startDate AND @endDate";**

**using (var conn = new NpgsqlConnection(connectionString))**

**{**

**conn.Open();**

**using (var cmd = new NpgsqlCommand(query, conn))**

**{**

**cmd.Parameters.AddWithValue("startDate", dtpStartDate.Value);**

**cmd.Parameters.AddWithValue("endDate", dtpEndDate.Value);**

**var result = cmd.ExecuteScalar();**

**lblCompletedTasks.Text = $"Выполнено задач: {result}";**

**}**

**}**

**}**

**catch (Exception ex)**

**{**

**MessageBox.Show($"Ошибка при расчёте статистики выполненных задач: {ex.Message}");**

**}**

**}**

**// 2. Статистика по среднему времени выполнения задач**

**private void btnAvgCompletionTime\_Click(object sender, EventArgs e)**

**{**

**try**

**{**

**string query = "SELECT AVG(EXTRACT(EPOCH FROM (due\_date - created\_at)) / 3600) FROM tasks WHERE status = 'Выполнено' AND due\_date BETWEEN @startDate AND @endDate";**

**using (var conn = new NpgsqlConnection(connectionString))**

**{**

**conn.Open();**

**using (var cmd = new NpgsqlCommand(query, conn))**

**{**

**cmd.Parameters.AddWithValue("startDate", dtpStartDate.Value);**

**cmd.Parameters.AddWithValue("endDate", dtpEndDate.Value);**

**var result = cmd.ExecuteScalar();**

**//lblAvgCompletionTime.Text = $"Среднее время выполнения: {result} ч";**

**}**

**}**

**}**

**catch (Exception ex)**

**{**

**MessageBox.Show($"Ошибка при расчёте среднего времени выполнения: {ex.Message}");**

**}**

**}**

**// 3. Статистика по проектам и исполнителям**

**private void btnProjectAssigneeStats\_Click(object sender, EventArgs e)**

**{**

**try**

**{**

**string query = "SELECT project\_name, assignee, COUNT(\*) FROM tasks GROUP BY project\_name, assignee";**

**using (var conn = new NpgsqlConnection(connectionString))**

**{**

**conn.Open();**

**using (var cmd = new NpgsqlCommand(query, conn))**

**{**

**using (var reader = cmd.ExecuteReader())**

**{**

**lstProjectAssigneeStats.Items.Clear();**

**while (reader.Read())**

**{**

**string projectName = reader.GetString(0);**

**string assignee = reader.GetString(1);**

**int taskCount = reader.GetInt32(2);**

**lstProjectAssigneeStats.Items.Add($"{projectName} - {assignee}: {taskCount} задач");**

**}**

**}**

**}**

**}**

**}**

**catch (Exception ex)**

**{**

**MessageBox.Show($"Ошибка при расчёте статистики по проектам и исполнителям: {ex.Message}");**

**}**

**}**

1. **Нефункциональные требования**   
    - Кроссплатформенность (поддержка Windows и macOS).  
    - Уровни доступа пользователей (администратор, руководитель, сотрудник).  
    - Удобный и понятный интерфейс.  
    - Быстрое время отклика на действия пользователя.
2. **Требования к реализации**   
    - Язык программирования: на усмотрение разработчика.  
    - СУБД: PostgreSQL или MySQL.

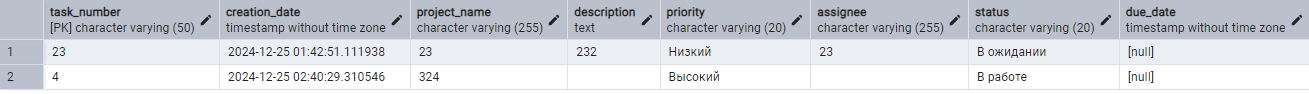


Рисунок 5 – База данных

CREATE TABLE tasks (

task\_number VARCHAR(50) PRIMARY KEY, -- Номер задачи

creation\_date TIMESTAMP DEFAULT CURRENT\_TIMESTAMP, -- Дата создания задачи

project\_name VARCHAR(255), -- Название проекта

description TEXT, -- Описание задачи

priority VARCHAR(20), -- Приоритет задачи (Низкий, Средний, Высокий)

assignee VARCHAR(255), -- Исполнитель задачи

status VARCHAR(20), -- Статус выполнения задачи (В ожидании, В работе, Выполнено)

due\_date TIMESTAMP -- Срок выполнения задачи

);

**Тестирование**

* 1. **Функциональное тестирование**   
      - Проверка корректности добавления задач с разными параметрами.

В моей программе реализовано добавление по удобному графическому интерфейсу

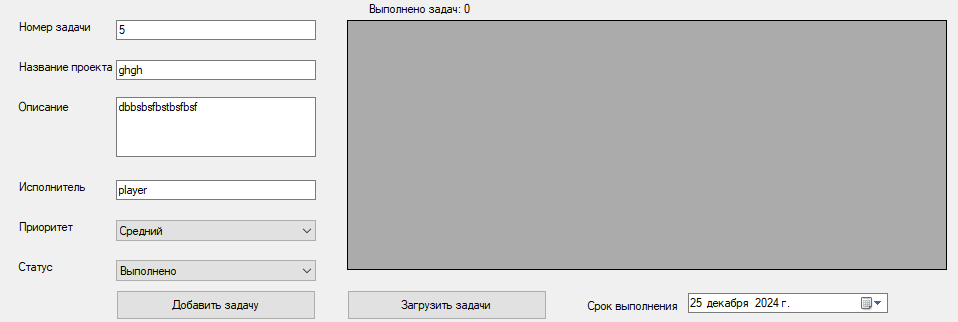


Рисунок 1 - Графический интерфейс

После ввода информации в специяльные ячейки нужно будет нажать на кнопку “Добавить задачу” после добовления пользователю придет уведомление о том что задача добавлена в бд

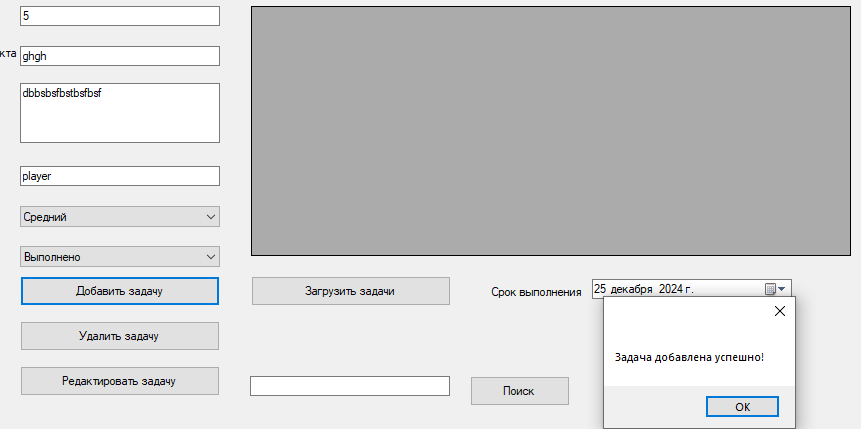


Рисунок 2 – Добавление задачи

После того как придет уведомления его можно будет закрыть, после этого можно использовать кнопку “Загрузить задачи” для отображения всех задач

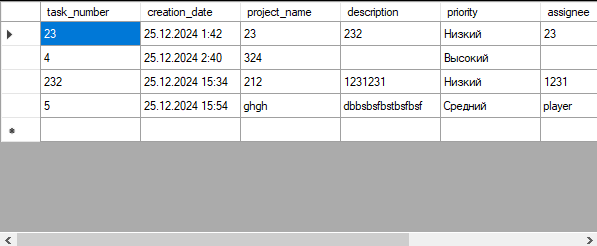


Рисунок 3 – Отображения всех задач

На рисунке выше можно увидеть что все задачи отобразятся корректно

- Тестирование изменения статусов задач.

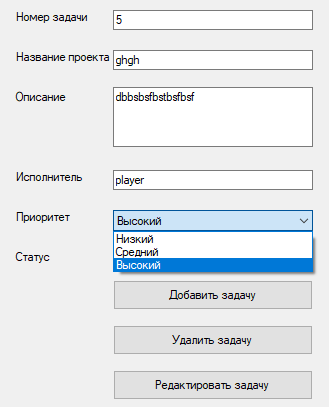


Рисунок 4 - Тестирование изменения статусов задач

После заполнения окна номера задачи мы можем вносить изменения в задаче, например поменять приоритет

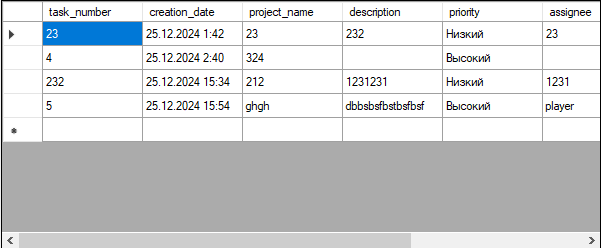


Рисунок 5 – Редактирования задач

На рисунке видно, что задание под номером 5 поменяло приоритет

- Проверка фильтрации и поиска задач по заданным параметрам.

После заполнения окна рядом с кнопкой “Поиск”, например 4 мы сможем использовать поиск всех задач

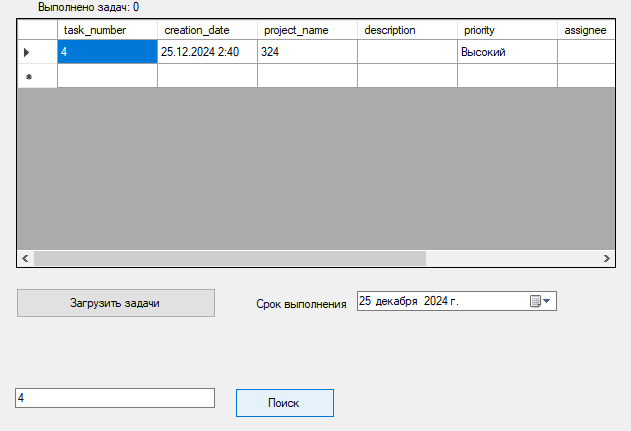


Рисунок 6 - Поиск

**2. Нагрузочное тестирование**

- Тестирование работы приложения при одновременном добавлении большого количества задач.

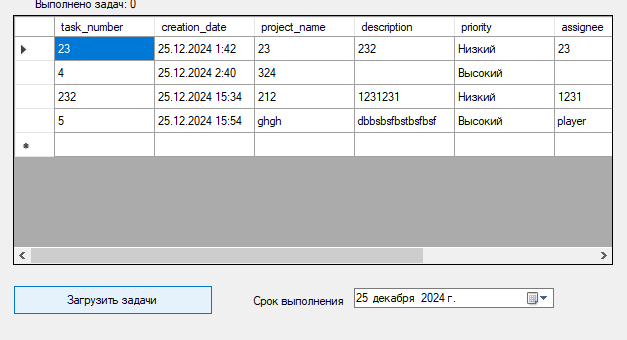


Рисунок 6 - Нагрузочное тестирование

После многократного нажатия кнопки “Загрузить задачи” пытаясь нагрузить программу и бд никаких сбоев я не обнаружен

- Измерение времени отклика системы при высоких нагрузках.

Время отклика перенагрузки не обнаружены

**3. Тестирование безопасности**

**4. UI тестирование**

Программа имеет интуитивно понятны интерфейс со всеми подписанными окнами

|  |  |  |  |
| --- | --- | --- | --- |
| Метод | Название метода | Функционал | Реализация |
| Добавить задачу | btnAddTask\_Click | Добавление задачи из бд | Копка добавляет задачу в бд |
| Удалить задачу | btnDeleteTask\_Click | Удаление задачи из бд | Кнопка удаляет задачу из бд |
| Редоктировать задачу | btnUpdateTask\_Click | - Изменение приоритета;  - Добавление/замена исполнителя;  - Изменение статуса выполнения;  - Корректировка описания задачи. | Редактирует задачи из бд |
| Поиск | btnSearchTask\_Click | - Просмотр списка задач с фильтрацией по статусу, исполнителю и приоритету;  - Поиск задачи по номеру или ключевым словам; | Исполняет подсчитать статистику |
| Посчитать статистику | buttonCalculateStatistics\_Click | Расчёт статистики работы:  - Количество выполненных задач за указанный период;  - Среднее время выполнения задачи;  - Статистика по проектам и исполнителям. | Выводит статистику задач |
| Сгенерировать QR код | btnGenerateQRCode\_Click | Генерация QR кода для оценки работы модуля (ввод данных через онлайн форму). | Генерирует QR код ссылку |

**5 Сопровождение и обслуживание программного обеспечения компьютерных систем**  
  
 **Функциональные обязанности сопровождающей команды**

**Улучшение пользовательского опыта**

- Сбор обратной связи от пользователей.  
 - Генерация QR кода для оценки работы модуля (ввод данных через онлайн форму).

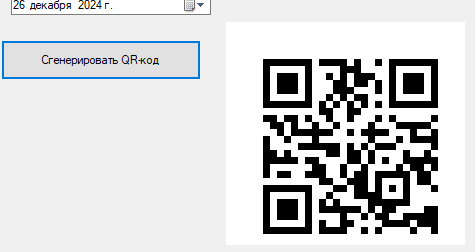


Рисунок 8 - графический интерфейс

Для того чтобы сгенерировать QR код нужно нажать на кнопку “Сгенерировать QR-код” после перехода по ссылке вы попадете на сайт где можете задать вопрос по моей программе

**Вывод**: Во время разработки программного модуля для управления задачами я работал с программой Winforms на основе языка C#, создавать графический интерфейс, а также подключать, создавать и писать SQL-запросы для базы данных построенной в программе pgAdmin 4.