

# **Лекция: Виртуальная и дополненная реальность. Язык C#.**

Лектор: Синицын Анатолий Васильевич

# Роль платформы .NET

- Поддержка нескольких языков
- Кроссплатформенность
- Мощная библиотека классов
- Разнообразие технологий
- Автоматическая сборка мусора

# Управляемый и неуправляемый код

Рисунок иллюстрирует процесс компиляции файлов с исходным кодом. Как видно из рисунка, исходный код программы может быть написан на любом языке, поддерживающем среду выполнения CLR. Затем соответствующий компилятор проверяет синтаксис и анализирует исходный код программы. Вне зависимости от типа используемого компилятора результатом компиляции будет являться управляемый модуль (managed module)— стандартный переносимый исполняемый (portable executable, PE) файл 32-разрядной (PE32) или 64-разрядной Windows (PE32+), который требует для своего выполнения CLR.



# Что такое .NET

При разработке платформы .NET Framework учитывались следующие цели:

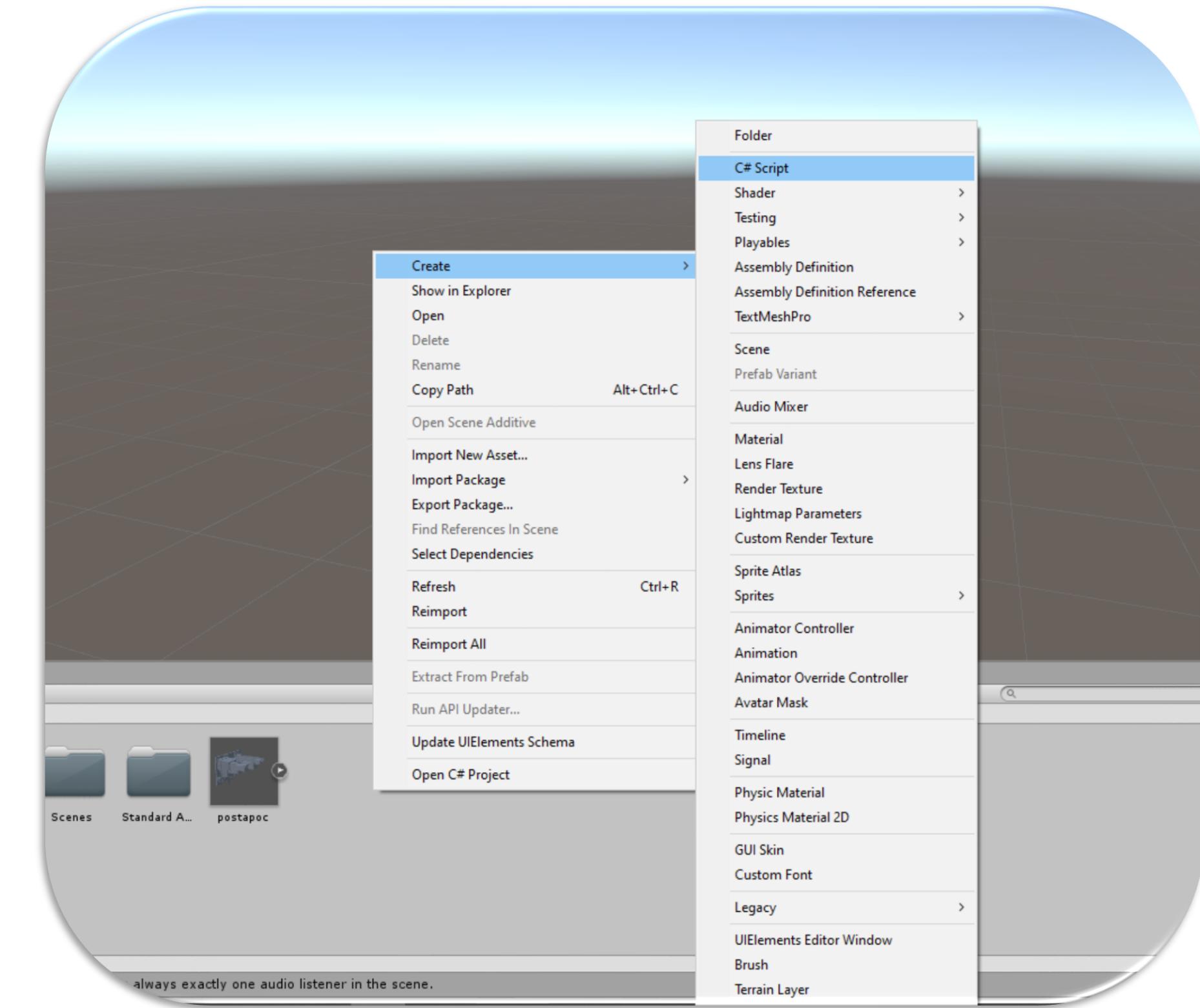
- Обеспечение среды выполнения кода, которая бы минимизировали конфликты при развертывании программного обеспечения и управлении версиями.
- Обеспечение объектно-ориентированной среды программирования для локального сохранения и выполнения объектного кода, либо для удаленного/распределенного выполнения.
- Предоставление среды выполнения кода, гарантирующей безопасное выполнение кода, включая код, созданный неизвестным или не полностью доверенным сторонним разработчиком.
- Предоставление единых принципов разработки для разных типов приложений, таких как приложения Windows и веб-приложения.
- Обеспечение среды выполнения кода, которая бы исключала проблемы с производительностью сред выполнения сценариев или интерпретируемого кода.
- Разработка взаимодействия на основе промышленных стандартов, что позволяет интегрировать код платформы .NET Framework с любым другим кодом

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

По сравнению с другими языками C# достаточно молодой, но в тоже время он уже прошел большой путь. Первая версия языка вышла вместе с релизом Microsoft Visual Studio .NET в феврале 2002 года. Текущей версией языка является версия C# 8.0, которая вышла в сентябре 2019 года вместе с релизом .NET Core 3.

C# является языком с Си-подобным синтаксисом и близок в этом отношении к C++ и Java.

Поведение игровых объектов контролируется с помощью компонентов (Components), которые присоединяются к ним. Несмотря на то, что встроенные компоненты Unity могут быть очень разносторонними, вскоре вы обнаружите, что вам нужно выйти за пределы их возможностей, чтобы реализовать ваши собственные особенности геймплея. Unity позволяет создавать свои компоненты, используя скрипты. Они позволяют активировать игровые события, изменять параметры компонентов, и отвечать на ввод пользователя каким вам угодно способом.



В отличии от других ассетов, скрипты обычно создаются непосредственно в Unity. Можно создать скрипт используя меню Create в левом верхнем углу панели Project или выбрав Assets > Create > C# Script (или JavaScript/Boo скрипт) в главном меню.

Новый скрипт будет создан в папке, которую вы выбрали в панели Project. Имя нового скрипта будет выделено, предлагая вам ввести новое имя.

Лучше ввести новое имя скрипта сразу после создания чем изменять его потом. Имя, которое вы введете будет использовано, чтобы создать начальный текст в скрипте, как описано ниже.

# Пример скрипта:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class cube : MonoBehaviour
{
    public GameObject obj;
    public Light myLight;
    private int numEnemis = 10;
    void Start()
    {
        myLight = GetComponent<Light>();
        for (int i = 0; i < numEnemis; i++)
            Debug.Log("Create " + i + " enemis!");
    }
}
```

```
void Update()
{
    if (Input.GetKeyUp(KeyCode.Space))
    {
        myLight.enabled = !myLight.enabled;
    }

    if (Input.GetKeyUp(KeyCode.A))
    {
        obj.SetActive(false);
    }

    if (Input.GetKeyUp(KeyCode.S))
    {
        Destroy(obj);
    }

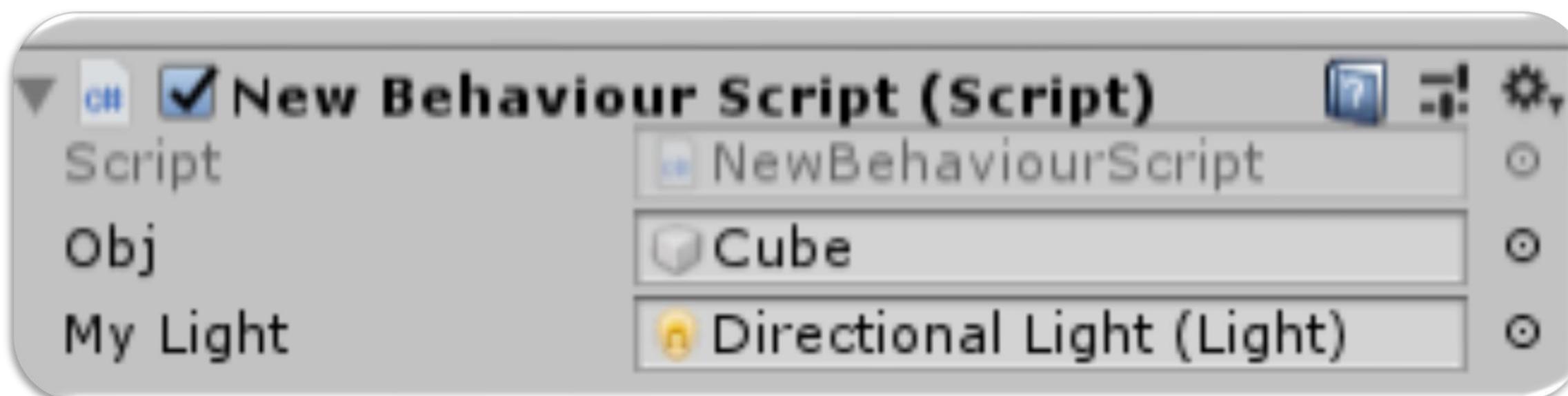
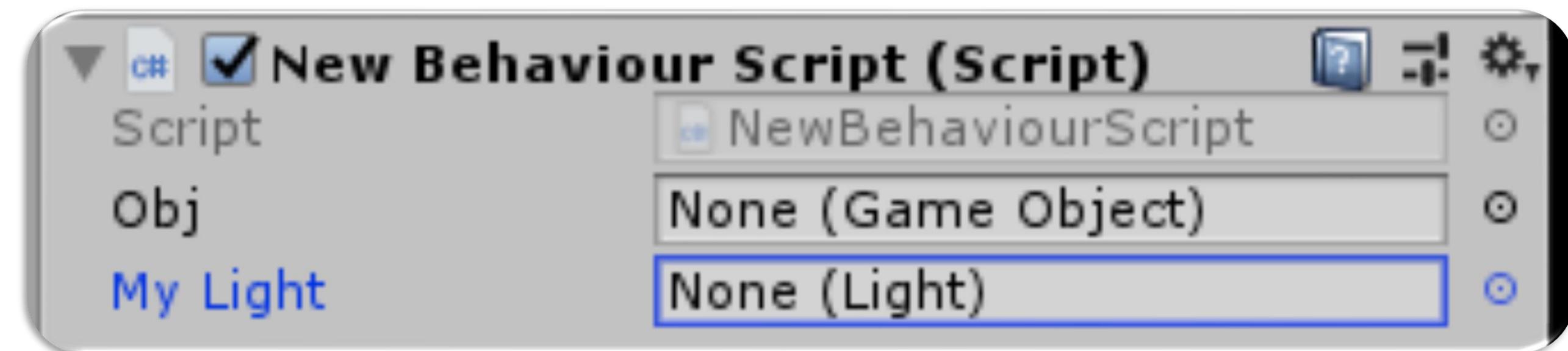
    if (Input.GetKeyUp(KeyCode.R))
        obj.GetComponent<Renderer>().material.color =
Color.red;
    else if (Input.GetKeyUp(KeyCode.G))
        obj.GetComponent<Renderer>().material.color =
Color.green;
    else if (Input.GetKeyUp(KeyCode.B))
        obj.GetComponent<Renderer>().material.color =
Color.blue;
}
```

Данный скрипт позволяет контролировать несколько игровых объектов:

- public GameObject obj
- public Light myLight

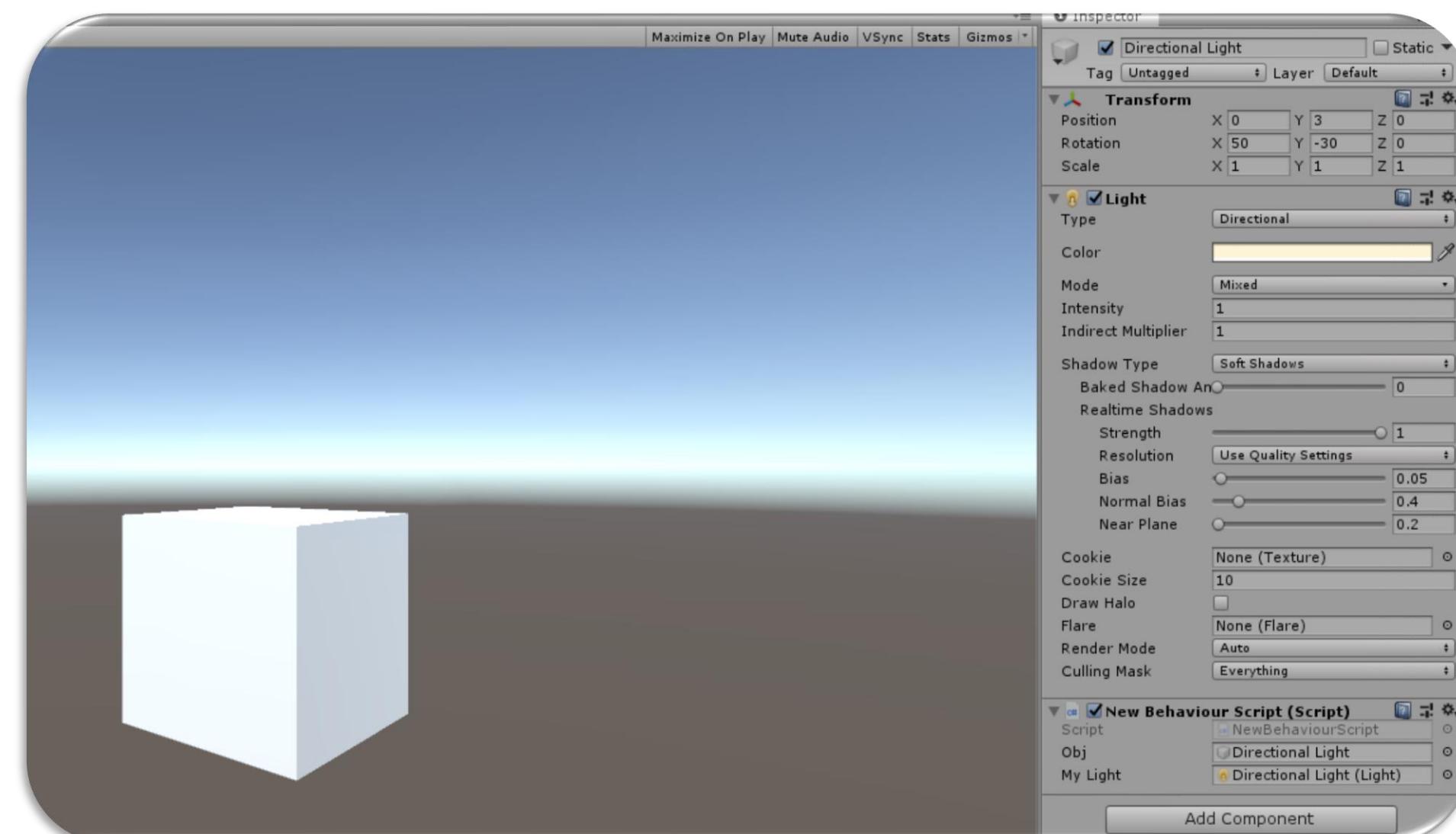
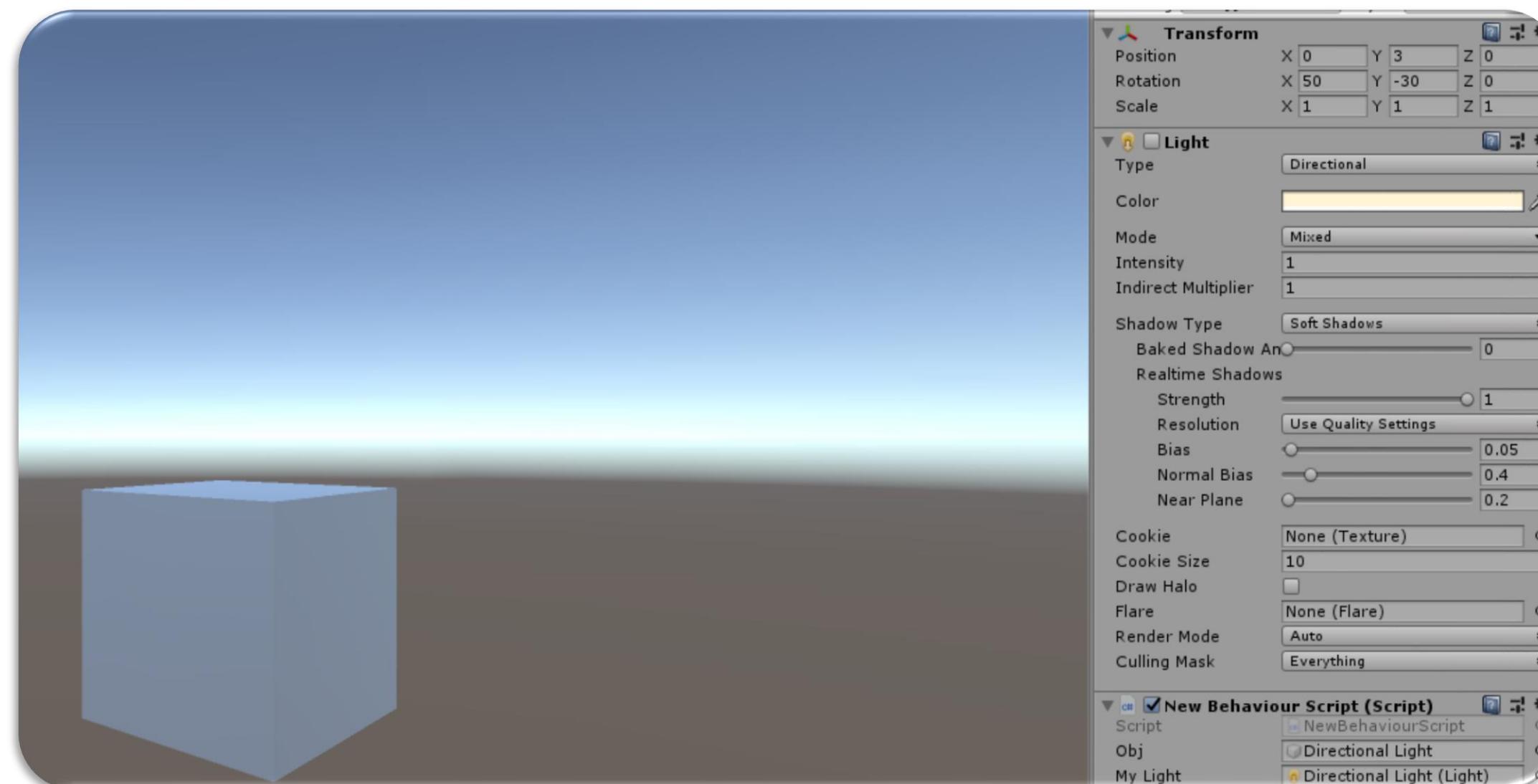
которые требуется просто перенести к меню скрипта.

В скрипте контролируется освещение и любой выбранный игровой объект (В данном случае куб)



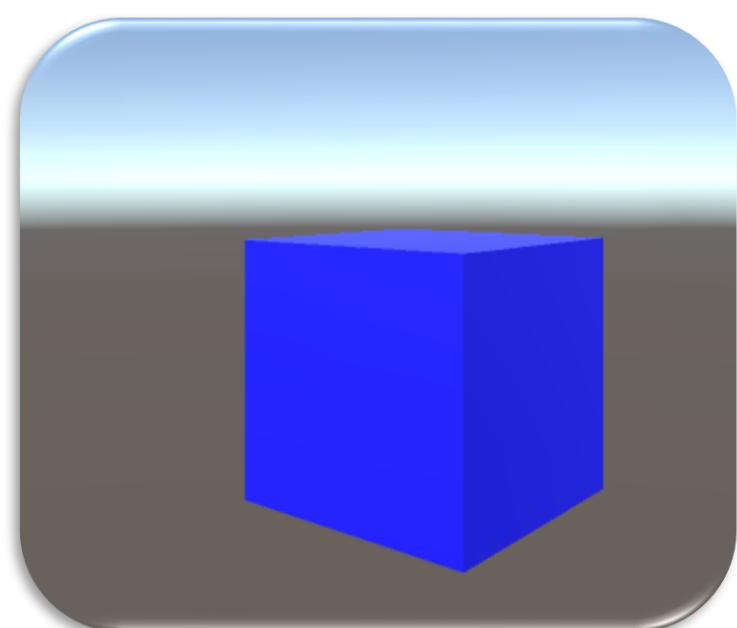
```
void Update()
{
if (Input.GetKeyUp(KeyCode.Space))
{
myLight.enabled = !myLight.enabled;
}
```

При нажатии клавиши Space свет будет включаться и выключаться



```
if (Input.GetKeyUp(KeyCode.R))  
obj.GetComponent<Renderer>().material.color = Color.red;  
else if (Input.GetKeyUp(KeyCode.G))  
obj.GetComponent<Renderer>().material.color = Color.green;  
else if (Input.GetKeyUp(KeyCode.B))  
obj.GetComponent<Renderer>().material.color = Color.blue;  
}  
}
```

При нажатии на клавиши  
**R G B** цвет объекта  
будет меняться на  
соответствующие цвета



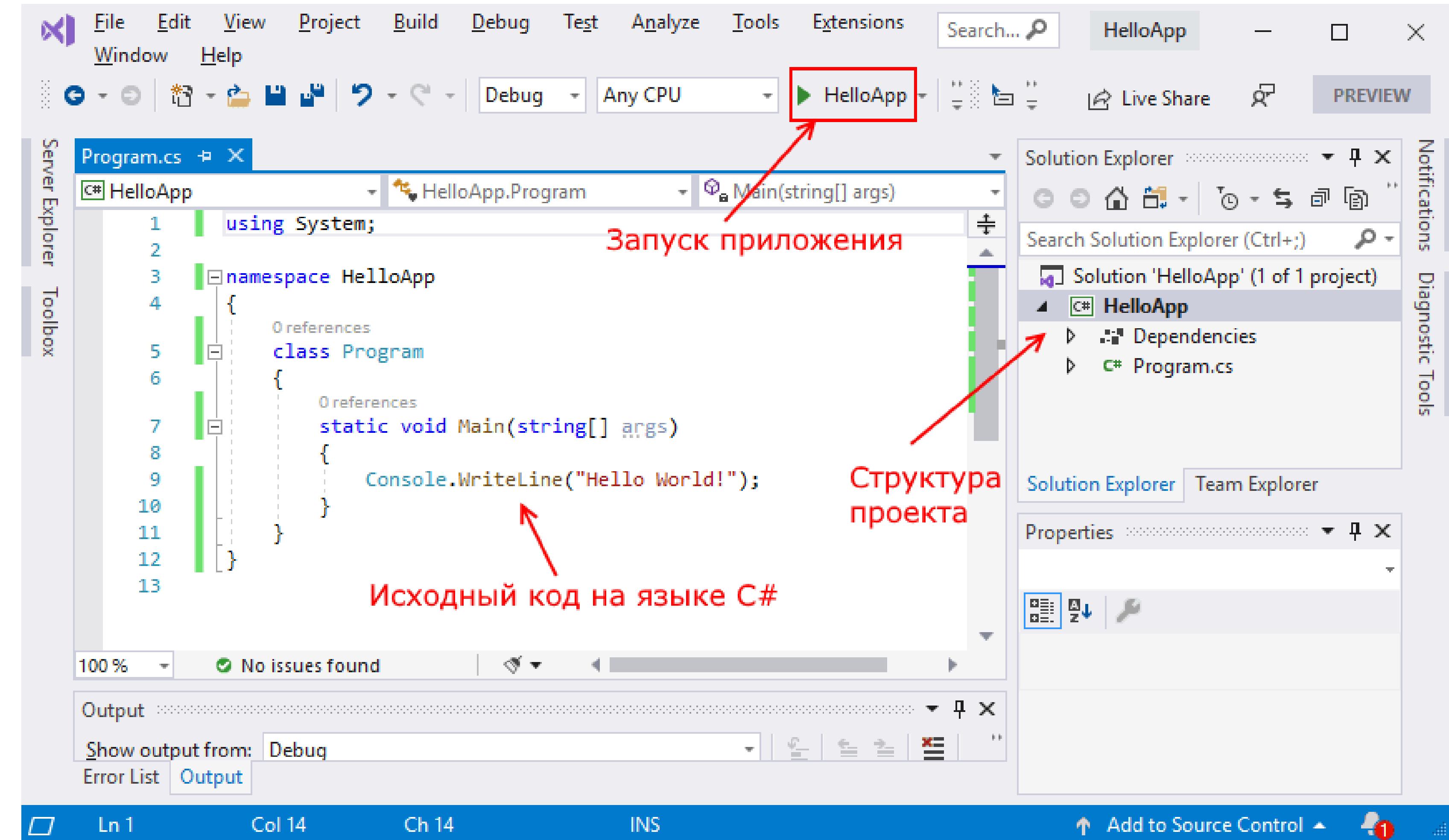
C# является объектно-ориентированным и в этом плане многое перенял у Java и C++. Например, C# поддерживает полиморфизм, наследование, перегрузку операторов, статическую типизацию.

Объектно-ориентированный подход позволяет решить задачи по построению крупных, но в тоже время гибких, масштабируемых и расширяемых приложений. И C# продолжает активно развиваться, и с каждой новой версией появляется все больше интересных функциональностей, как, например, лямбды, динамическое связывание, асинхронные методы и т.д.



# Создадим первое приложение на языке C#

Для создания приложений на C# можно использовать бесплатную и полнофункциональную среду разработки - Visual Studio Community.



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

Справа находится окно Solution Explorer, в котором можно увидеть структуру нашего проекта. В данном случае у нас сгенерированная по умолчанию структура: узел Properties или Свойств (он хранит файлы свойств приложения и пока нам не нужен); узел Dependencies - это узел содержит сборки dll, которые добавлены в проект по умолчанию. Эти сборки как раз содержат классы библиотеки .NET, которые будет использовать C#. Однако не всегда все сборки нужны.

Далее идет непосредственно сам файл кода программы Program.cs. Как раз этот файл и открыт в центральном окне. Вначале разберем, что весь этот код представляет:

```
using System; // подключаемое пространство имен

namespace HelloApp // объявление нового пространства имен
{
    class Program // объявление нового класса
    {
        static void Main(string[] args) // объявление нового метода
        {
            Console.WriteLine("Hello World!"); // действия метода

        } // конец объявления нового метода

    } // конец объявления нового класса

} // конец объявления нового пространства имен
```

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:

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

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:

```
{  
    Console.WriteLine("Hello");  
    Console.WriteLine("C#");  
}
```

С# является регистрозависимым языком.

# Метод Main

Точной входа в программу на языке C# является метод Main. При создании проекта консольного приложения в Visual Studio, например, создается следующий метод Main:

```
class Program
{
    static void Main(string[] args)
    {
        // здесь помещаются
        // выполняемые инструкции
    }
}
```

По умолчанию метод Main размещается в классе Program. Название класса может быть любым. Но метод Main является обязательной частью консольного приложения. Если мы изменим его название, то программа не скомпилируется.

По сути и класс, и метод представляют своего рода блок кода: блок метода помещается в блок класса. Внутри блока метода Main располагаются выполняемые в программе инструкции.

В языке C# есть следующие примитивные 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`

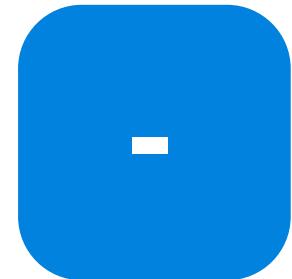
- int: хранит целое число от -2147483648 до 2147483647 и занимает 4 байта. Представлен системным типом System.Int32. Все целочисленные литералы по умолчанию представляют значения типа int:
- uint: хранит целое число от 0 до 4294967295 и занимает 4 байта. Представлен системным типом System.UInt32
- long: 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  $-3.4 \times 10^{38}$  до  $3.4 \times 10^{38}$  и занимает 4 байта. Представлен системным типом System.Single
- double: хранит число с плавающей точкой от  $\pm 5.0 \times 10^{-324}$  до  $\pm 1.7 \times 10^{308}$  и занимает 8 байта. Представлен системным типом System.Double

- decimal: хранит десятичное дробное число. Если употребляется без десятичной запятой, имеет значение от  $\pm 1.0 \cdot 10^{-28}$  до  $\pm 7.9228 \cdot 10^{28}$ , 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 символьные 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.

# Арифметические операторы



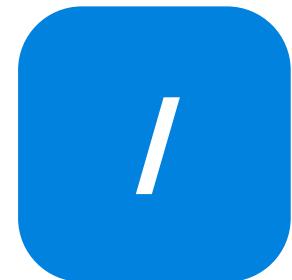
Сложение



Вычитание



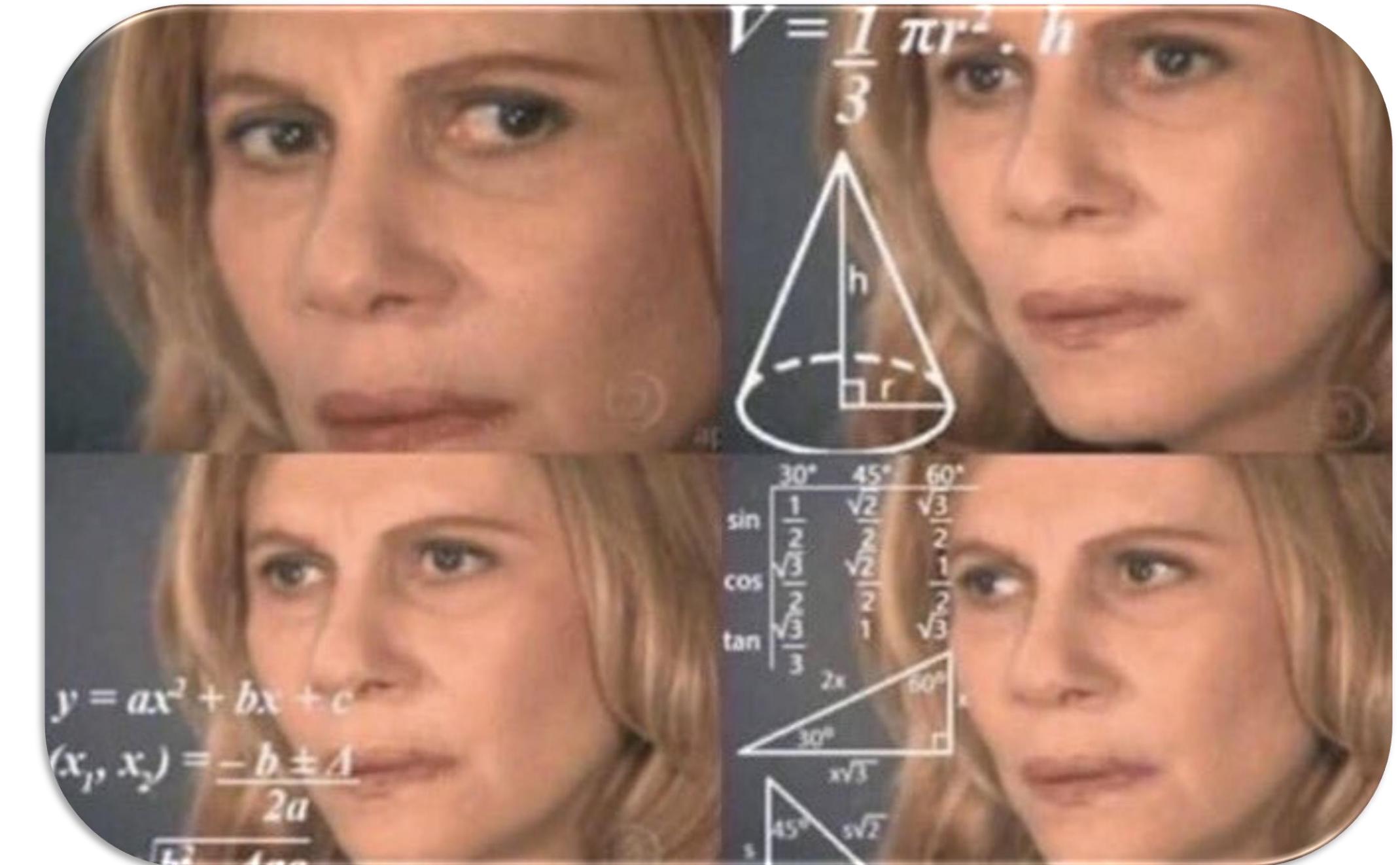
Умножение



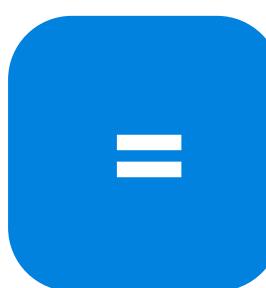
Деление



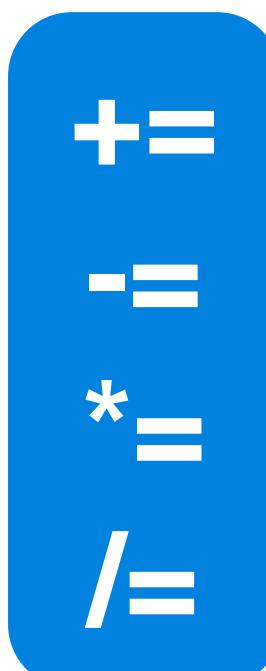
Деление по модулю



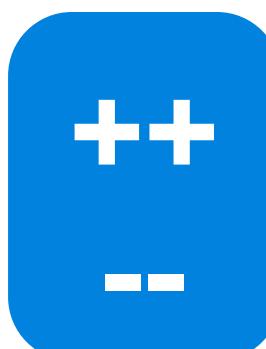
# Операторы присваивания



Присваивает значение переменной справа переменной слева

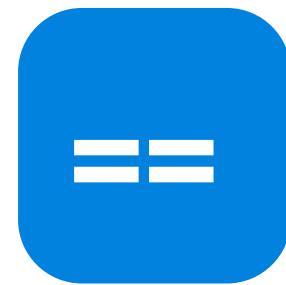


Сокращенный оператор присваивания, который выполняет некоторую арифметическую операцию, в зависимости от используемого символа, а затем присваивает полученный результат переменной слева  $x=x+5 = x+=5$

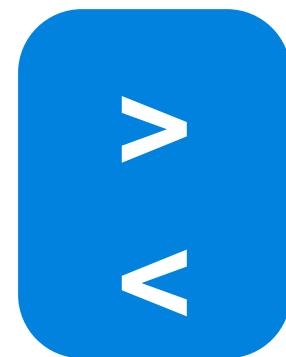


Сокращенные операторы инкремента и декремента. Увеличивают или уменьшают число на 1

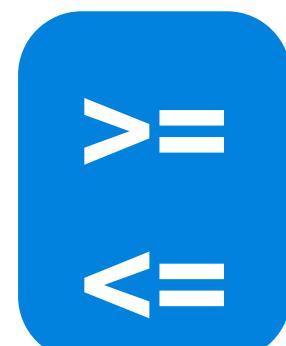
# Операторы сравнения



Этот оператор возвращает истину, только в том случае, если оба значения равны



Операторы «больше» и «меньше» возвращают истину только в том случае, если одно из значений больше или меньше



Операторы «больше или равно» и «меньше или равно» возвращают истину только в том случае, если одно из значений больше либо равно или меньше либо равно



Оператор «не равно» возвращает истину, если оба значения не равны

# Логические операторы

&&

Оператор «И», сравнивает два логических значения и возвращает истину только в том случае если оба истины

||

Оператор «Или», сравнивает два логических значения и определяет, истина ли оба. Если одно или оба значения истинны, этот оператор возвращает истину.

!

Оператор «Не», возвращает противоположное логическое значение.

# Условные операторы

Условные конструкции - один из базовых компонентов многих языков программирования, которые направляют работу программы по одному из путей в зависимости от определенных условий.

В языке C# используются следующие условные конструкции: if..else и switch..case

## Конструкция if/else

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

```
int num1 = 8;  
int num2 = 6;  
if (num1 > num2)  
{  
    Console.WriteLine($"Число {num1} больше числа {num2}");  
}
```

После ключевого слова `if` ставится условие. И если это условие выполняется, то срабатывает код, который помещен далее в блоке `if` после фигурных скобок. В качестве условий выступают ранее рассмотренные операции сравнения.

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

Чтобы при несоблюдении условия также выполнялись какие-либо действия требуется добавить блок `else`:

```
int num1 = 8;  
int num2 = 6;  
if (num1 > num2)  
{  
    Console.WriteLine($"Число {num1} больше числа {num2}");  
}  
else  
{  
    Console.WriteLine($"Число {num1} меньше числа {num2}");  
}
```

при сравнении чисел можно насчитать три состояния: первое число больше второго, первое число меньше второго и числа равны. Используя конструкцию `else if`, можно обрабатывать дополнительные условия:

```
int num1 = 8;  
int num2 = 6;  
if (num1 > num2)  
{  
    Console.WriteLine($"Число {num1} больше числа {num2}");  
}  
else if (num1 < num2)  
{  
    Console.WriteLine($"Число {num1} меньше числа {num2}");  
}  
else  
{  
    Console.WriteLine("Число num1 равно числу num2");  
}
```

можно соединить сразу несколько условий, используя логические операторы:

```
int num1 = 8;  
int num2 = 6;  
if(num1 > num2 && num1==8)  
{  
    Console.WriteLine($"Число {num1} больше числа {num2}");  
}
```

# Конструкция **switch**

Конструкция **switch/case** аналогична конструкции **if/else**, так как позволяет обработать сразу несколько условий:

```
Console.WriteLine("Нажмите Y или N");
string selection = Console.ReadLine();
switch (selection)
{
    case "Y":
        Console.WriteLine("Вы нажали букву Y");
        break;
    case "N":
        Console.WriteLine("Вы нажали букву N");
        break;
    default:
        Console.WriteLine("Вы нажали неизвестную букву");
        break;
}
```

```
int number = 1;
switch (number)
{
    case 1:
        Console.WriteLine("case 1");
        goto case 5; // переход к
    case 5
    case 3:
        Console.WriteLine("case 3");
        break;
    case 5:
        Console.WriteLine("case 5");
        break;
    default:
        Console.WriteLine("default")
;
        break;
}
```

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.

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

Если мы хотим также обработать ситуацию, когда совпадения не будет найдено, то можно добавить блок default.

Применение оператора return позволит выйти не только из блока case, но и из вызывающего метода. То есть, если в методе Main после конструкции switch..case, в которой используется оператор return, идут какие-либо операторы и выражения, то они выполнятся не будут, а метод Main завершит работу.

# 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:

```
for ([инициализация счетчика]; [условие]; [изменение счетчика])
{
    // действия
}
```

## Рассмотрим стандартный цикл for:

```
for (int i = 0; i < 9; i++) {  
    Console.WriteLine($"Квадрат числа {i} равен {i*i}");  
}
```

Первая часть объявления цикла - `int i = 0` - создает и инициализирует счетчик `i`. Счетчик необязательно должен представлять тип `int`. Это может быть и другой числовой тип, например, `float`. И перед выполнением цикла его значение будет равно 0. В данном случае это то же самое, что и объявление переменной.

Вторая часть - условие, при котором будет выполняться цикл. Пока условное выражение возвращает `true`, будет выполняться цикл. В данном случае цикл будет выполняться, пока счетчик `i` не достигнет 9.

И третья часть - приращение счетчика на единицу. Опять же нам необязательно увеличивать на единицу. Можно уменьшать: `i--`.

В итоге блок цикла сработает 9 раз, пока значение `i` не станет равным 9. И каждый раз это значение будет увеличиваться на 1.

# Цикл do

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

```
int i = 6;  
do  
{  
    Console.WriteLine(i);  
    i--;  
}  
while (i > 0);
```

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

То есть можно написать:

```
int i = -1;  
do  
{  
    Console.WriteLine(i);  
    i--;  
}  
while (i > 0);
```

Хотя 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:

```
int i = 6;  
while (i > 0)  
{  
    Console.WriteLine(i);  
    i--;  
}
```

# 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:

```
for (int i = 0; i < 9; i++)
{
    if (i == 5)
        break;
    Console.WriteLine(i);
}
```



0  
1  
2  
3  
4

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. Для того чтобы при проверке цикл не завершался, а просто пропускал текущую итерацию.

Можно воспользоваться оператором `continue`:

Можно воспользоваться оператором continue:

```
for (int i = 0; i < 9; i++)  
{  
    if (i == 5)  
        continue;  
    Console.WriteLine(i);  
}
```

0  
1  
2  
3  
4  
6  
7  
8

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

# Массивы

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

тип\_переменной[] название\_массива;

Определим массив целых чисел

```
int[] nums = new int[4];
```

вначале был объявлен массив `nums`, который будет хранить данные типа `int`. Далее используя операцию `new`, была выделена память для 4 элементов массива: `new int[4]`.

Число 4 еще называется длиной массива. При таком определении все элементы получают значение по умолчанию, которое предусмотрено для их типа. Для типа `int` значение по умолчанию - 0.

# Также можно сразу указать значения для этих элементов:

```
int[] nums2 = new int[4] { 1, 2, 3, 5 };  
int[] nums3 = new int[] { 1, 2, 3, 5 };  
int[] nums4 = new[] { 1, 2, 3, 5 };  
int[] nums5 = { 1, 2, 3, 5 };
```

Все перечисленные выше способы будут равноценны.

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. А чтобы обратиться к четвертому элементу в массиве, нам надо использовать индекс 3, к примеру: `nums[3]`. Используем индексы для получения и установки значений элементов массива:

```
int[] nums = new int[4];  
nums[0] = 1;  
nums[1] = 2;  
nums[2] = 3;  
nums[3] = 5;  
Console.WriteLine(nums[3]); // 5
```

массив определен только для 4 элементов, то мы не можем обратиться, например, к шестому элементу: `nums[5] = 5;`. Если мы так попытаемся сделать, то мы получим исключение `IndexOutOfRangeException`.

# Перебор массивов. Цикл foreach

Цикл foreach предназначен для перебора элементов в контейнерах, в том числе в массивах. Формальное объявление цикла foreach:

foreach (тип\_данных название\_переменной in контейнер)

```
{  
    // действия  
}
```

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:

Например:

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

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

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

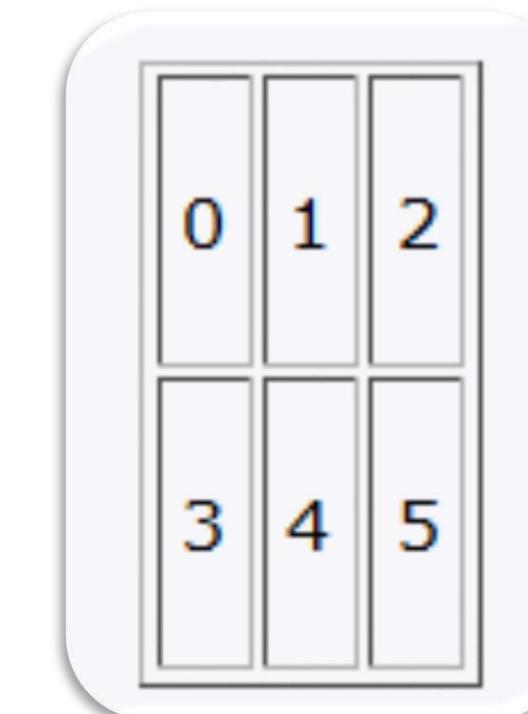
```
int[] numbers = new int[] { 1, 2, 3, 4, 5 };
for (int i = 0; i < numbers.Length; i++)
{
    numbers[i] = numbers[i] * 2;
    Console.WriteLine(numbers[i]);
}
```

Визуально оба массива можно представить следующим образом:

## Одномерный массив nums1



## Двухмерный массив nums2



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

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

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

```
int[, , ] nums3 = new int[2, 3, 4];
```

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

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

```
int[,] mas = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 }, { 10, 11, 12 } };  
foreach (int i in mas)  
    Console.WriteLine($"{i}");  
Console.WriteLine();
```

В данном случае длина массива mas равна 12. И цикл foreach выводит все элементы массива в строку:



1 2 3 4 5 6 7 8 9 10 11 12

При необходимости взаимодействовать с отдельной строкой в каждой таблице требуется получить количество элементов в размерности. В частности, у каждого массива есть метод `GetUpperBound(dimension)`, который возвращает индекс последнего элемента в определенной размерности. И если мы говорим непосредственно о двухмерном массиве, то первая размерность (с индексом 0) по сути это и есть таблица. И с помощью выражения `mas.GetUpperBound(0) + 1` можно получить количество строк таблицы, представленной двухмерным массивом. А через `mas.Length / rows` можно получить количество элементов в каждой строке:

```
int[,] mas = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 }, { 10, 11, 12 } };
```

```
int rows = mas.GetUpperBound(0) + 1;
int columns = mas.Length / rows;
// или так
// int columns = mas.GetUpperBound(1) + 1;

for (int i = 0; i < rows; i++)
{
    for (int j = 0; j < columns; j++)
    {
        Console.Write($"{mas[i, j]} \t");
    }
    Console.WriteLine();
}
```

1	2	3
4	5	6
7	8	9
10	11	12

# Массив массивов

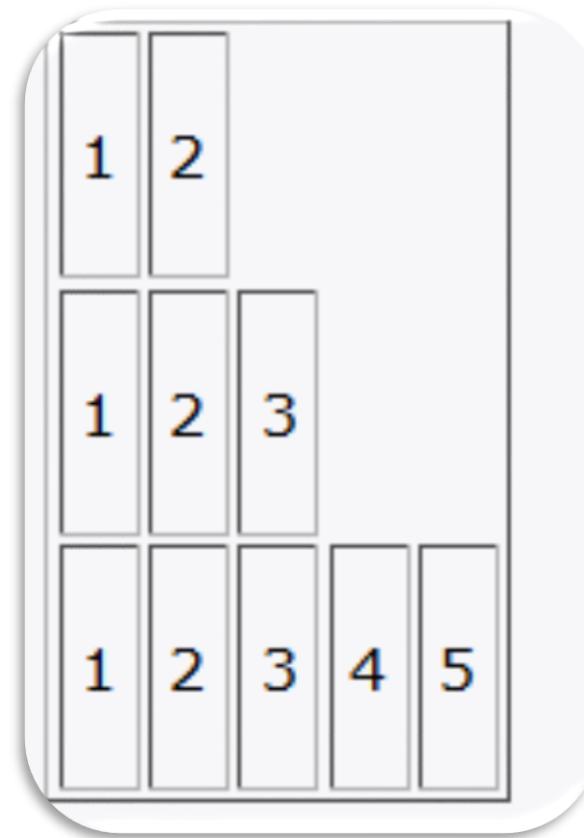
От многомерных массивов надо отличать массив массивов или так называемый "зубчатый массив":

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

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

# Зубчатый массив `nums`

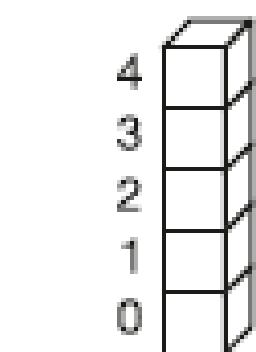
Примеры массивов:



## Основные понятия массивов

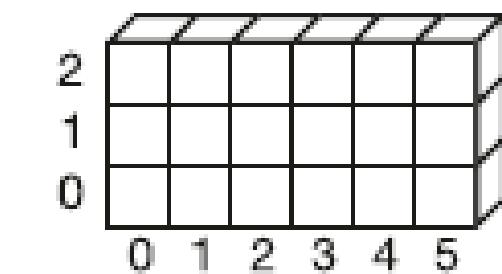
Суммирую основные понятия массивов:

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

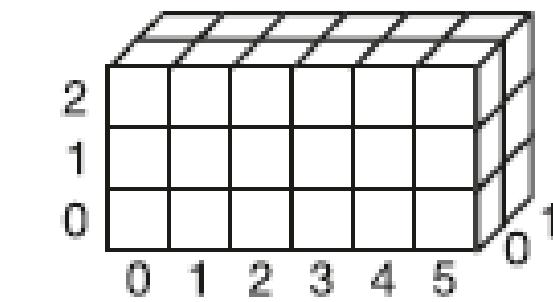


Одномерный массив  
`int[5]`

Многомерные массивы

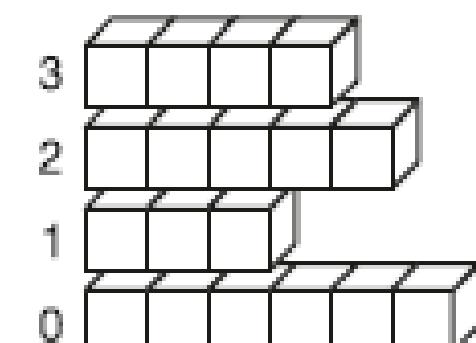


Двухмерный массив  
`int[3,6]`



Трехмерный массив  
`int[3,6,2]`

Зубчатый массив



Зубчатый массив  
`int[4][]`

# Методы

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

Общее определение методов выглядит следующим образом:

```
[модификаторы] тип_возвращаемого_значения  
название_метода ( [параметры] )  
{  
    // тело метода  
}
```

Модификаторы и параметры необязательны.

Например, по умолчанию консольная программа на языке C# должна содержать как минимум один метод - метод Main, который является точкой входа в приложение:

```
static void Main(string[] args)
{
}
```

Ключевое слово static является модификатором. Далее идет тип возвращаемого значения. В данном случае ключевое слово void указывает на то, что метод ничего не возвращает.

Далее идет название метода - Main и в скобках параметры - string[] args. И в фигурные скобки заключено тело метода - все действия, которые он выполняет. В данном случае метод Main пуст, он не содержит никаких операторов и по сути ничего не выполняет.

Определим еще пару методов:

```
namespace HelloApp
{
    class Program
    {
        static void Main(string[] args)
        {
        }

        static void SayHello()
        {
            Console.WriteLine("Hello");
        }

        static void SayGoodbye()
        {
            Console.WriteLine("GoodBye");
        }
    }
}
```

В данном случае определены еще два метода: `SayHello` и `SayGoodbye`. Оба метода определены в рамках класса `Program`, они имеют модификатор `static`, а в качестве возвращаемого типа для них определен тип `void`. То есть данные методы ничего не возвращают, просто производят некоторые действия. И также оба метода не имеют никаких параметров, поэтому после названия метода указаны пустые скобки.

Оба метода выводят на консоль некоторую строку. Причем для вывода на консоль методы используют другой метод, который определен в .NET по умолчанию - `Console.WriteLine()`.

Но если мы запустим данную программу, то мы не увидим никаких сообщений, которые должны выводить методы `SayHello` и `SayGoodbye`. Потому что стартовой точкой является метод `Main`. При запуске программы выполняется только метод `Main` и все операторы, которые составляют тело этого метода. Все остальные методы не выполняются.

Для вызова метода указывается его имя, после которого в скобках идут значения для его параметров (если метод принимает параметры).

название\_метода (значения\_для\_параметров\_метода) ;

```
using System;  
namespace HelloApp  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            SayHello();  
            SayGoodbye();  
            Console.ReadKey();  
        }  
        static void SayHello()  
        {  
            Console.WriteLine("Hello");  
        }  
        static void SayGoodbye()  
        {  
            Console.WriteLine("GoodBye");  
        }  
    }  
}
```

Например, вызовем методы `SayHello` и `SayGoodbye`:  
Преимуществом методов является то, что их можно повторно и многократно вызывать в различных частях программы. Например, в примере выше в двух методах для вывода строки на консоль используется метод `Console.WriteLine`.

# Возвращение значения

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.

Если метод имеет любой другой тип, отличный от void, то такой метод обязан вернуть значение этого типа. Для этого применяется оператор return, после которого идет возвращаемое значение:

return возвращаемое значение;

Например, определим еще пару методов:

```
static string GetHello()
{
    return "Hello";
}
static int GetSum()
{
    int x = 2;
    int y = 3;
    int z = x + y;
    return z;
}
```

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

Метод GetSum 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`).

После оператора `return` также можно указывать сложные выражения, которые возвращают определенный результат.

```
using System;
namespace HelloApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string message = GetHello();
            int sum = GetSum();

            Console.WriteLine(message); // Hello
            Console.WriteLine(sum); // 5
            Console.ReadKey();
        }

        static string GetHello()
        {
            return "Hello";
        }

        static int GetSum()
        {
            int x = 2;
            int y = 3;
            return x + y;
        }
    }
}
```

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

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

Второй метод - GetSum - возвращает значение типа int, поэтому его можно присвоить переменной, которая принимает значение этого типа:  
int sum = GetSum();.

# Выход из метода

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

Например:

```
static string GetHello()
{
    return "Hello";
    Console.WriteLine("After return");
}
```

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

Однако можно 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 определённых условий произвести выход из метода:

```
static void SayHello()
{
    int hour = 23;
    if (hour > 22)
    {
        return;
    }
    else
    {
        Console.WriteLine("Hello");
    }
}
```

# Сокращенная запись методов

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

```
static void SayHello()  
{  
    Console.WriteLine("Hello");  
}
```

Его можно сократить следующим образом:

```
static void SayHello() => Console.WriteLine("Hello");
```

То есть после списка параметров ставится знак равно и больше чем, после которого идет выполняемая инструкция.

# Параметры методов

Параметры позволяют передать в метод некоторые входные данные.  
Например, определим метод, который складывает два числа:

```
static int Sum(int x, int y)
{
    return x + y;
}
```

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.

```
class Program
{
    static void Main(string[] args)
    {
        int result = Sum(10, 15);
        Console.WriteLine(result);
        Console.ReadKey();
    }

    static int Sum(int x, int y)
    {
        return x + y;
    }
}
```

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. Значения, которые передаются параметрам, еще называются аргументами. То есть передаваемые числа 10 и 15 в данном случае являются аргументами.

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

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

```
class Program
{
    static void Main(string[] args)
    {
        int a = 25;
        int b = 35;
        int result = Sum(a, b);
        Console.WriteLine(result); // 60
        result = Sum(b, 45);
        Console.WriteLine(result); // 80
        result = Sum(a + b + 12, 18); // "a + b + 12" представляет значение параметра x
        Console.WriteLine(result); // 90
        Console.ReadKey();
    }
    static int Sum(int x, int y)
    {
        return x + y;
    }
}
```

# Кортежи

Кортежи предоставляют удобный способ для работы с набором значений, который был добавлен в версии C# 7.0.

Кортеж представляет набор значений, заключенных в круглые скобки:

```
var tuple = (5, 10);
```

В данном случае определен кортеж `tuple`, который имеет два значения: 5 и 10. В дальнейшем мы можем обращаться к каждому из этих значений через поля с названиями `Item[порядковый_номер_поля_в_кортеже]`.

Например:

```
static void Main(string[] args) {  
    var tuple = (5, 10);  
    Console.WriteLine(tuple.Item1); // 5  
    Console.WriteLine(tuple.Item2); // 10  
    tuple.Item1 += 26;  
    Console.WriteLine(tuple.Item1); // 31  
    Console.Read();  
}
```

# Классы. Объектно-ориентированное программирование. Классы и объекты

```
1 class Person
2 {
3
4 }
```

```
1 using System;
2
3 namespace HelloApp
4 {
5     class Person
6     {
7
8     }
9     class Program
10    {
11         static void Main(string[] args)
12         {
13
14         }
15     }
16 }
```

Вся функциональность класса представлена его членами - полями (полями называются переменные класса), свойствами, методами, событиями. Например, определим в классе Person поля и метод:

```
1 using System;
2
3 namespace HelloApp
4 {
5     class Person
6     {
7         public string name; // имя
8         public int age = 18; // возраст
9
10    public void GetInfo()
11    {
12        Console.WriteLine($"Имя: {name} Возраст: {age}");
13    }
14 }
15 class Program
16 {
17     static void Main(string[] args)
18     {
19         Person tom;
20     }
}
```

Для создания объекта Person используется выражение new Person(). Оператор new выделяет память для объекта Person. И затем вызывается конструктор по умолчанию, который не принимает никаких параметров. 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.

```
1 class Person
2 {
3     public string name; // Имя
4     public int age;    // возраст
5
6     public void GetInfo()
7     {
8         Console.WriteLine($"Имя: {name} Возраст: {age}");
9     }
10}
11class Program
12{
13    static void Main(string[] args)
14    {
15        Person tom = new Person();
16        tom.GetInfo();      // Имя: Возраст: 0
17
18        tom.name = "Tom";
19        tom.age = 34;
20        tom.GetInfo();    // Имя: Том Возраст: 34
21        Console.ReadKey();

```

Если конструктор не инициализирует значения переменных объекта, то они получают значения по умолчанию. Для переменных числовых типов это число 0, а для типа string и классов - это значение null (то есть фактически отсутствие значения).

После создания объекта мы можем обратиться к переменным объекта Person через переменную tom и установить или получить их значения, например, tom.name = "Tom";.

Консольный вывод данной программы:

Имя: Возраст: 0

Имя: Том Возраст: 34

# Создание конструкторов

Выше для инициализации объекта использовался конструктор по умолчанию. Однако мы сами можем определить конструктор:

```
1 class Person
2 {
3     public string name;
4     public int age;
5
6     public Person() { name = "Неизвестно"; age = 18; }      // 1 конструктор
7
8     public Person(string n) { name = n; age = 18; }          // 2 конструктор
9
10    public Person(string n, int a) { name = n; age = a; }    // 3 конструктор
11
12    public void GetInfo()
13    {
14        Console.WriteLine($"Имя: {name} Возраст: {age}");
15    }
16 }
```

Консольный вывод данной программы:

```
Имя: Неизвестно Возраст: 18
```

```
Имя: Bob Возраст: 18
```

```
Имя: Sam Возраст: 25
```

Теперь в классе определено три конструктора, каждый из которых принимает различное количество параметров и устанавливает значения полей класса. Используем эти конструкторы:

```
1 static void Main(string[] args)
2 {
3     Person tom = new Person();           // вызов 1-ого конструктора без параметров
4     Person bob = new Person("Bob");      //вызов 2-ого конструктора с одним параметром
5     Person sam = new Person("Sam", 25);  // вызов 3-его конструктора с двумя параметрами
6
7
8     bob.GetInfo();                     // Имя: Bob Возраст: 18
9     tom.GetInfo();                    // Имя: Неизвестно Возраст: 18
10    sam.GetInfo();                   // Имя: Sam Возраст: 25
11 }
```

# Ключевое слово this

```
1 class Person
2 {
3     public string name;
4     public int age;
5
6     public Person() : this("Неизвестно")
7     {
8     }
9     public Person(string name) : this(name, 18)
10    {
11    }
12    public Person(string name, int age)
13    {
14        this.name = name;
15        this.age = age;
16    }
17    public void GetInfo()
18    {
19        Console.WriteLine($"Имя: {name} Возраст: {age}");
20    }
```

## Инициализация

```
1 Person tom = new Person { name = "Том", age=31 };
2 tom.GetInfo(); // Имя: Том Возраст: 31
```

# Типы значений и ссылочные типы

## Типы значений:

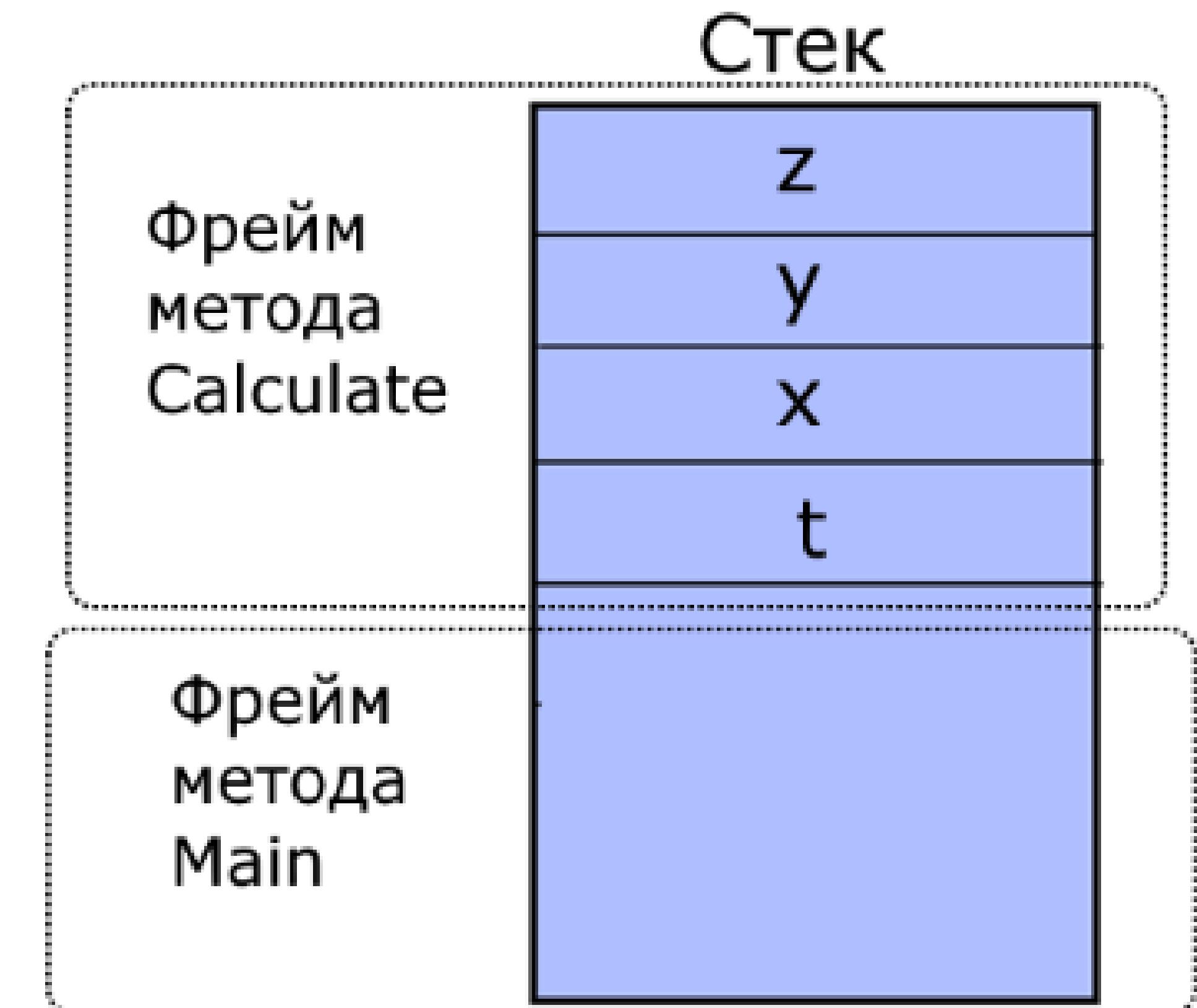
- Целочисленные типы (byte, short, int, long...)
- Типы с плавающей запятой (float, double)
- Тип decimal
- Тип bool
- Тип char
- Перечисления enum
- Структуры (struct)

## Ссылочные типы:

- Тип object
- Тип string
- Классы (class)
- Интерфейсы (interface)
- Делегаты (delegate)

Например:

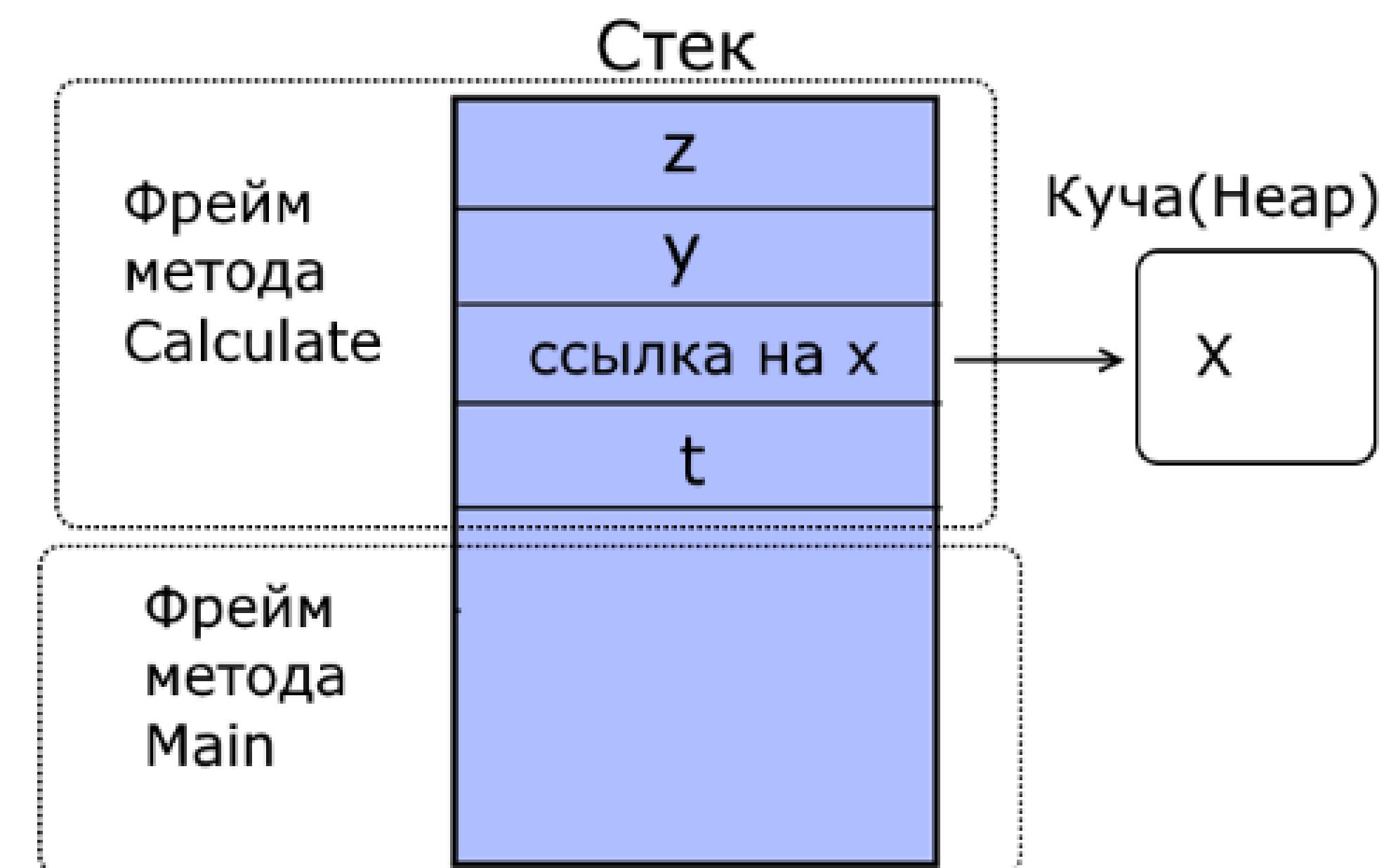
```
1 class Program
2 {
3     static void Main(string[] args)
4     {
5         Calculate(5);
6         Console.ReadKey();
7     }
8
9     static void Calculate(int t)
10    {
11        int x = 6;
12        int y = 7;
13        int z = y + t;
14    }
15 }
```



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.

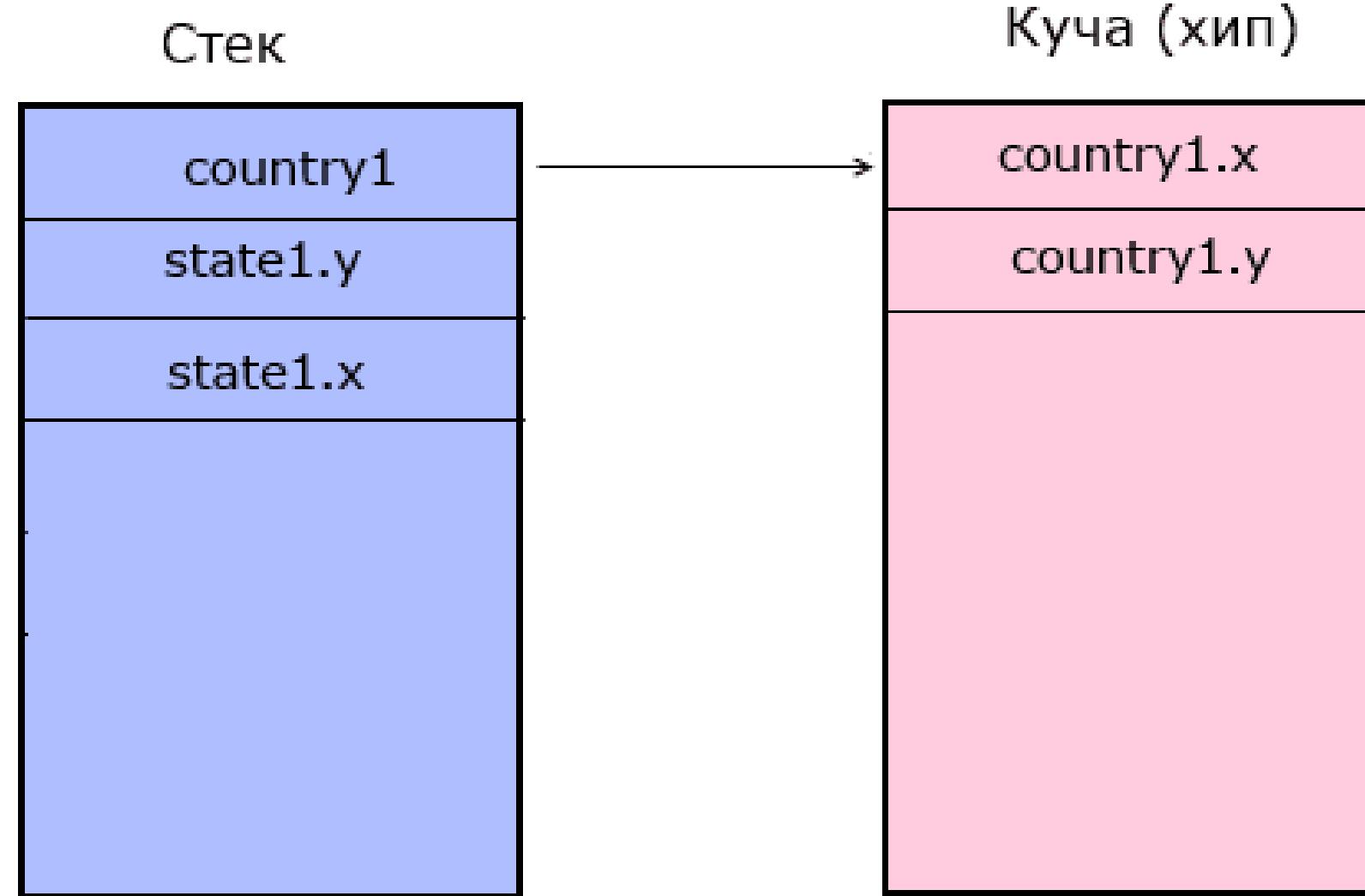
```
1 static void Calculate(int t)
2 {
3     object x = 6;
4     int y = 7;
5     int z = y + t;
6 }
```



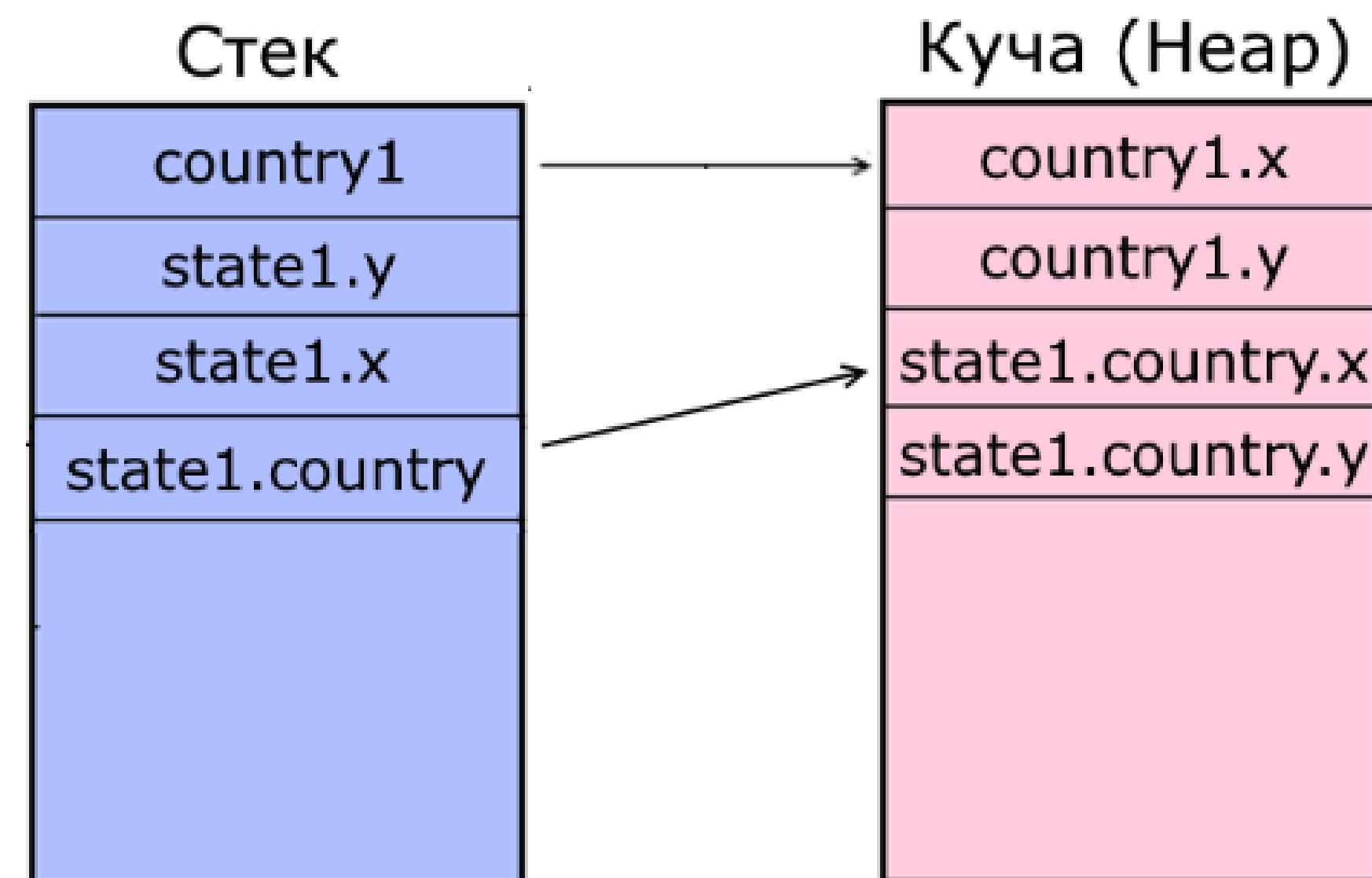
# Составные типы

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

```
1 class Program
2 {
3     private static void Main(string[] args)
4     {
5         State state1 = new State(); // State - структура, ее данные размещены в стеке
6         Country country1 = new Country(); // Country - класс, в стек помещается ссылка на адрес в хипе
7             // а в хипе располагаются все данные объекта country1
8     }
9 }
10 struct State
11 {
12     public int x;
13     public int y;
14     public Country country;
15 }
16 class Country
17 {
18     public int x;
19     public int y;
20 }
```



```
1 private static void Main(string[] args)
2 {
3     State state1 = new State();
4     state1.country = new Country();
5     Country country1 = new Country();
6 }
```



# Интерфейсы

```
1 interface IMovable
2 {
3     // константа
4     const int minSpeed = 0;      // минимальная скорость
5     // статическая переменная
6     static int maxSpeed = 60;    // максимальная скорость
7     // метод
8     void Move();                // движение
9     // свойство
10    string Name { get; set; }   // название
11
12    delegate void MoveHandler(string message); // определение делегата для события
13    // событие
14    event MoveHandler MoveEvent; // событие движения
15 }
```

И например, мы могли бы обратиться к константе `minSpeed` и переменной `maxSpeed` интерфейса `IMovable`:

```
1 static void Main(string[] args)
2 {
3     Console.WriteLine(IMovable.maxSpeed);
4     Console.WriteLine(IMovable.minSpeed);
5 }
```

```
1 interface IMovable
2 {
3     public const int minSpeed = 0;      // минимальная скорость
4     private static int maxSpeed = 60;    // максимальная скорость
5     public void Move();
6     protected internal string Name { get; set; } // название
7     public delegate void MoveHandler(string message); // определение делегата для события
8     public event MoveHandler MoveEvent; // событие движения
9 }
```

```
1 interface IMovable
2 {
3     // реализация метода по умолчанию
4     void Move()
5     {
6         Console.WriteLine("Walking");
7     }
8 }
```

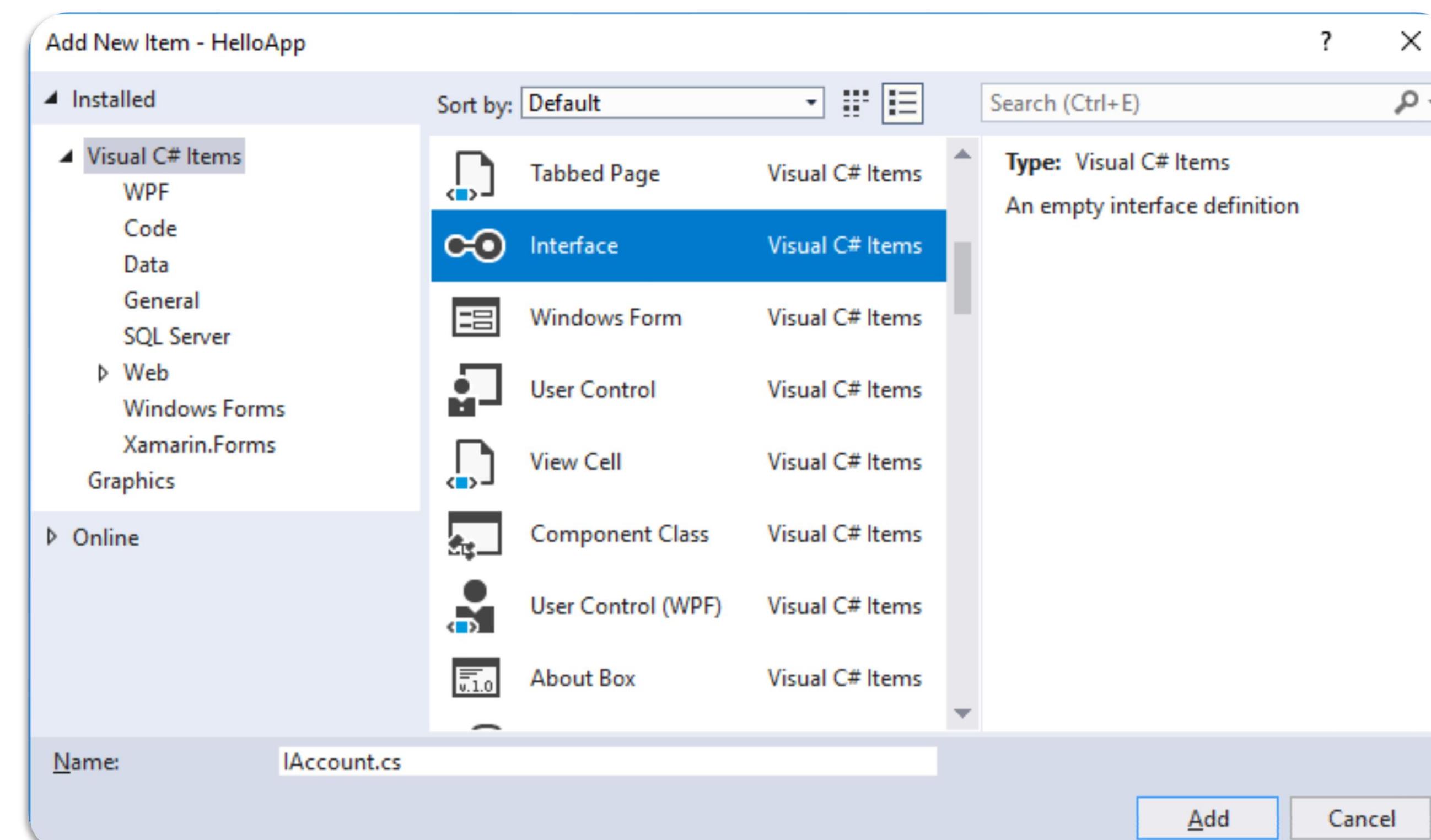
Тем не менее реализацию по умолчанию для свойств мы тоже можем определять:

```
1 interface IMovable
2 {
3     void Move()
4     {
5         Console.WriteLine("Walking");
6     }
7     // реализация свойства по умолчанию
8     // свойство только для чтения
9     int MaxSpeed { get { return 0; } }
10 }
```

```
1 interface IMovable
2 {
3     public const int minSpeed = 0;      // минимальная скорость
4     private static int maxSpeed = 60;    // максимальная скорость
5         // находим время, за которое надо пройти расстояние distance со скоростью speed
6     static double GetTime(double distance, double speed) => distance / speed;
7     static int MaxSpeed
8     {
9         get { return maxSpeed; }
10        set
11        {
12            if (value > 0) maxSpeed = value;
13        }
14    }
15 }
16 class Program
17 {
18     static void Main(string[] args)
19     {
20         Console.WriteLine(IMovable.MaxSpeed);
```

# Модификаторы доступа интерфейсов

```
1 public interface IMovable
2 {
3     void Move();
4 }
```



```
Character
AutoBackToTitle.cs
ClickToStart.cs
Explosion.cs
Explosive.cs
Fire.cs
FloorSection.cs
GameControl.cs
GameGUI.cs
Hose.cs
MapIcons.cs
MessageGUI.cs
MoveBetweenPoints
Player.cs
Priority Particle Add.
PriorityAlphaParticle
SceneChanger.cs
SmokeParticles.cs
WaterHoseParticles
WaterSplash.cs
world.cs
```

```
50 vignette.blur = (1-health) * 2 + smokeEffect * 30 * (Health - 1);
51 vignette.blurDistance = (1-health) * 2 * smokeEffect * 30;
52 vignette.chromaticAberration = heatEffect * 50;
53 }
54
55
56 void OnTriggerStay(Collider c)
57 {
58     var fire = c.GetComponent<Fire>();
59     if (fire && fire.alive)
60     {
61         float dist = 1-((transform.position - fire.transform.position).magnitude);
62         NearHeat(dist);
63     }
64
65
66     var smoke = c.GetComponent<ClickStartToSmash>();
67     if (smoke && smoke.GetComponent<ParticleSystem>().active)
68     {
69         float dist = 1-((transform.position - smoke.transform.position).magnitude);
70         NearSmoke(dist);
71     }
72 }
73
74 void OnCollisionEnter(Collision c)
75 {
76     var healthBox = c.gameObject.GetComponent<HealthBox>();
77     if (healthBox)
78     {
```

# Создание и Использование Скриптов

Unity изначально поддерживает три языка программирования:

- **C#** (произносится как Си-шарп),
- Стандартный в отрасли язык подобный Java или C++;
- **UnityScript**, язык, разработанный специально для использования в Unity по образцу JavaScript;

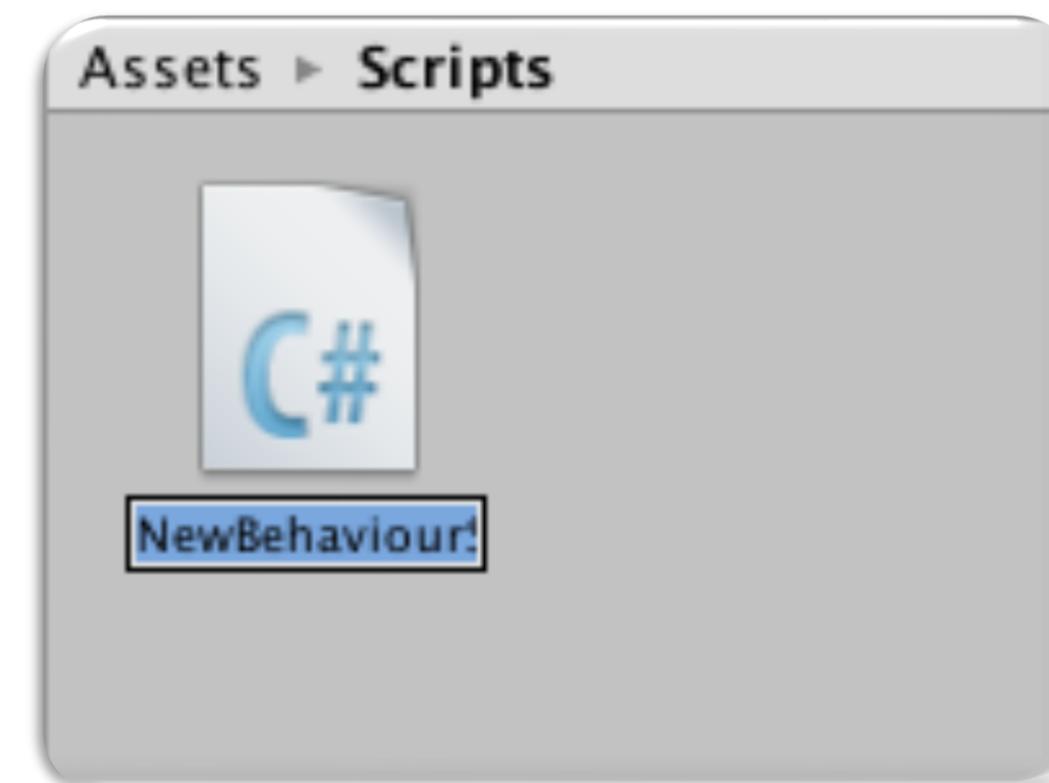
В дополнение к этим, с Unity могут быть использованы многие другие языки семейства .NET, если они могут компилировать совместимые DLL

# Создание скриптов

В отличии от других ассетов, скрипты обычно создаются непосредственно в Unity. Можно создать скрипт используя меню Create в левом верхнем углу панели Project или выбрав Assets > Create > C# Script (или JavaScript/Boo скрипт) в главном меню.

Новый скрипт будет создан в папке, которую вы выбрали в панели Project. Имя нового скрипта будет выделено, предлагая вам ввести новое имя.

Лучше ввести новое имя скрипта сразу после создания чем изменять его потом. Имя, которое вы введете будет использовано, чтобы создать начальный текст в скрипте



# Структура файла скрипта

После двойного щелчка на скрипте в Unity, он будет открыт в текстовом редакторе.

Содержимое файла будет выглядеть примерно так:

```
using UnityEngine;
using System.Collections;
public class MainPlayer : MonoBehaviour
{
    // Use this for initialization
    void Start ()
    {
    }
    // Update is called once per frame
    void Update () {
    }
}
```

# Библиотеки C# Unity3D

```
using UnityEngine  
//подключает функции Unity3D  
using System  
//содержит фундаментальные и базовые классы  
using System.Collections  
//содержит типы, определяющие различные объекты коллекций  
using System.IO  
//содержит типы, поддерживающие ввод и вывод, включая  
возможности чтения и записи данных в потоках  
using System.Linq  
//содержит классы, для поддерживания массивов  
using System.Xml  
//содержит типы для обработки языка XML
```

# Управление игровым объектом



После присоединения скрипта начнет работать, когда вы нажмете Play и запустите проект.

## Переменные и инспектор

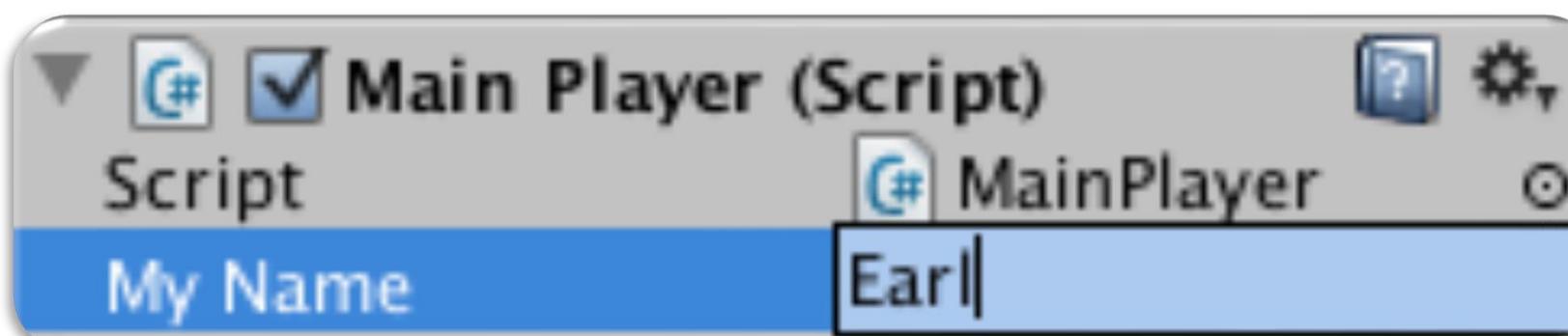
Этот код создает редактируемое поле в Инспекторе, помеченное «My Name».

```
using UnityEngine;
using System.Collections;

public class MainPlayer : MonoBehaviour {
    public string myName;

    // Use this for initialization
    void Start () {
        Debug.Log("I am alive and my name is " + myName);
    }

    // Update is called once per frame
    void Update () {
    }
}
```



# Управление игровыми объектами (GameObjects) с помощью компонентов

В редакторе Unity вы изменяете свойства Компонента используя окно Inspector. Так, например, изменения позиции компонента Transform приведет к изменению позиции игрового объекта. Аналогично, вы можете изменить цвет материала компонента Renderer или массу твёрдого тела (Rigidbody) с соответствующим влиянием на отображение или поведение игрового объекта. По большей части скрипты также изменяют свойства компонентов для управления игровыми объектами. Разница, однако, в том, что скрипт может изменять значение свойства постепенно со временем или по получению ввода от пользователя. За счет изменения, создания и уничтожения объектов в заданное время может быть реализован любой игровой процесс.

# Обращение к компонентам

```
void Start () {  
    Rigidbody rb = GetComponent<Rigidbody>();  
}
```

Как только у вас есть ссылка на экземпляр компонента, вы можете устанавливать значения его свойств, тех же, которые вы можете изменить в окне Inspector:

```
void Start () {  
    Rigidbody rb = GetComponent<Rigidbody>();  
  
    // Change the mass of the object's Rigidbody.  
    rb.mass = 10f;  
}
```

# Обращение к другим объектам

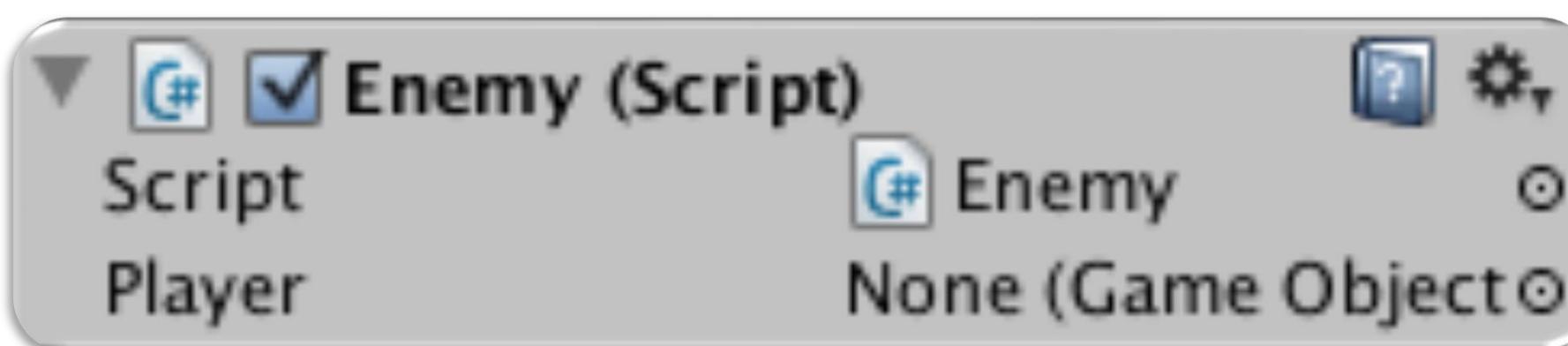
Пусть иногда они и существуют изолированно, все же, обычно, скрипты отслеживают другие объекты.

## Связывание объектов через переменные

Самый простой способ найти нужный игровой объект - добавить в скрипт переменную типа `GameObject` с уровнем доступа `public`:

```
public class Enemy : MonoBehaviour {  
    public GameObject player;  
  
    // Other variables and functions...  
}
```

Переменная будет видна в окне `Inspector`, как и любые другие:



```
public class Enemy : MonoBehaviour {  
    public GameObject player;  
  
    void Start() {  
        // Start the enemy ten units behind the player character.  
        transform.position = player.transform.position - Vector3.forward * 10f;  
    }  
}
```

public Transform playerTransform;

# Нахождение дочерних объектов

```
using UnityEngine;

public class WaypointManager : MonoBehaviour {
    public Transform[] waypoints;

    void Start() {
        waypoints = new Transform[transform.childCount];
        int i = 0;

        foreach (Transform t in transform) {
            waypoints[i++] = t;
        }
    }
}
```

Также можно найти заданный дочерний объект по имени, используя функцию `Transform.Find`:

```
transform.Find("Gun");
```

Это может быть полезно, когда объект содержит дочерний элемент, который может быть добавлен или удален в игровом процессе.

# Нахождение объектов по имени или тегу

Отдельные объекты могут быть получены по имени, используя функцию `GameObject.Find`:

```
GameObject player;

void Start() {
    player = GameObject.Find("MainHeroCharacter");
}
```

Объект или коллекция объектов могут быть также найдены по их тегу, используя функции `GameObject.FindWithTag` и `GameObject.FindGameObjectsWithTag`:

```
GameObject player;
GameObject[] enemies;

void Start() {
    player = GameObject.FindWithTag("Player");
    enemies = GameObject.FindGameObjectsWithTag("Enemy");
}
```

# Функции событий. Обычные Update события

Проект в Unity - это что-то вроде анимации, в которой кадры генерируются на ходу. Ключевой концепт в программировании в Unity заключается в изменении позиции, состояния и поведения объектов в проекте прямо перед отрисовкой кадра. Такой код в Unity обычно размещают в функции Update. Update вызывается перед отрисовкой кадра и перед расчётом анимаций.

```
void Update() {
    float distance = speed * Time.deltaTime * Input.GetAxis("Horizontal");
    transform.Translate(Vector3.right * distance);
}
```

## Функция FixedUpdate.

```
void FixedUpdate() {
    Vector3 force = transform.forward * driveForce * Input.GetAxis("Vertical");
    rigidbody.AddForce(force);
}
```

## Функция LateUpdate.

```
void LateUpdate() {
    Camera.main.transform.LookAt(target.transform);
}
```

# События инициализации

Зачастую полезно иметь возможность вызвать код инициализации перед любыми обновлениями, происходящими во время игры. Функция Start вызывается до обновления первого кадра или физики объекта. Функция Awake вызывается для каждого объекта в сцене в момент загрузки сцены.

# События GUI

```
void OnGUI() {  
    GUI.Label(labelRect, "Game Over");  
}
```

# События физики

Физический движок сообщает о столкновениях с объектом с помощью вызова функций событий в скрипте этого объекта. Функции `OnCollisionEnter`, `OnCollisionStay` и `OnCollisionExit` будут вызваны по началу, продолжению и завершению контакта. Соответствующие функции `OnTriggerEnter`, `OnTriggerStay` и `OnTriggerExit` будут вызваны когда колайдер объекта настроен как Trigger (т.е. этот колайдер просто определяет, что его что-то пересекает и не реагирует физически).

```
void OnCollisionEnter(otherObj: Collision) {  
    if (otherObj.tag == "Arrow") {  
        ApplyDamage(10);  
    }  
}
```

# Порядок выполнения функций событий

В скрипtinge Unity есть некоторое количество функций события, которые исполняются в заранее заданном порядке по мере исполнения скрипта.

## Редактор

- **Reset:** Reset вызывается для инициализации свойств скрипта, когда он только присоединяется к объекту и тогда, когда используется команда *Reset*.

## Первая загрузка сцены

Эти функции вызываются при запуске.

- **Awake:** Эта функция всегда вызывается до любых функций Start и также после того, как префаб был вызван в сцену.
- **OnEnable:** (вызывается только если объект активен): Эта функция вызывается сразу после включения объекта.

## Перед первым обновлением кадра

- **Start:** Функция Start вызывается до обновления первого кадра(first frame) только если скрипт включен.

## Между кадрами

- **OnApplicationPause:** Эта функция вызывается в конце кадра, во время во время которого вызывается пауза, что эффективно между обычными обновлениями кадров.

## Порядок обновления

Когда вы отслеживаете игровую логику и взаимодействия, анимации, позиции камеры и т.д. есть несколько разных событий, которые вы можете использовать.

- **FixedUpdate:** Зачастую случается, что FixedUpdate вызывается чаще чем Update.
- **Update:** Update вызывается раз за кадр. Это главная функция для обновлений кадров.
- **LateUpdate:** LateUpdate вызывается раз в кадр, после завершения Update. Любые вычисления произведённые в Update будут уже выполнены на момент начала LateUpdate. Часто LateUpdate используют для преследующей камеры от третьего лица.

## Рендеринг

- **OnPreCull:** Вызывается до того, как камера отсечёт сцену. Отсечение определяет, какие объекты будут видны в камере. OnPreCull вызывается прямо перед тем, как начинается отсечение.
- **OnBecameVisible/OnBecameInvisible:** Вызывается тогда, когда объект становится видимым/невидимым любой камере.
- **OnWillRenderObject:** Вызывается один раз для каждой камеры, если объект в поле зрения.
- **OnPreRender:** Вызывается перед тем, как камера начнёт рендерить сцену.
- **OnRenderObject:** Вызывается, после того, как все обычные рендеры сцены завершатся. Вы можете использовать класс GL или Graphics.DrawMeshNow, чтобы рисовать пользовательскую геометрию в данной точке.
- **OnPostRender:** Вызывается после того, как камера завершит рендер сцены.
- **OnRenderImage(только в Pro версии):** Вызывается после завершения рендера сцены, для возможности пост-обработки изображения экрана.
- **OnGUI:** Вызывается несколько раз за кадр и отвечает за элементы интерфейса (GUI). Сначала обрабатываются события макета и раскраски, после чего идут события клавиатуры/мышки для каждого события.
- **OnDrawGizmos:** Используется для отрисовки гизмо в окне Scene View в целях визуализации.

# Сопрограммы

Разные способы использования сопрограмм:

- **yield** Сопрограмма продолжит выполнение, после того, как все Update функции были вызваны в следующем кадре.
- **yield WaitForSeconds** Продолжает выполнение после заданной временной задержки, и после всех Update функций, вызванных в итоговом кадре.
- **yield WaitForFixedUpdate** Продолжает выполнение после того, как все функции FixedUpdate были вызваны во всех скриптах
- **yield WWW** продолжает выполнение после завершения WWW-загрузки.
- **yield StartCoroutine** сцепляет сопрограмму, и будет ждать, пока не завершится сопрограмма MyFunc.

## Когда объект разрушается

- **OnDestroy:** Эта функция вызывается после всех обновлений кадра в последнем кадре объекта, пока он ещё существует (объект может быть уничтожен при помощи Object.Destroy или при закрытии сцены).

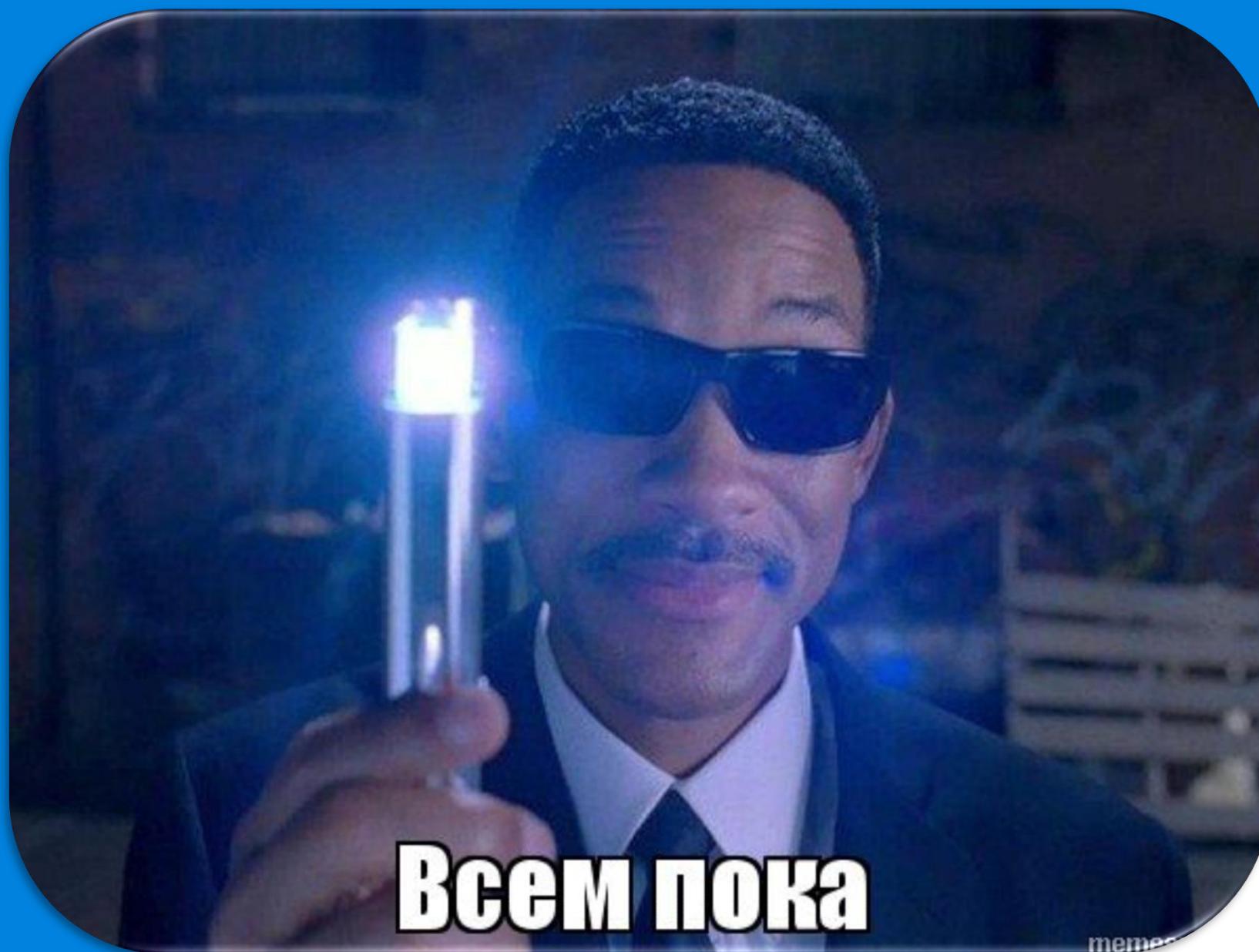
## При выходе

Эти функции вызываются во всех активных объектах в вашей сцене:

- **OnApplicationQuit:** Эта функция вызывается для всех игровых объектов перед тем, как приложение закрывается. В редакторе вызывается тогда, когда игрок останавливает игровой режим. В веб-плеере вызывается по закрытия веб окна.
- **OnDisable:** Эта функция вызывается, когда объект отключается или становится неактивным.

# Благодарю за внимание

Синицын Анатолий Васильевич  
Заместитель заведующего кафедрой ИиППО



Кафедра ИиППО | РТУ МИРЭА  
[vk.com/ippo\\_it](https://vk.com/ippo_it)

