

# ОСНОВЫ C#



# Code convention

**PascalCasing**

*IasdGen*

**camelCasing**

*iasdGenTyi*

**UPPERCASE**

***ID***

**Hungarian notation** *strName, iYear*

1) При именовании идентификаторов не используются аббревиатуры или сокращения - полностью

2) имена типов, структур, перечислений, интерфейсов, методов, свойств – pascal case  
SampleClass    ISampleInterface    SampleMethod();

3) Имена локальных переменных, аргументов методов, защищенных (protected) полей – camel  
sampleArgument

4) Закрытые поля  
mJJJJ, m\_JJJJ, \_Jjjjjjj

5) Функции и методы - pascal case

*void HelloWorld();*

Имя функции начинается с глагола, указывающего на то, какое действие она выполняет

6) Константы - **pascal case**

# Форматирование

- ▶ В одном файле не объявляется больше одного namespace'a и одного класса (исключение – небольшие вспомогательные private классы);
- ▶ Фигурные скобки размещаются всегда на отдельной строке;
- ▶ В условии **if-else** всегда используются фигурные скобки;
- ▶ Размер tab'a – 4;

- ▶ Использование строк длиннее 100 символов не желательно. При необходимости инструкция переносится на другую строку. При переносе части кода на другую строку вторая и последующая строки сдвигаются вправо на один символ табуляции;
- ▶ Каждая переменная объявляется на отдельной строке;
- ▶ Все подключения **namespace**'ов (**using**) размещаются в начале файла
- ▶ Функции, поля и свойства группируются внутри класса по своему назначению. Такие группы объединяются в регионы;

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

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

Обозреватель решений — поиск (Ctrl+;)

+ Решение "OOP\_Lect" (проектов: 1)

▲ + C# OOP\_Lect

▷ + Properties

▷ Ссылки

+ App.config

▲ + C# Program.cs

▷ Program

## ► Пространства имён

используются для логической группировки родственных типов

```
namespace MyCompany  
{  
    class Utility {}  
}
```

В C# директива namespace заставляет компилятор добавлять к каждому имени типа определенную приставку

- не обязательно должны ограничиваться единственной единицей компиляции



```
namespace MyCompany
{
    namespace Utilities
    {
        class SomeUtility {}
    }
}
```

MyCompany.Utilities.SomeUtility

Могут быть вложены

# ► использование директивы using

## импортирует все имена из заданного пространства имён в окружающее пространство имён

```
using MyCompany.Utilities; //резервируем пространство MyCompany.U
namespace ConsoleApplication
{
    static class EntryPoint
    {
        static void Main()
        {
            MyCompany.Utilities.SomeUtility su1 =
                new MyCompany.Utilities.SomeUtility();
            //аналогично, благодаря using
            SomeUtility su2 = new SomeUtility();
        }
    }
}
```

не хватает лаконичности

using заставляет компилятор C# добавлять к имени указанный префикс, пока не будет найдено совпадение

```
using System;
```

содержит фундаментальные и базовые классы платформы .NET

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

```
using System.Linq;
```

```
using System.Text;
```

описаны типы для обработки строк

```
namespace ConsoleApplication1
```

```
{
```

```
    class Program
```

```
    {
```

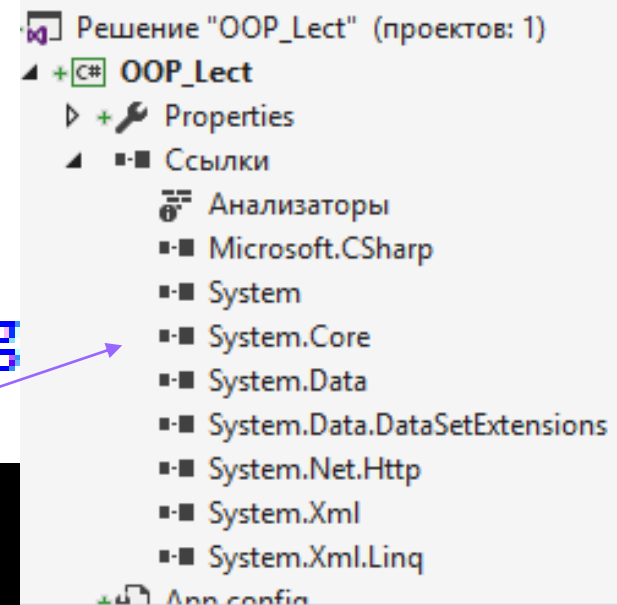
```
        static void Main(string
```

```
        {
```

```
        }
```

```
    }
```

Физически пространства имен находятся в подключаемых библиотеках dll



форма директивы using, позволяющая создать псевдоним для отдельного типа или пространства имен

## ► псевдонимы (alias) имен

```
using SomeUtil = MyCompany.Utilities.SomeUtility; //псевд  
namespace ConsoleApplication  
{  
    static class EntryPoint  
    {  
        static void Main()  
        {  
            SomeUtil su = new SomeUtil();  
        }  
    }  
}
```

# XML-теги

для создания самодокументируемых классов.

///

Например:

|              |               |                |
|--------------|---------------|----------------|
| <c>          | <para>        | <see>*         |
| <code>       | <param>*      | <seealso>*     |
| <example>    | <paramref>    | <summary>      |
| <exception>* | <permission>* | <typeparam>*   |
| <include>*   | <remarks>     | <typeparamref> |
| <list>       | <returns>     | <value>        |

/// <summary>

XML-дескрипторы

/// Точка входа приложения.

/// </summary>

/// <param name="args">Пар.ком. строки.</p>

static void Main(string[] args) {}

# Синтаксис C#

► Сохранились :

if..else, while, do..while

унарные и бинарные операции

# Класс Console.

## Консольный ввод/вывод

статический класс System.Console

```
int x = Console.ReadLine();  
Console.WriteLine((char)x);
```

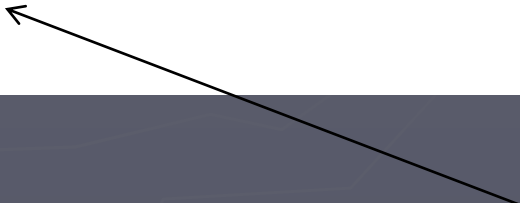
```
String s = Console.ReadLine();  
Console.WriteLine(s);
```

# Методы класса Console

- **Clear:** очистка консоли
- **WriteLine:** вывод строки текста с переводом на новую строку
- **Write:** вывод строки текста
- **Read:** считывание введенного символа в виде числового кода данного символа. С помощью преобразования к типу `char` мы можем получить введенный символ
- **ReadKey:** считывание нажатой клавиши клавиатуры  
(`ConsoleKeyInfo key= Console.ReadKey();`)
- **ReadLine:** считывание строки текста со входного потока



```
int i = 10;  
int j = 20;  
Console.WriteLine  
("{0} плюс {1} равно {2}", i, j, i + j);
```



указатели места заполнения  
(placeholder)

```
10 плюс 20 равно 30  
  10  
+ 20  
----  
 30
```

```
Console.WriteLine("{0,4}\n+{1,3}\n----\n{2,4}",  
                  i, j, i + j);
```

## ► МОЖНО ВСТАВЛЯТЬ переменные в строку

```
int oneMillion = 1_000_000;  
double balans = 1_23.456;  
  
Console.WriteLine($"У меня: {oneMillion}");  
  
Console.WriteLine($"баланс: {balans}");
```

# ► Выра- жения

в C#

идентичны

C++

| Группа операций              | Входящие операции      |
|------------------------------|------------------------|
| Первичные                    | x.m                    |
|                              | x(...)                 |
|                              | x[...]                 |
|                              | x++, x--               |
|                              | new T(...), new T[...] |
| Унарные                      | +x, -x                 |
|                              | !x                     |
|                              | ~x                     |
|                              | ++x, --x               |
|                              | (T)x                   |
| Мультипликативные            | x*y, x/y, x%y          |
| Аддитивные                   | x+y, x-y               |
| Сдвига                       | x<<y, x>>y             |
| Отношений и проверки типа    | x<y, x>y; x<=y, x>=y   |
|                              | x is T                 |
|                              | x as T                 |
| Эквивалентности              | x == y, x != y         |
| Логического И                | x & y                  |
| Логического исключающего ИЛИ | x ^ y                  |
| Логического ИЛИ              | x   y                  |
| Условное И                   | x && y                 |
| Условное ИЛИ                 | x    y                 |
| Сравнения с null             | x ?? y                 |
| Условные                     | x ? y : z              |
| Присваивания                 | x = y                  |
|                              | x op= y                |

► **Операторы** в C# идентичны C++

► **Типы данных C#**

поддерживает общую систему типов (CTS):

- для объявления того или иного встроенного типа данных из CTS обычно предусмотрено свое уникальное ключевое слов

# Система типов С#

## Типы значений (value types)

- 1) Определяются `struct` или `enum`
- 2) Размещение в стеке потока
- 3) поля экземпляра размещаются в самой переменной
- 4) не обрабатываются сборщиком мусора

### Типы значения

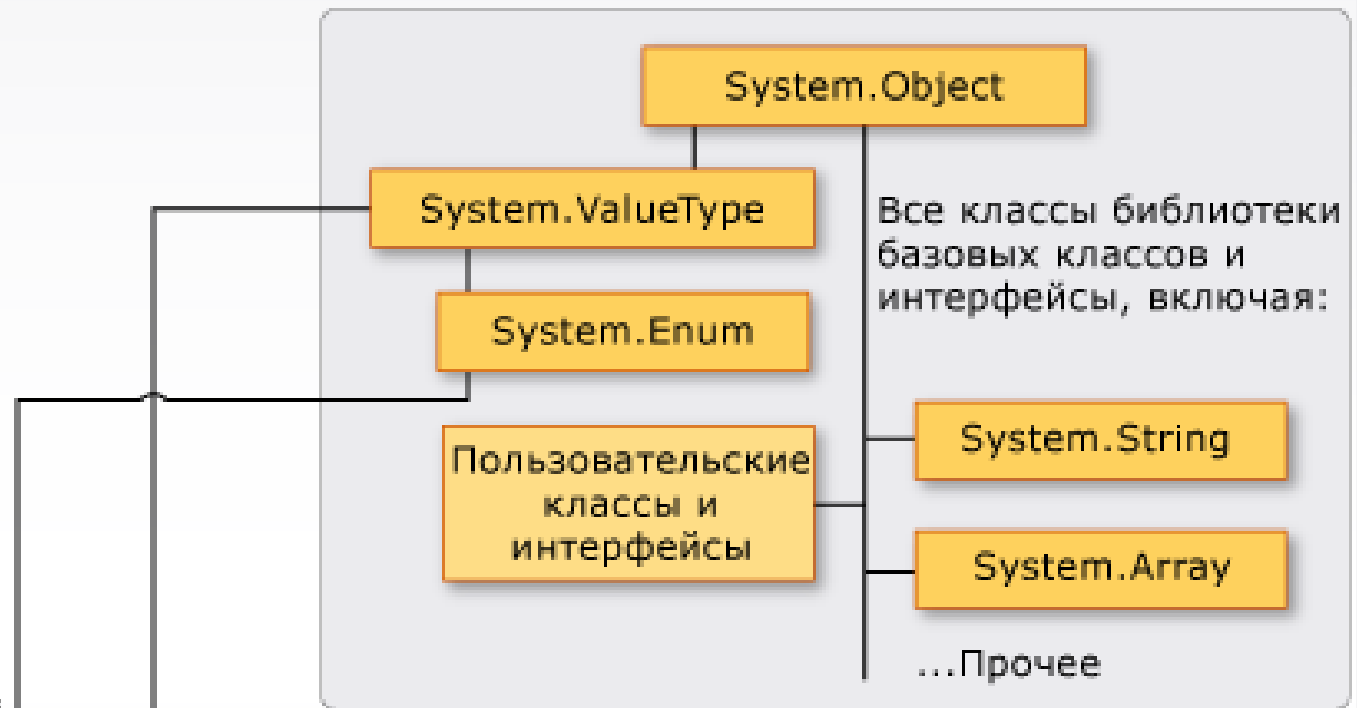
- Структуры
- Перечисления
- Простые типы

### Типы- ссылки

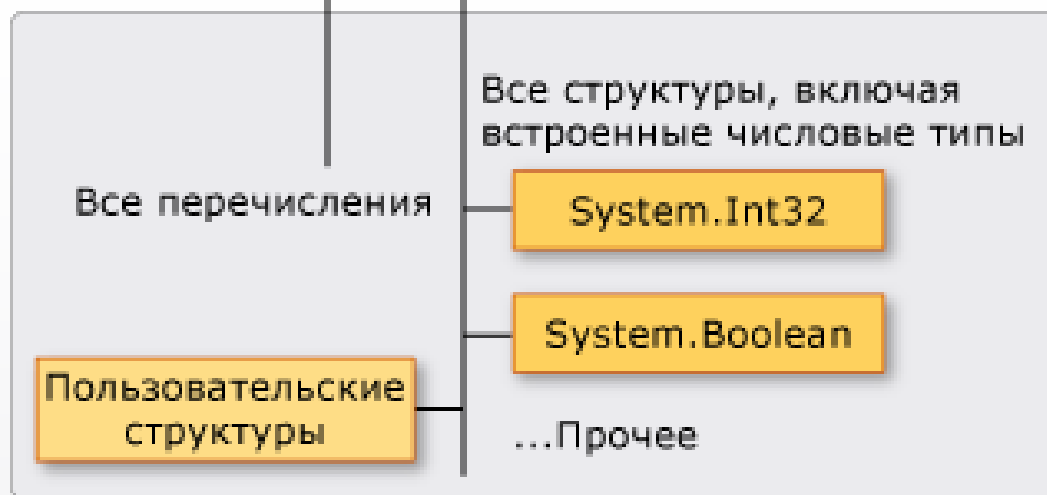
- Классы
- Строки
- Массивы
- Делегаты
- Интерфейсы

**Ссылочные типы**  
(reference types).  
определяются  
`class` (в куче)

## Ссылочные типы



## Типы значений



```
System.Int32 a1 = new System.Int32();
```

```
int a2 = 0;
```

- ▶ Типы данных, которые поддерживаются компилятором напрямую, называются примитивными (primitive types) или встроенными у них существуют прямые аналоги в библиотеке классов .NET Framework Class Library

```
using short = System.Int16;  
using ushort = System.UInt16;  
using int = System.Int32;  
using uint = System.UInt32;
```




преобразование

## ► Встроенные (примитивные) типы C#

| Тип C#  | Размер<br>в битах | Тип System (FCL) |
|---------|-------------------|------------------|
| sbyte   | 8                 | System.Sbyte     |
| short   | 16                | System.Int16     |
| int     | 32                | System.Int32     |
| long    | 64                | System.Int64     |
| byte    | 8                 | System.Byte      |
| ushort  | 16                | System.UInt16    |
| uint    | 32                | System.UInt32    |
| ulong   | 64                | System.UInt64    |
| char    | 16                | System.Char      |
| bool    | 8                 | System.Boolean   |
| float   | 32                | System.Single    |
| double  | 64                | System.Double    |
| decimal | 128               | System.Decimal   |
| string  | -                 | System.String    |
| object  | -                 | System.Object    |

до 29 десятичных цифр  
 $1.0 \times 10^{-28}$  до  $7.9 \times 10^{28}$





## ► КОМПИЛЯТОР ВЫПОЛНЯЕТ ЯВНОЕ И НЕЯВНОЕ ПРИВЕДЕНИЕ МЕЖДУ ПРИМИТИВНЫМИ ТИПАМИ

```
Int32 i32 = 5;  
Int64 i64 = i32;      // Неявное приведение Int32 к Int64  
Single s = i32;       // Неявное приведение Int32 к Single
```

разрешает неявное приведение типа, если это преобразование «безопасно», то есть не сопряжено с потерей данных

```
Byte b = (Byte)i32;    // Явное приведение Int32 к Byte  
Int16 v = (Int16)s;    // Явное приведение Single к Int16
```

«небезопасное» преобразование означает «связанное с потерей точности или величины числа»

```
System.Console.WriteLine(42.ToString());
```

примитивные типы могут использовать литеральную форму записи

## ► Ссылочные типы

```
Something objSomething;
```

CLR требует, чтобы все объекты создавались оператором new.

нет оператора delete, то есть нет явного способа освобождения памяти, занятой объектом. Уборкой мусора занимается среда CLR

```
Something objSomething;  
objSomething = new Something();
```

| Имя (псевдоним) | Тип CTS       |
|-----------------|---------------|
| object          | System.Object |
| string          | System.String |

## ► Ссылочный тип Object

- В CLR каждый объект прямо или косвенно является производным от System.Object

```
// Тип, неявно производный от Object
class Student
{
//...
}

// Тип, явно производный от Object
class Person : System.Object
{
//...
}
```

# 1. Где выделяется память под ссылочные типы?

- 1) обычно в стеке потока, но могут быть встроены в ссылочные типы
- 2) память выделяется из очереди
- 3) память выделяется из управляемой кучи
- 4) память выделяется из кэша

## 2. Как правильно объявить?

```
int age = 19;
```

или

```
System.Int32 age = 19;
```

Почему?

# Упаковка и распаковка ЗНАЧИМЫХ ТИПОВ

- ▶ Упаковкой (boxing) называется процесс преобразования типа значения в тип `System.Object` или в тип интерфейса, который реализуется данным типом-значением

```
Int32 x = 5;
```

```
Object o = x;
```

```
// Упаковка x; o ссылается на упакованный объект
```

1. в управляемой куче выделяется память
2. поля копируются
3. возвращается адрес объекта

- ▶ распаковка (unboxing)

получение указателя на исходный значимый тип (поля данных), содержащийся в объекте

Объекты значимого типа существуют в двух формах: неупакованной (unboxed) и упакованной (boxed). Ссылочные типы бывают только в упакованной форме.

```
int x = 5;  
Object o = x;           // Упаковка x  
byte m = (byte)o;  
           // Генерируется InvalidCastException
```

1. Если переменная, содержащая ссылку на упакованный значимый тип, равна null, генерируется исключение NullPointerException.

2. Если ссылка указывает на объект, не являющийся упакованным значением требуемого значимого типа, генерируется исключение InvalidCastException

```
int x = 5;  
Object o = x; // Упаковка x  
byte m = (byte)(int)o; // Распаковка, а затем приведение типа
```

упаковка и распаковка/копирование  
снижают производительность  
приложения

```
class lection
{ struct A
    { public int Val; }

public static void Main()
{
    A myA = new A();
    myA.Val = 5;
    object refType = myA; // упаковка
    A ValType2 = (A)refType; // распаковка
}
}
```

приведение неупакованного экземпляра значимого типа к  
одному из интерфейсов этого типа требует, чтобы экземпляр  
был упакован



## Назначение:

- ▶ позволяет использовать типы-значения в коллекциях ( где элементы являются элементами типа object)
- ▶ внутренний механизм, который обеспечивает возможность вызывать для типов-значений, подобных int и struct, методы Object.

### 3. В каких строках приведенного ниже кода происходит распаковка (unboxing)

```
int f = 42; //1  
object bar = f; // 2  
int f2 = (int)bar; //3
```

# Сравнение типов

|                             | Значимые  | Ссылочные  |
|-----------------------------|---|--|
| размещение                  | В стеке потока                                  | В управляемой куче                                 |
| формы                       | В неупакованной (unboxed) и упакованной (boxed) | В упакованной (boxed)                              |
| Наследование                | System.ValueType (есть те же методы)            | System.Object                                      |
| по умолчанию присваивается  | 0   | null<br>может привести к<br>NullReferenceException |
| Операция =                  | выполняется копирование всех полей              | копируется только адрес                            |
| Освобождение                | нет   | Требует уборки мусора                              |
| Освобождение памяти         | Сразу   | Ожидает уборки мусора                              |
| Не преднамеренное изменение | Имеет собственную копию данных (не возможно)    | Могут ссылаться на один объект в куче (можно)      |

# Локальная переменная по ссылке

```
int a = 1;  
int b = a;  
a = 42;  
Console.WriteLine($"a: {a} b: {b}");
```

► C# 7

```
a: 42 b: 1
```

```
int a = 1;  
ref int b = ref a;  
a = 42;  
Console.WriteLine($"a: {a} b: {b}");
```

```
a: 42 b: 42
```

Создает ссылочную  
переменную, которая  
инициализируется  
ссылкой

# Тип данных dynamic

## ► Назначение :

- Отражения или коммуникации с другими компонентами

```
Int32 demo = 1; //0
```

```
Int32: 10  
String: AA
```

```
dynamic value;  
value = (demo == 0) ? (dynamic)5 : (dynamic)"A";  
value = value + value;
```

## ► Использование:

- для членов класса - поля, свойства/индексаторы, структур , для метода, делегата, или унарных/бинарных операторов

## ► Что происходит

может получить какое угодно начальное значение, и на протяжении времени его существования это значение может быть заменено новым

```
dynamic value;
```

```
val dynamic
```

```
Sys Представляет объект, операции которого будут разрешаться во время выполнения.
```

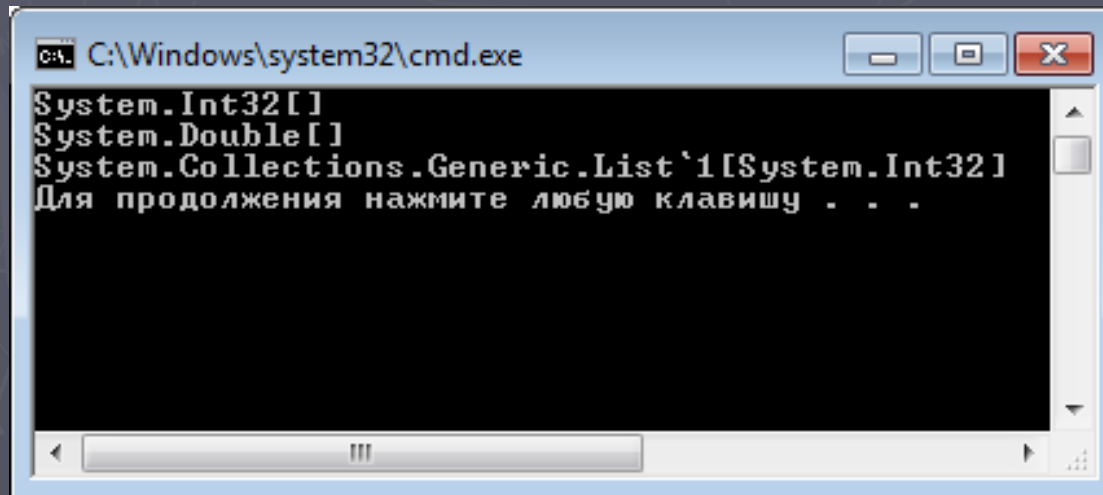
- ▶ компилятор конвертирует → `System.Object`
- ▶ применяет атрибут `System.Runtime.CompilerServices.DynamicAttribute`

### Ограничение динамических типов

- могут использоваться только для обращения к членам экземпляров (должна ссылаться на объект)
- не могут использовать лямбда-выражения или анонимные методы C# при вызове метода
- не могут воспринимать расширяющие методы

# Неявно типизированные локальные переменные. Ключевое слово var

```
var mas = new []{ 2, 5, 6, 3, 7, 2, 9, 2, 8, 4, 3, 2, 5, 12, 43, 124, 42, 1 };  
Console.Write(mas.GetType()); Console.WriteLine();  
var mas1 = new[] { 2, 1.45 };  
Console.Write(mas1.GetType()); Console.WriteLine();  
var mas2 = new List<int>  
(new int[] { 2, 5, 6, 3, 7, 2, 9, 2, 8, 4, 3, 2, 5, 12, 43, 124, 42, 1 });  
Console.Write(mas2.GetType()); Console.WriteLine();
```



при  
компиляции  
компилятор  
сам выводит  
тип данных  
исходя из  
присвоенного  
значения

# Ограничения:

- ▶ должно включать инициализатор (null нельзя)

```
int i1, i2 = 4; //допускается  
var v1, v2 = 2; //вызовет ошибки CS0810 и CS0811
```



- ✖ CS0819 Неявно типизированные переменные не могут быть многократно объявлены.
- ✖ CS0818 Неявно типизированные переменные должны быть инициализированы

- ▶ ключевое слово 'var' может применяться только в объявлении локальной переменной
- ▶ неявно типизированные локальные переменные не допускают множественного объявления



# 4. Какое значение по умолчанию имеет переменная типа-значение?

1)0

2)null

3>false

4)1

5)-98238483821

# типы Nullable

```
int x = null;
```

Не удастся преобразовать значение NULL в "int", поскольку этот тип значений не допускает значение NULL.

```
int? x = null;
```

упрощенная форма использования  
структуры **System.Nullable<T>**,  
которая позволяет null значения

```
Nullable<int> x = 5;
```

применяется только для типов значений

## ► Проверка на равенство объектов

```
int? x1 = null;  
int? x2 = null;  
System.Console.WriteLine(x1 == x2); //Tr
```

равны не только,  
когда они имеют  
ненулевые  
значения, которые  
совпадают, но и  
когда оба объекта  
равны null

## ► Преобразование

```
int? x1 = null;  
    if (x1.HasValue)  
    {  
        int x2 = (int)x1; //явное  
        int? x3 = x2; //неявное к T?  
        long? x4 = x2; //неявное расширяющее  
    }
```

## ► Оператор ?? (null-объединение)

```
int? x = null;  
int y = x ?? 1; // 1
```

```
int? z = 2;  
int t = z ?? 1; // 2
```

возвращает левый операнд, если этот операнд не равен null  
Для nullable-типов

# Что будет выведено на консоль?

```
int? a = 1;  
int? b = null;  
Console.WriteLine(b ?? a );
```

# Строки string

Тип `string` предназначен для работы со строками символов в кодировке Unicode. Ему соответствует базовый класс `System.String` библиотеки .NET.

*Создание строки:*

```
char[] a = { '0', '0', '0' };  
    // создание массива символов:  
string s;  
    // инициализация отложена  
string t = "qqq";  
    // инициализация строковым литералом  
string u = new string(' ', 20);  
    // с пом. конструктора  
string v = new string(a);  
    // создание из массива символов
```

# Операции для строк

- ▶ присваивание (=);
  - ▶ проверка на равенство содержимого (==);
  - ▶ проверка на неравенство (!=);
  - ▶ обращение по индексу ([]);
  - ▶ сцепление (конкатенация) строк (+)
  - ▶ <, >, >=, <= - сравнивают ссылки!!!!!!!
- 
- ❖ Строки равны, если имеют одинаковое количество символов и совпадают посимвольно.
  - ❖ Обращаться к отдельному элементу строки по индексу можно **только для получения значения**, но не для его изменения.
  - ❖ строки типа string относятся к **неизменяемым типам данных**.
  - ❖ Методы, изменяющие содержимое строки, на самом деле создают новую копию строки. Неиспользуемые «старые» копии автоматически удаляются сборщиком мусора.

# Строковые литералы

C# строки являются объектами

```
String path;  
path = "C:\\Windows\\regedit.exe"; //верно  
path = @"C:\Windows\regedit.exe"; //верно, на  
path = "C:/Windows/regedit.exe"; //верно  
path = "C:\Windows\regedit.exe"; //неверно, "
```

- ▶ Поставить ещё один символ обратной косой черты ("\\");
- ▶ Предварить строковый литерал символом '@';
- ▶ В случае, если литерал есть путь к файлу (локальный или сетевой), каталогу или web-странице, можно использовать альтернативный символ '/' вместо "\\".



# Некоторые элементы класса System.String

## Название

## Описание

### Compare

Сравнение двух строк в алфавитном порядке. Разные реализации метода позволяют сравнивать строки и подстроки с учетом и без учета регистра и особенностей национального представления дат и т. д.

### CompareOrdinal

Сравнение двух строк по кодам символов. Разные реализации метода позволяют сравнивать строки и подстроки

### CompareTo

Сравнение текущего экземпляра строки с другой строкой

### Concat

Конкатенация строк. Метод допускает сцепление произвольного числа строк

### Copy

Создание копии строки

**Format**

Форматирование в соответствии с заданными спецификаторами формата

**IndexOf,  
LastIndexOf,  
...**

Определение индексов первого и последнего вхождения заданной подстроки или любого символа из заданного набора

**Insert**

Вставка подстроки в заданную позицию

**Join**

Слияние массива строк в единую строку. Между элементами массива вставляются разделители (см. далее)

**Length**

Длина строки (количество символов)

**Remove**

Удаление подстроки из заданной позиции

**Replace**

Замена всех вхождений заданной подстроки или символа новой подстрокой или символом

**Split**

Разделение строки на элементы, используя заданные разделители. Результаты помещаются в массив строк

**Substring**

Выделение подстроки, начиная с заданной позиции

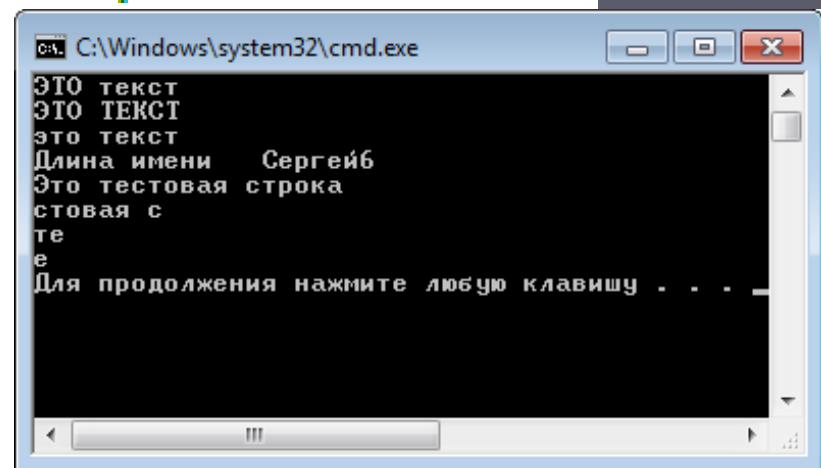
```
// Конвертируем в строчные и заглавные буквы
```

```
string test = "ЭТО текст";  
string testUpper = test.ToUpper();  
string testLower = test.ToLower();  
Console.WriteLine(test);  
Console.WriteLine(testUpper);  
Console.WriteLine(testLower);
```

```
string name = "Сергей";  
int length = name.Length;  
Console.WriteLine("Длина имени {0} ", name + length);
```

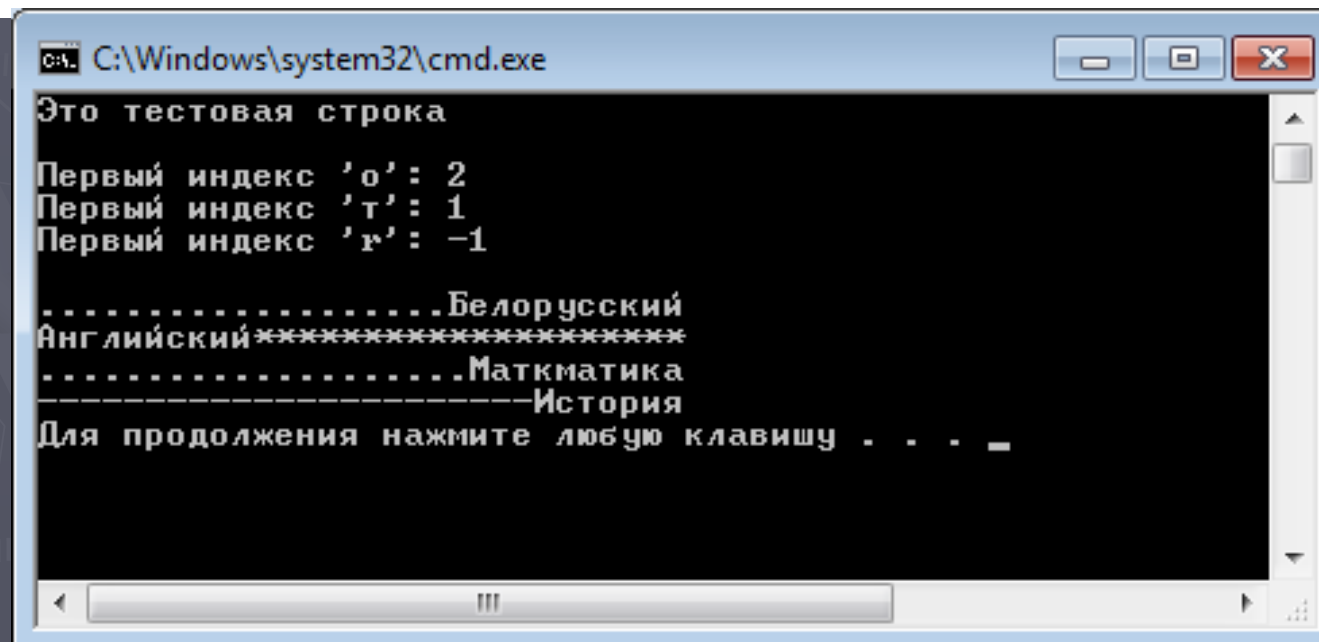
```
//Выделение подстрок
```

```
string name1 = "Это тестовая строка";  
Console.WriteLine(name1);  
string name2 = name1.Substring(6, 8); // начало и длина  
Console.WriteLine(name2);  
string name3 = name1.Substring(4, 2);  
Console.WriteLine(name3);  
string name4 = name1.Substring(5, name.Length - 5);  
Console.WriteLine(name4);
```



```
// Находим позицию текста в строке
string name5 = "Это тестовая строка";
Console.WriteLine(name5);
Console.WriteLine(Environment.NewLine + "Первый индекс 'о': " + name5.IndexOf("о"));
Console.WriteLine("Первый индекс 'т': " + name5.IndexOf("т"));
Console.WriteLine("Первый индекс 'r': " + name5.IndexOf("r"));

string s1 = "Белорусский", s2 = "Английский", s3 = "Маткматика", s4 = "История";
Console.WriteLine(Environment.NewLine + s1.PadLeft(30, '.'));
Console.WriteLine(s2.PadRight(30, '*'));
Console.WriteLine(s3.PadLeft(30, '.'));
Console.WriteLine(s4.PadLeft(30, '-'));
```



The screenshot shows a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The output of the C# code is displayed as follows:

```
Это тестовая строка
Первый индекс 'о': 2
Первый индекс 'т': 1
Первый индекс 'r': -1

.....Белорусский
Английский*****
.....Маткматика
-----История
Для продолжения нажмите любую клавишу . . . _
```

// ???????

```
string text = "This is a text. New string";

char[] delims = ".,:!?\n\r\xD\xA\" ".ToCharArray();
string[] words = text.Split(delims,
    StringSplitOptions.RemoveEmptyEntries);

foreach (string word in words)
    Console.WriteLine(word);
```

// ???????

```
foreach (string word in words)
    if (word[word.Length - 1] == 's')
        Console.WriteLine(word);
```

# Пустые строки и строки null

- ▶ **Пустая строка** — экземпляр объекта `System.String`, содержащий 0 СИМВОЛОВ:

```
string s = "";
```

Для пустых строк можно вызывать методы.

- ▶ Строки со значениями **null** не ссылаются на экземпляр объекта `System.String`, попытка вызвать метод для строки **null** вызовет исключение `NullReferenceException`. строки **null** можно использовать в операциях объединения и сравнения с другими строками.

# Форматированный вывод. Метод String.Format

```
using System;
class Program
{
    static void Main()
    {
        float
            par1 = 20.45F,
            par2 = 40.76F;
        String composite =
            String.Format("{0} + {1} = {2}",
                par1,
                par2,
                par1 + par2);
        Console.WriteLine(composite);
        Console.ReadKey();
    }
}
```

20,45 + 40,76 = 61,21

формат

{index[,alignment][:formatString]}

"C" – для валюты, 

"D" – для десятичных чисел,

"E" – для научной нотации,

"F" – для нотации с фиксированной точкой

{0,10:E}

интерфейсы IFormatter и ICustomFormatter



# Класс StringBuilder

- ▶ Любые модификации строки происходят внутри блока памяти

```
using System;  
using System.Text;
```

```
public sealed class App  
{
```

```
    static void Main()  
    {
```

```
        // Создаём StringBuilder максимум на 50 символов.  
        // Инициализируем StringBuilder строкой "ABC".  
        StringBuilder sb = new StringBuilder("ABC", 50);
```

```
        // Добавляем три символа (D, E, и F) в конец.  
        sb.Append(new char[] { 'D', 'E', 'F' });
```

```
        // Добавляем форматную строку в конец.  
        sb.AppendFormat("GHI{0}{1}", 'J', 'k');
```

```
        // Выводим число символов и строку, 'заклѳчѳнную' в StringBuilder  
        Console.WriteLine("{0} chars: {1}", sb.Length, sb.ToString());
```

```
        // Вставляем строку в начало.  
        sb.Insert(0, "Alphabet: ");
```

# Основные элементы класса `System.Text.StringBuilder`

## Append

Добавление в конец строки. Разные варианты метода позволяют добавлять в строку величины любых встроенных типов, массивы символов, строки и подстроки типа `string`

## AppendFormat

Добавление форматированной строки в конец строки

## Capacity

Получение или установка емкости буфера. Если устанавливаемое значение меньше текущей длины строки или больше максимального, генерируется исключение `ArgumentOutOfRangeException`

## Insert

Вставка подстроки в заданную позицию

## Length

Длина строки (количество символов)

## MaxCapacity

Максимальный размер буфера

## Remove

Удаление подстроки из заданной позиции

## Replace

Замена всех вхождений заданной подстроки или символа новой подстрокой или символом

## ToString

Преобразование в строку типа `string`

# Массив

```
int[] w = new int[10];  
string[] z = new string[100];
```

одномерные, многомерные  
и ступенчатые (не регулярные).

```
int []w = new int[10];  
int sizeW= w.Length;
```

System.Array → System.Object

Ссылочный тип - в куче

# Одномерные массивы (single-dimensional)

тип[] имя;

тип[] имя = new тип [ размерность ];

тип[] имя = { список инициализаторов };

тип[] имя = new тип [] { список  
инициализаторов };

тип[] имя = new тип [ размерность ] {  
список инициализаторов };

```
int[] a;  
int[] b;  
int[] c = { 61, 2, 5, -9 };  
int[] d = new int[] { 6, 2, 5, -9 };  
int[] e = new int[4] { 61, 2, 5, -9 };
```

```
int[] e = new int[4] { 61, 2, 5, -9 };
```

```
foreach ( int x in e ) Console.WriteLine (x );
```

# Прямоугольные (многомерные) массивы

тип[,] имя;

тип[,] имя = new тип [ разм\_1, разм\_2 ];

тип[,] имя = { список инициализаторов };

тип[,] имя = new тип [,] { список инициализаторов };

тип[,] имя = new тип [ разм\_1, разм\_2 ] { список  
инициализаторов };

```
int[,] a; //элементов нет
int[,] b = new int[2, 3]; // элементы равны 0
int[,] c = {{1, 2, 3}, {4, 5, 6}}; // new подразумевается
int[,] f = new int[,] {{1, 2, 3}, {4, 5, 6}}; // размерность вычисляется
int[,] d = new int[2,3] {{1, 2, 3}, {4, 5, 6}}; // избыточное описание
```

► a[1, 4]

► b[i, j]

```
int[,] a = { { 1, 2 }, { 2, 3 } };
    foreach (var x in a)
        Console.WriteLine("\t" + x);
```

# Ступенчатые,зубчатые jagged массивы

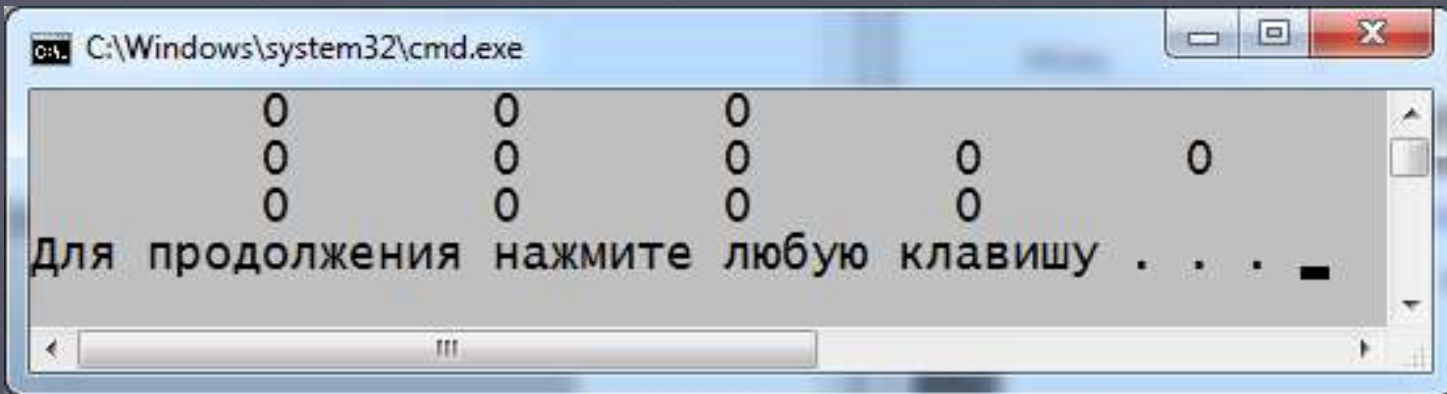
тип[][] имя;

```
int[][] a = new int[3][]; // выделение памяти под ссылки на три строки
a[0] = new int[5]; // выделение памяти под 0-ю строку (5 элементов)
a[1] = new int[3]; // выделение памяти под 1-ю строку (3 элемента)
a[2] = new int[4]; // выделение памяти под 2-ю строку (4 элемента)
```

```
int[][] b = { new int[5], new int[3], new int[4] };
```



```
int[][] a = { new int[3], new int[5], new int[4] };  
foreach (int[] x in a)  
{  
    foreach (int b in x)  
        Console.Write("\t" + b);  
  
    Console.WriteLine();  
}
```



C:\Windows\system32\cmd.exe

```
0      0      0      0      0  
0      0      0      0      0  
0      0      0      0  
Для продолжения нажмите любую клавишу . . .
```

# Комбинации размерностей массивов

```
//принципиально знание характеристик первой составляющей!  
int val1 = 100;  
int [][] x0 = new int[15][];  
int [][][] x1 = new int[val1][][];  
int [,][] x2 = new int[val1,7][];  
int [,,][,][] x3 = new int[2,val1,7][,][];
```

```
int[, ,] arr0;  
int[, ,] arr1;  
  
// Объявлена ссылка на  
// ОДНОМЕРНЫЙ(!) массив  
// ОДНОМЕРНЫХ(!) элементов массива, каждый из которых является  
// ОДНОМЕРНЫМ(!) массивом элементов типа int.  
int[][][] arr3;  
// Объявлена ссылка на  
// ОДНОМЕРНЫЙ(!) массив составляющих, каждая из которых является  
// ДВУМЕРНЫМ(!) массивом массивов элементов типа int.  
int[][,] arr4;  
// Объявлена ссылка на  
// ДВУМЕРНЫЙ(!) массив составляющих, каждая из которых является  
// ОДНОМЕРНЫМ(!) массивом элементов типа int.  
int[,][] arr5;
```

## ► Цикл *foreach*

Цикл `foreach` предназначен для перебора элементов в контейнерах, в том числе в массивах.

```
foreach (тип_иден. название_иден. in контейнер)

    { операторы }
```

цикл работает только на чтение, но не на запись элементов (наполнять нельзя)

Д.б. поддержка интерфейса `IEnumerable`

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

```
String[] collection =
```

```
    new String[]  
    { "1-й элемент",  
      "2-й элемент",  
      "3-й элемент"  
    };
```

```
//Последовательно выводим в консоль элементы массива
```

```
foreach (String element in collection)  
{  
    Console.WriteLine(element);  
}
```

```
//Цикл for, выполняющий аналогичные действия
```

```
for (int i = 0; i < collection.Length; ++i)  
{  
    Console.WriteLine(collection[i]);  
}
```

```
int[][] numbers = new int[3][];  
    numbers[0] = new int[] { 1, 2 };  
    numbers[1] = new int[] { 1, 2, 3 };  
    numbers[2] = new int[] { 1, 2, 3, 4, 5 };
```

```
foreach (int[] row in numbers)  
{  
    foreach (int number in row)  
    {  
        Console.Write($"{number} \t");  
    }  
    Console.WriteLine();  
}
```

# Кортежи

**Кортежи (*tuple*)** комбинируют объекты различных типов (от одного до восьми). Типы и выражения

## ► NuGet пакет

The screenshot shows the NuGet Package Manager interface in Visual Studio. The search bar contains 'ValueTuple'. The results list includes 'System.ValueTuple' by Microsoft, 'ValueTupleBridge' by Nobuyuki Iwanaga, 'Theraot.Core' by Theraot, and 'TupleExtensions.VictorGavrish' by Victor Gavrish. The right pane shows details for 'System.ValueTuple' v4.4.0, including a table of installed versions.

NuGet — решение ▾ X App.config Program.cs Обозреватель объектов

Обзор Установлено Обновления Консолидировать Управление пакетами для решения

ValueTuple x ↻ ☐ Включить предварительные версии Источник пакета: nuget.org

**System.ValueTuple** автор: Microsoft, Скачиваний: 1,92M v4.4.0  
Provides the System.ValueTuple structs, which implement the underlying types for tuples in C# and Visual Basic.

**ValueTupleBridge** автор: Nobuyuki Iwanaga, Скачиваний: 360 v0.1.2  
Backporting of System.ValueTuple for .NET 3.5.

**Theraot.Core** автор: Theraot, Скачиваний: 1,16K v1.0.3  
.NET Backport (ValueTask, ValueTuple, Task, Expressions, Linq, ThreadLocal, etc...) for .NET 2.0, 3.0, 3.5, 4.0, 4.5

**TupleExtensions.VictorGavrish** автор: Victor Gavrish, Скачиваний: 82K v1.2.3  
A collection of convenience extensions that leverages C# 7 tuples

Все пакеты лицензируются их владельцами. NuGet не несет ответственности за пакеты сторонних производителей и не предоставляет лицензии на такие пакеты.

☐ Больше не показывать

**System.ValueTuple**

Версии — 1

| <input checked="" type="checkbox"/> | Проект ^ | Версия |
|-------------------------------------|----------|--------|
| <input checked="" type="checkbox"/> | OOP_Lect | 4.4.0  |

Установлено: 4.4.0 Удалить

Версия: Последняя ста Установить

## ► C# 7.0 tuple литералы

```
ValueTuple < string, int> student = ("Olga", 19);
```

```
(string, string, int) namesAndAge = ("Olga", "Krol", 22);
```

```
Console.WriteLine(student.GetType().Name);
```

```
Console.WriteLine(namesAndAge.GetType().Name);
```

```
Console.WriteLine($" {student}");
```

```
Console.WriteLine($" {namesAndAge}");
```

```
ValueTuple`2
```

```
ValueTuple`3
```

```
(Olga, 19)
```

```
(Olga, Krol, 22)
```



## ► C# 7.0 именованное элементов

```
var names = (first: "Никита", second: "Грицевич");  
string firstName = names.first;  
string secondName = names.second;
```

## ► Именованное при объявлении

```
(string first, string second) names2 =  
    ( "Никита", "Крицевич");
```

# Tuple<T1,T2,T3,T4,T5,T6,T7,TRest>

```
static Tuple<int, string, char> CreateCortage(string name)
{
    int len = name.Length;
    string s = "My first name is" + name;
    char ch = (char)(name[0]);
    return Tuple.Create<int, string, char>(len, s, ch);
}

static void Main(string[] args)
{
    var someTuple = CreateCortage("Anna");
    Console.WriteLine(someTuple.Item1.ToString(),
        someTuple.Item2, someTuple.Item3);

    var someTuple2 = Tuple.Create< decimal,
        Tuple<int,string, char>> (12, someTuple);
}
```

(по умолчанию они называются Item1, Item2)

## Свойства:


- ▶ создается один раз и остается неизменным (все свойства доступны только для чтения)
- ▶ позволяют использовать методы `CompareTo`, `Equals`, `GetHashCode` и `ToString`, свойство `Size`
- ▶ реализуют интерфейсы `IStructuralEquatable`, `IStructuralComparable` и `IComparable` (можно сравнивать)

# C# 7

## Кортежи (tuple)



```
(int, string, char) CreateCortage(string name)
{
    int len = name.Length;
    string s = "My first name is" + name;
    char ch = (char)(name[0]);
    return (len, s, ch);
}
```



## Литерал кортежа

Кортежи - значимый тип,  
*System.ValueTuple*.  
элементы - открытые поля,  
могут сравниваться на равенство,  
присвоится др.др

## ► Можно дать данным имена

```
(int length, string fullName, char firstLetter)  
CreateCortage(string name){  
}
```

***Или так***

```
return (f:len, s:s, t:ch);
```

## ► Обращение по имени

```
var iGetIt = CreateCortage(name);  
Console.Write("получено {iGetIt.fullName}");
```

## ► Распаковка кортежей

```
var (one, two, three) = CreateCortage(name);
```

# C# 7

## Локальные функции

- ▶ вспомогательная функция - внутри метода, в котором вызывается

```
public int Method(int x)
{
    return LocFun(x).current;

    int LocFun(int i)
    {
        if (i == 0) return 0;
        var p = LocFun(i - 1);
        return p+1;
    }
}
```

Аргументы внешнего метода и его локальные переменные доступны для локальной функции