Основы C#

Code convention
Станда́рт оформле́ния ко́да

набор правил и соглашений, используемых при написании исходного кода на некотором языке программирования.

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions

Code convention

стили написания составных слов

PascalCase	*IasdGen*
camelCase	*iasdGenTyi*
UPPERCASE	*ID*
Hungarian notation	*str*Name, *i*Year

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions

Суффиксы и префиксы

- имена пользовательских классов исключений всегда заканчиваются суффиксом “Exception”;

```
public class SampleException: System.Exception
{
  public SampleException()
```

- имена интерфейсов всегда начинаются с префикса «I»;

```
interface ISample
{
  void SampleMethod();
}
```

- имена пользовательских атрибутов всегда заканчиваются суффиксом «Attribute»;

```
[System.AttributeUsage(System.AttributeTargets.All, Inherited = false, AllowMultiple = true)]
sealed class SampleAttribute: System.Attribute
{
 public SampleAttribute()
```

- имена делегатов обработчиков событий всегда оканчиваются суффиксом EventHandler, имена классов-наследников от EventArgs всегда заканчиваются суффиксом EventArgs.

```
  }
} public delegate void AnswerCreatedEventHandler(object sender, AnswerCreatedEventArgs
e);
```

1)имена типов, структур, перечислений, интерфейсов, методов, свойств – pascal case

SampleClass ISampleInterface SampleMethod();

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

sampleArgument

3) Закрытые поля

private int m_SamplePrivateField;

private int mSamplePrivateField;

private int _samplePrivateField;

4) Функции и методы - **pascal case**

void HelloWorld();

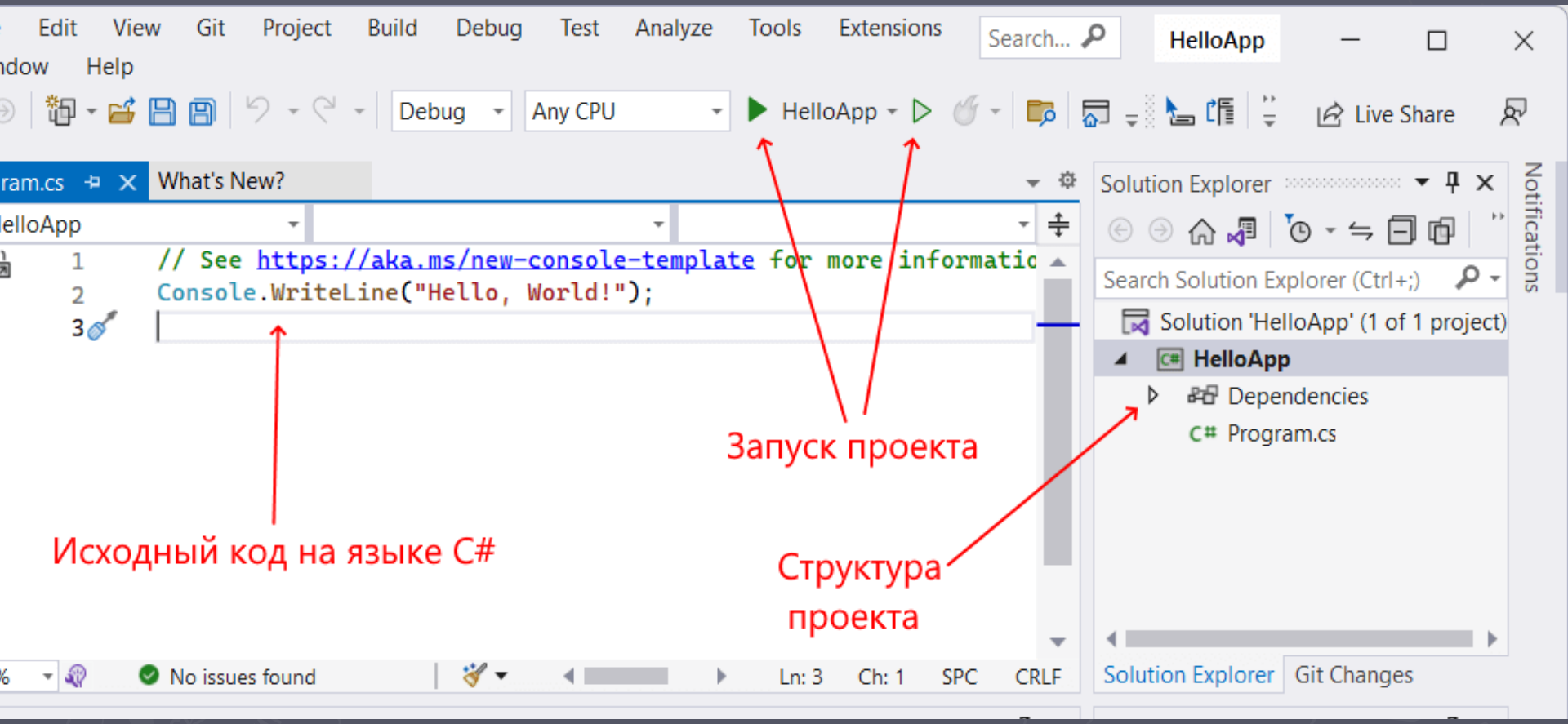
Имя функции начинается с глагола, указывающего на то, какое действие она выполняет

5) Константы - **pascal case**

Форматирование

- В одном файле не объявляется больше одного namespace’а 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’а – 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;


Edit View Git Project Build Debug Test Analyze Tools Extensions
Search...
HelloApp
Help
Debug
Any CPU
HelloApp
Live Share
What's New?
// See https://aka.ms/new-console-template for more informatic
Console.WriteLine("Hello, World!");
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'HelloApp' (1 of 1 project)
HelloApp
Dependencies
Program.cs
Notifications
No issues found
Ln: 3
Ch: 1
SPC
CRLF
Solution Explorer
Git Changes
Запуск проекта
Исходный код на языке C#
Структура проекта

► *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.
namespace ConsoleApplication
{
    static class EntryPoint
    {
        static void Main()
        {
            MyCompany.Utilities.SomeUtility su1 =
                new MyCompany.Utilities.SomeUtility();
            //аналогично, благодаря using
            SomeUtility su2 = new SomeUtility();

    }
}
```

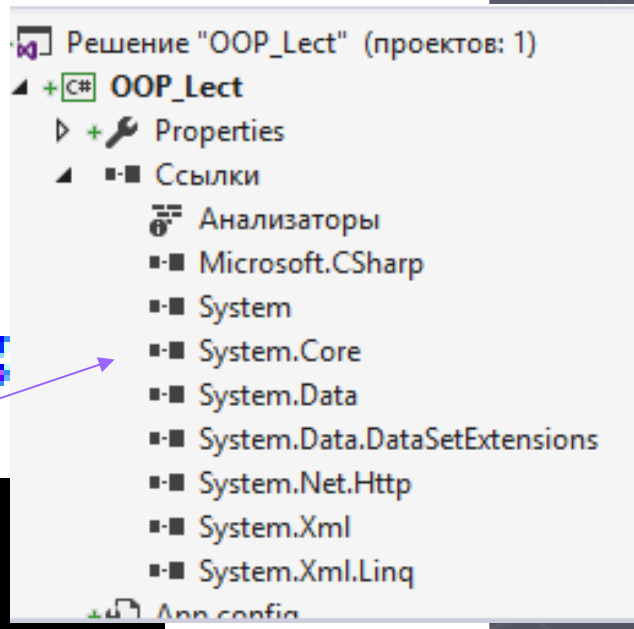
не хватает лаконичности

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

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

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string
        {
        }
    }
}
```

содержит фундаментальные и базовые классы платформы .NET

описаны типы для обработки строк



Физически пространства имен находятся в подключаемых библиотеках dll

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

псевдонимы (alias) имен

```
1  using printer = System.Console;
2
3
4  printer.WriteLine("Laudate omnes gentes laudate");
   printer.WriteLine("Magnificat in secula");
```

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

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}");
```

Выражения 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

```
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;
```

```
int a = 0;           // Самый удобный синтаксис
System.Int32 a = 0; // Удобный синтаксис
int a = new int();  // Неудобный синтаксис
System.Int32 a = new System.Int32(); // Самый неудобный синтаксис
```

- Примитивные типы соответствуют
  е.типы.FCL

Тип C#	Размер 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
1.0×10^{-28} до 7.9×10^{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

CLR поддерживает две разновидности типов:

ссылочные (reference types)
значимые (value types).

► 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

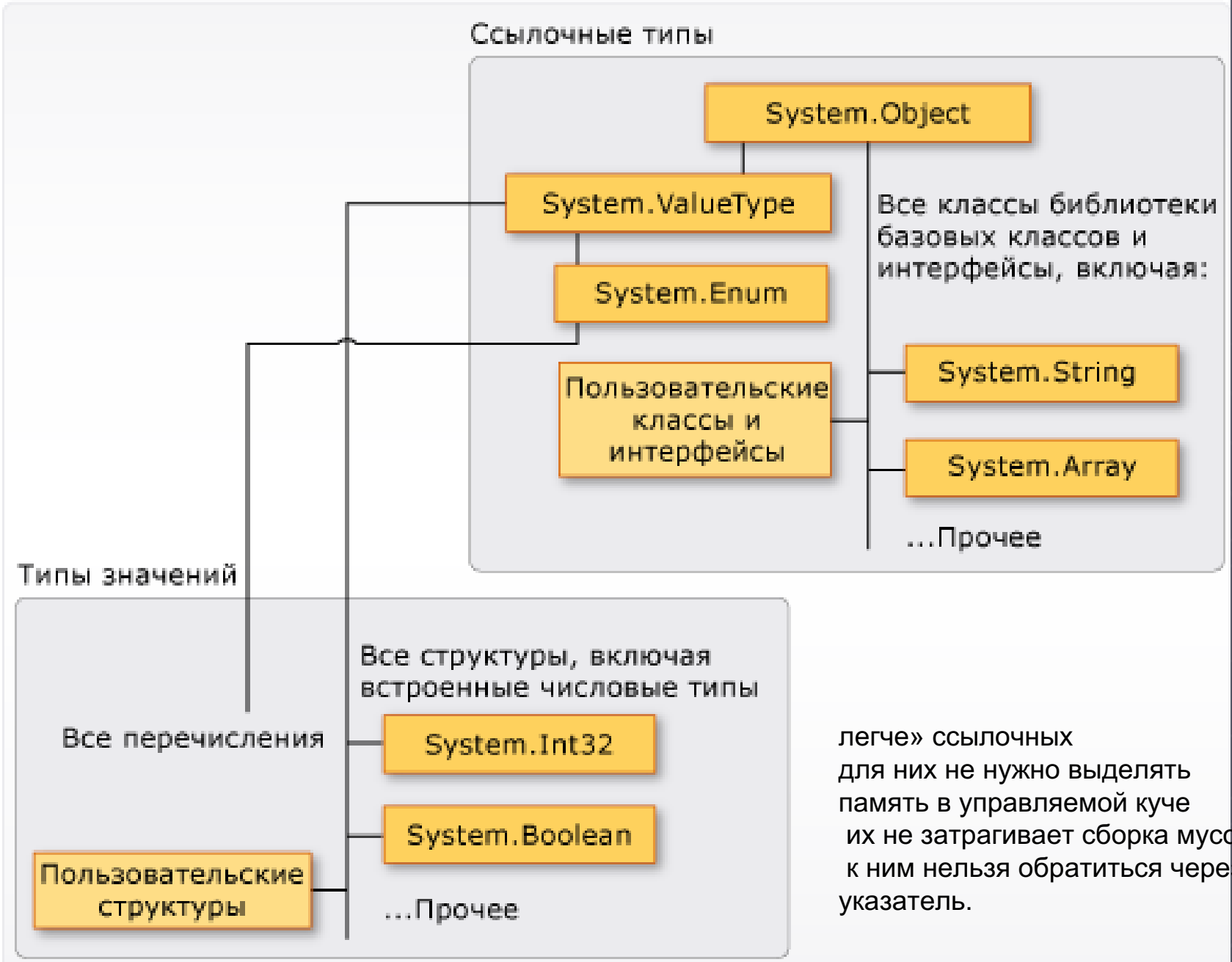
```
Something objSomething;
objSomething = new Something();
```

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
    {
//...
}
```



легче» ссылочных
для них не нужно выделять
память в управляемой куче
их не затрагивает сборка мусора
к ним нельзя обратиться через
указатель.

Система типов C#

Типы значений
(value types)
1) Определяются
struct или emun
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)

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

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

```
Int32 x = 5;
 Object o = x;
// Упаковка x; o ссылается на упакованный объект
```

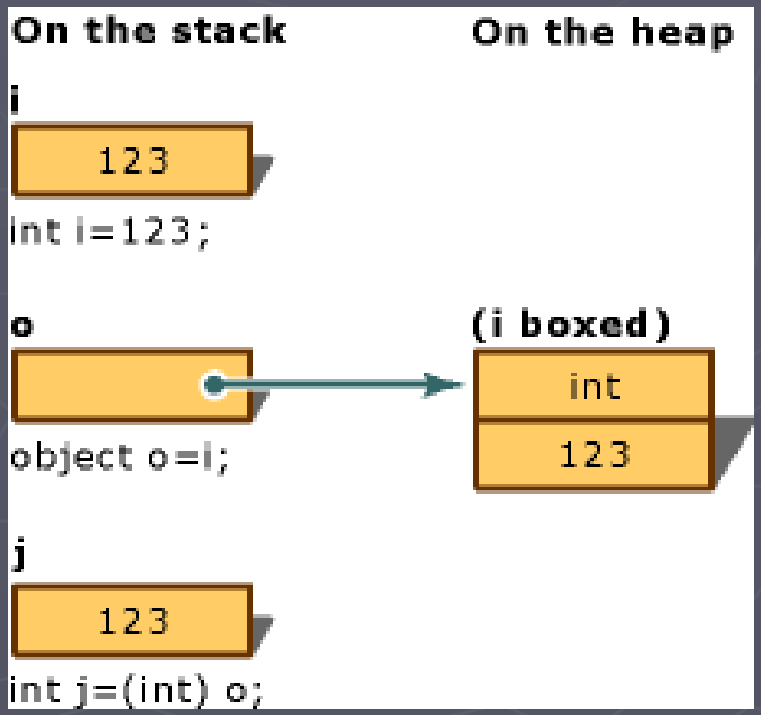
- распаковка (unboxing)

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.

Упаковка и распаковка значимых типов

```
int i = 123;
object o = i;
int k = (int)o;
```



При упаковке экземпляра значимого типа происходит следующее.

1. В управляемой куче выделяется память. Ее объем определяется длиной значимого типа и двумя дополнительными членами — указателем на типовой объект и индексом блока синхронизации. Эти члены необходимы для всех объектов в управляемой куче.
2. Поля значимого типа копируются в память, только что выделенную в куче.
3. Возвращается адрес объекта. Этот адрес является ссылкой на объект, то есть значимый тип превращается в ссылочный.

```
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 NullReferenceException.

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, так как интерфейсные переменные всегда должны содержать ссылку на объект в куче

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.

Работает ли данный код?

```
public static void Main() {
   Int32 x = 5;
   Object o = x;
   Int16 y = (Int16) o;
}
```

```
public static void Main() {
  Int32 x = 5;
  Object o = x;         // Упаковка x; o указывает на упакованный объект
  Int16 y = (Int16) o; // Генерируется InvalidCastException
}
```

при распаковке объекта должно быть выполнено приведение к неупакованному типу

```
public static void Main() {
  Int32 x = 5;
  Object o = x; // Упаковка x; o указывает на упакованный объект
  Int16 y = (Int16)(Int32) o; // Распаковка, а затем приведение типа
}
```

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 может привести к 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)

Инициализация переменных по умолчанию

для ссылок на объекты - null

тип значений - в ноль

```
Person a;
int c;

String aStr = a.ToString();
```

Стек

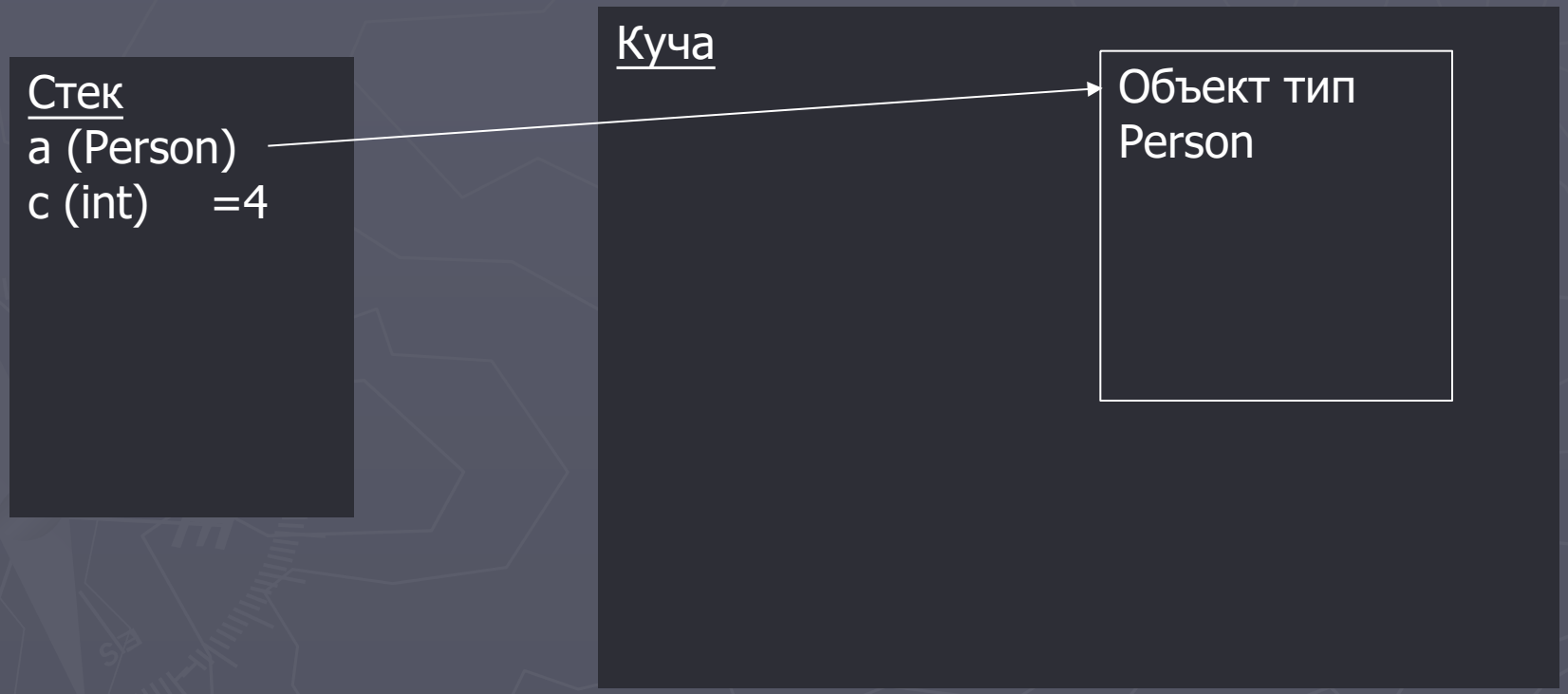
a (Person) → null

c (int) → 0

(локальная переменная) Person a

Использование локальной переменной "a", которой не присвоено значение.

```
Person a;
int c = 4;

a = new Person();
```



Локальная переменная по ссылке

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

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

Для определения локальной переменной-ссылки (ref local) перед ее типом ставится ключевое слово **ref**:

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

Создает ссылочную переменную, которая инициализируется ссылкой

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

```
ref int xRef;    // ошибка
```

Тип данных dynamic

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

```
Int32 demo = 1; //0

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

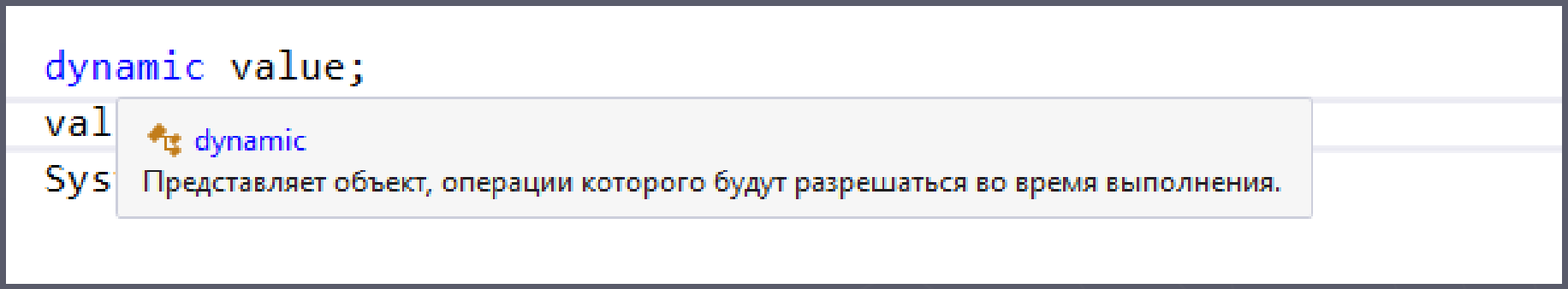
Int32: 10
String: AA

- Что происходит

```
dynamic obj = 3;            // здесь obj - целочисленное int
Console.WriteLine(obj);     // 3

obj = "Hello world";        // obj - строка
Console.WriteLine(obj);     // Hello world
```

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

НО!!!! корректность указываемых членов компилятором не проверяется!

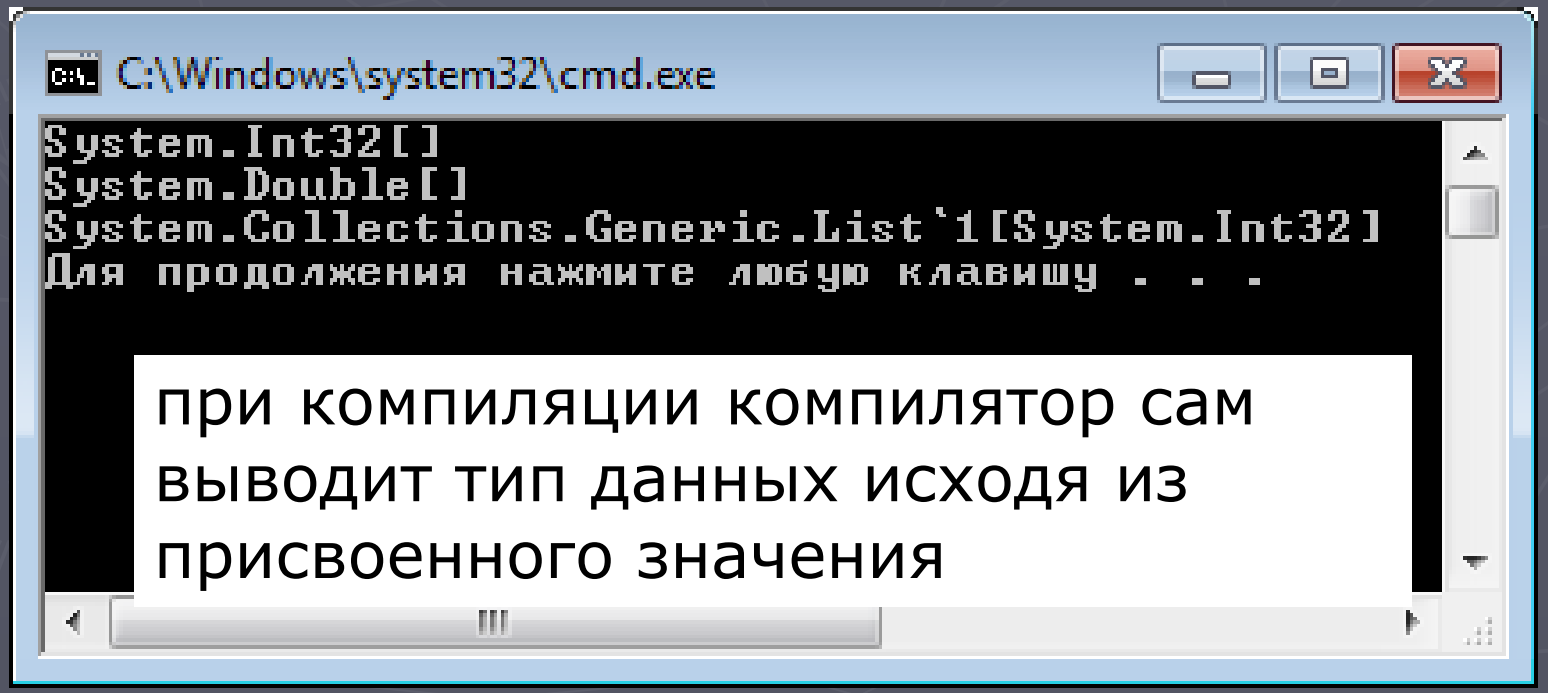


- компилятор конвертирует System.Object
- применяет атрибут System.Runtime.CompilerServices.DynamicAttribute

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();
```



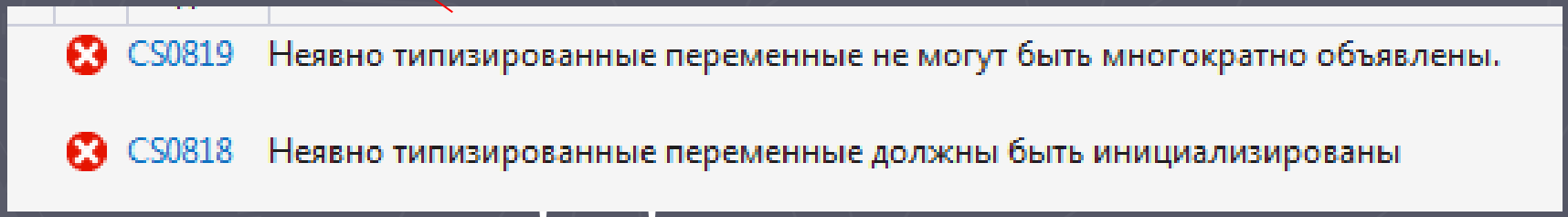
```
// Этот код не компилируется
??? unnamedTypeVar = new {firstArg = 1, secondArg = "Alex"};
```

```
var unnamedTypeVar = new { firstArg = 1, secondArg = "Alex" };
Console.WriteLine(unnamedTypeVar.firstArg + ". " + unnamedTypeVar.secondArg);
```

Ограничения:

- должно включать инициализатор (null нельзя)

```
int i1, i2 = 4; //допускается
var v1, v2 = 2; //вызовет ошибки CS0810 и CS081
```



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

типы Nullable

```
int x = null;
```

Не удается 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

структура **Nullable<T>** имеет два свойства:

- **Value** - значение объекта
- **HasValue**: возвращает true, если объект хранит некоторое значение, и false, если объект равен null.

```
int? x1 = null;
int? x2 = null;
System.Console.Write(x1 == x2); //True
```

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

Оператор ?? (null-объединение)

```
левый_операнд ?? правый_операнд
```

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

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

применяется для установки значений по умолчанию для типов значений и ссылочных типов, которые допускают значение null

возвращает левый операнд, если этот операнд не равен `null`
Иначе возвращается правый операнд
Для nullable-типов

```
int? id = 100;
id ??= 1;
// аналогично
//id = id ?? 1;
Console.WriteLine(id); // 100
```

Целые числа собственного размера C#9

nint	Зависит от платформы (вычисленной во время выполнения)	32- или 64-разрядное целое число со знаком	System.IntPtr
nuint	Зависит от платформы (вычисленной во время выполнения)	32- или 64-разрядное целое число без знака	System.UIntPtr

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!!!!!!!!

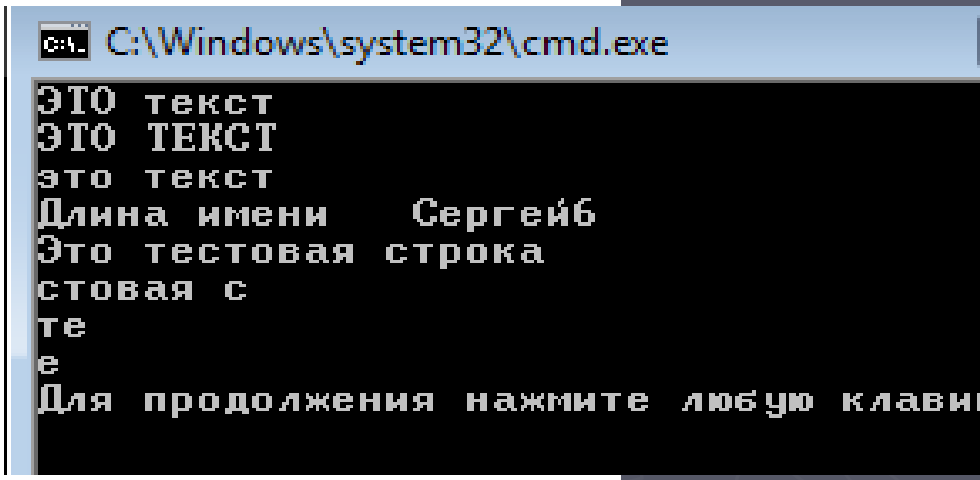
- Строки равны, если имеют одинаковое количество символов и совпадают посимвольно.
- Обращаться к отдельному элементу строки по индексу можно только для получения значения, но не для его изменения.
- строки типа string относятся к неизменяемым типам данных.
- Методы, изменяющие содержимое строки, на самом деле создают новую копию строки. Неиспользуемые «старые» копии автоматически удаляются сборщиком мусора.

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 (>0, 0, >0)
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);
```



извлекает из строки подстроку, начиная с указанной позиции

String. IndexOf()

```
// 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, '-'));
```

```
Это тестовая строка

Первый индекс 'о': 2
Первый индекс 'т': 1
Первый индекс 'r': -1

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

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);

    }
}
```

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

```
double number = 23.7;
string result = string.Format("{0:C0}", number);
Console.WriteLine(result); // 24 р.
string result2 = string.Format("{0:C2}", number);
Console.WriteLine(result2); // 23,70 р.
```

сколько чисел будет использоваться после разделителя между целой и дробной частью

денежного знака для текущей культуры компьютера

Класс StringBuilder

```
1  using System.Text;
2
3  StringBuilder sb = new StringBuilder();
```

Можно сразу инициализировать объект определенной строкой:

```
1  StringBuilder sb = new StringBuilder("Привет мир");
```

С помощью метода ToString() мы можем получить строку, которая хранится в StringBuilder

```
1  var sb = new StringBuilder("Hello World");
2  Console.WriteLine(sb.ToString());    //  Hello World
```

Либо можно просто передать объект StringBuilder:

```
1  var sb = new StringBuilder("Hello World");
2  Console.WriteLine(sb);    //  Hello World
```

Класс StringBuilder

- Любые модификации строки происходят внутри блока памяти
  - *Length* - длина строки
  - *Capacity* - максимальная длина строки,

```
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, 'заключённую' 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`

```
1  using System.Text;
2
3  StringBuilder sb = new StringBuilder("Привет мир");
4  Console.WriteLine($"Длина: {sb.Length}");        // Длина: 10
5  Console.WriteLine($"Емкость: {sb.Capacity}");    // Емкость: 16
```

по умолчанию 16 символов

при создании строки StringBuilder выделяет памяти больше, чем необходимо этой строке.
При увеличении строки в StringBuilder, когда количество символов превосходит начальную емкость, то емкость увеличивается в два и более раз.

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[] numbers = { 1, 2, 3, 5 };

// получение элемента массива
Console.WriteLine(numbers[3]);  // 5
```

Перебор массивов

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

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

for более гибкий по сравнению с **foreach**

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

foreach последовательно извлекает элементы контейнера и только для чтения, то в цикле for мы можем перескакивать на несколько элементов вперед в зависимости от приращения счетчика, а также можем изменять элементы

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

```
// Объявляем двумерный массив
int[,] myArr = new int[4, 5];

Random ran = new Random();

// Инициализируем данный массив
for (int i = 0; i < 4; i++)
{
    for (int j = 0; j < 5; j++)
    {
        myArr[i, j] = ran.Next(1, 15);
        Console.Write("{0}\t", myArr[i, j]);
    }
    Console.WriteLine();
}
```

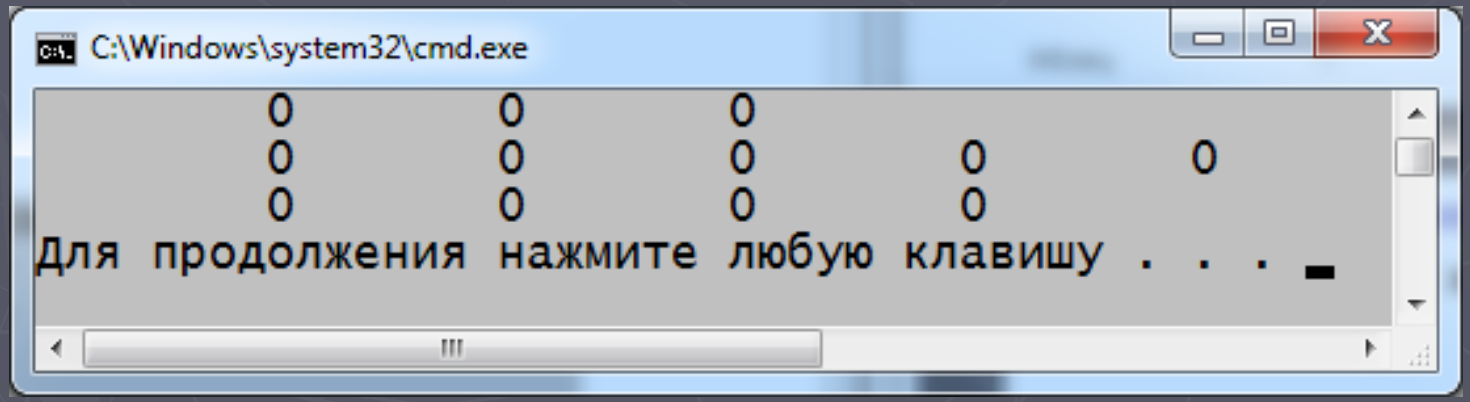
```
5       2       3       14      4
13      8       10      14      1
14      5       10      6       1
2       7       7       11      1
```

Ступенчатые,зубчатые jagged массивы

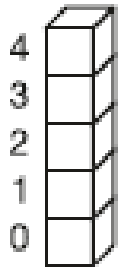
тип[][] имя;

```
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();
            }
```

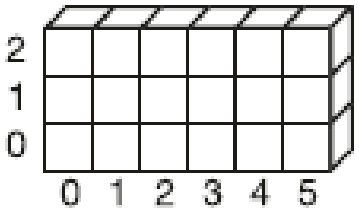


Одномерный массив

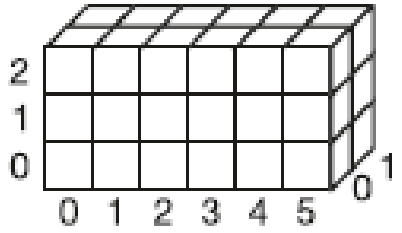


Одномерный массив
int[5]

Многомерные массивы

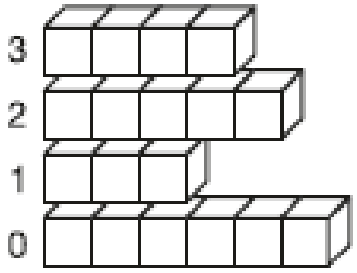


Двухмерный массив
int[3,6]



Трехмерный массив
int[3,6,2]

Зубчатый массив



Зубчатый массив
int[4][]

```
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. реализация 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]);
            }
```

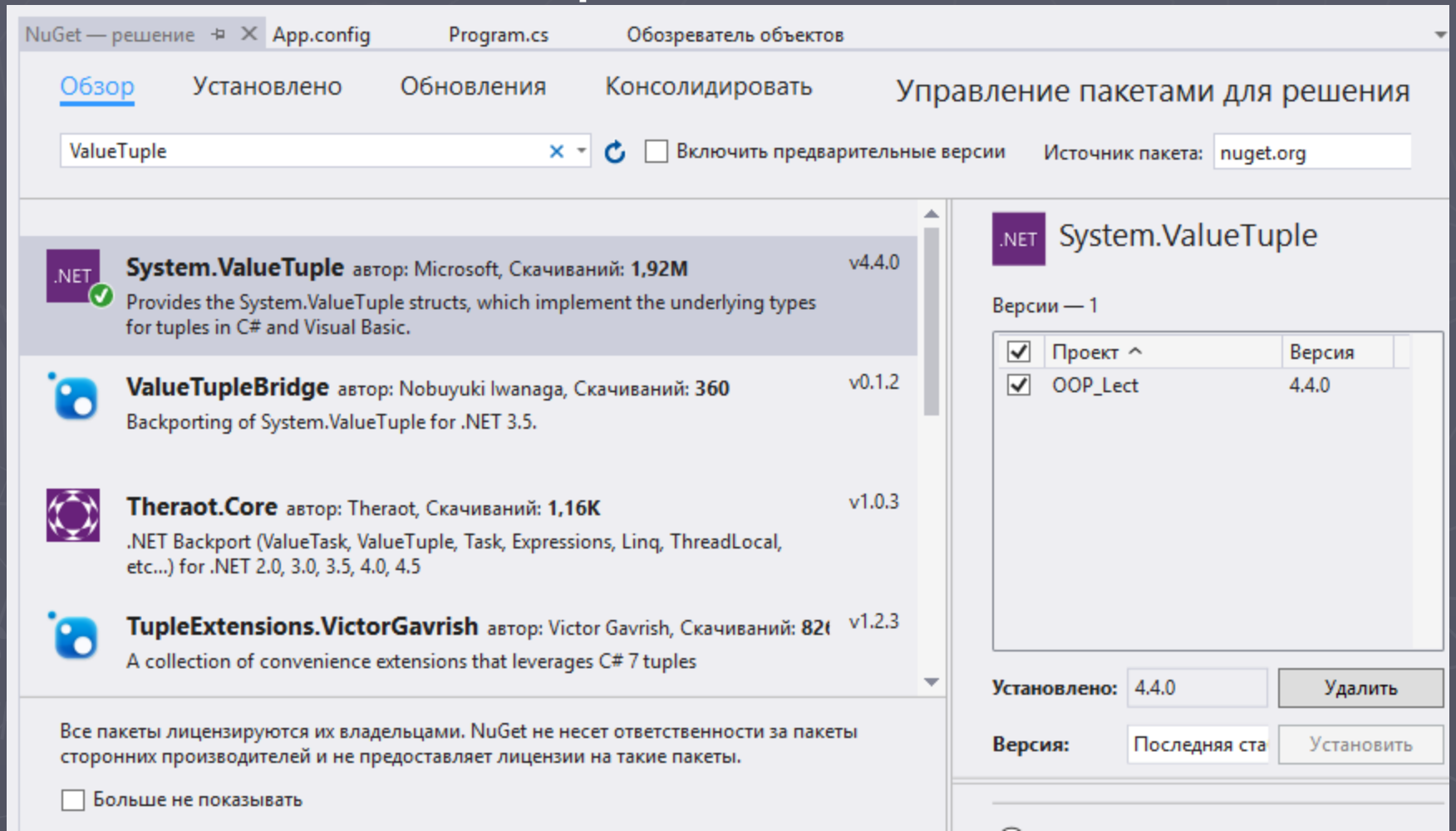
```
int[][] numbers = new int[3][];
            numbers[0] = new int[] { 1, 2 };
            numbers[1] = new int[] { 1, 2, 3 };
            numbers[2] = new int[] { 1, 2, 3, 4, 5 };



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

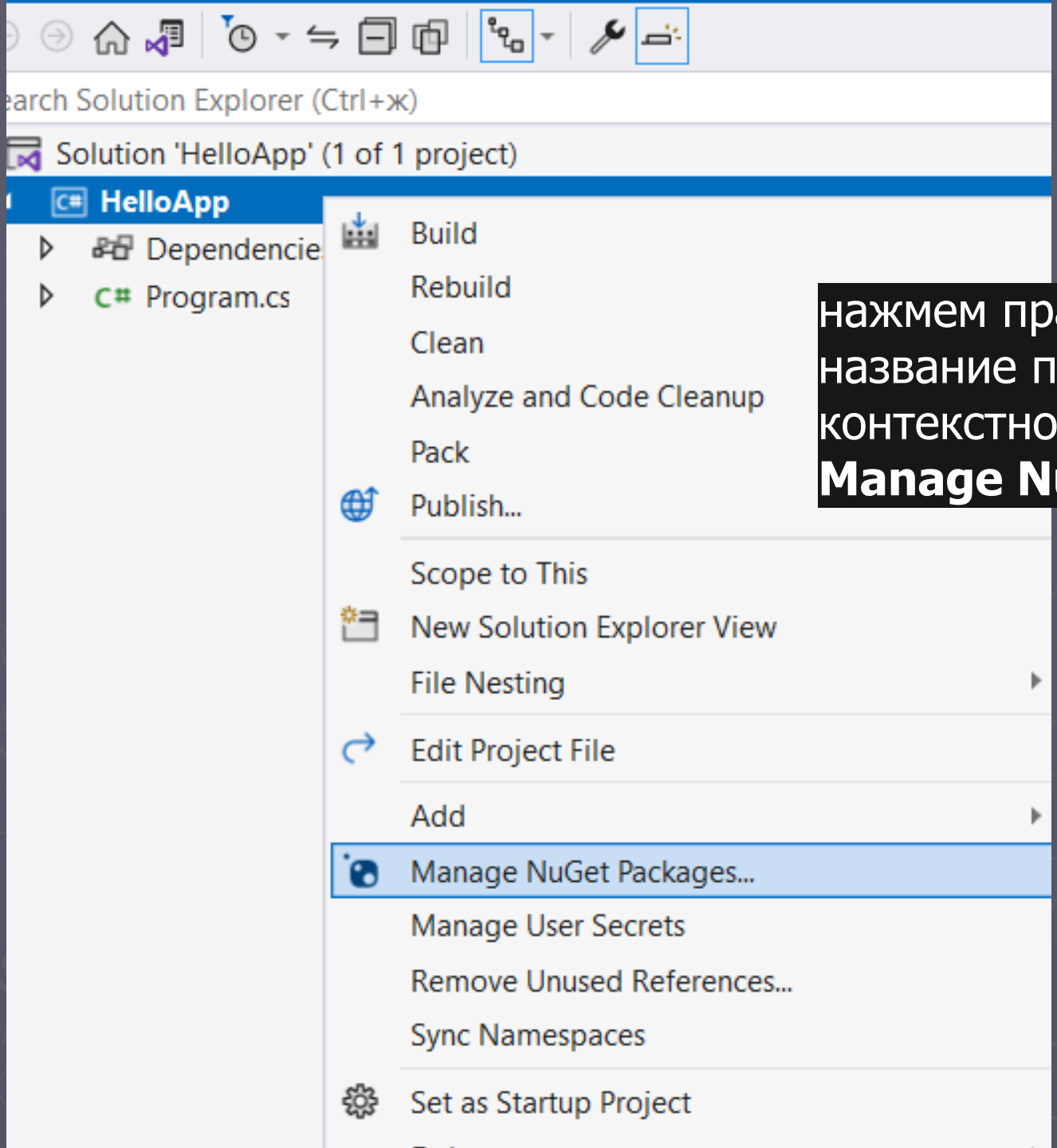
менеджер пакетов

Кортежи

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

- NuGet менеджер пакетов





нажмем правой кнопкой на название проекта и в контекстном меню выберем **Manage NuGet Packages...**

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 именование элементов

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

Именование 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 неименным (все свойства доступны только для чтения)
- позволяют использовать методы CompareTo, Equals, GetHashCode и ToString, свойство Size
- реализуют интерфейсы IStructuralEquatable, IStructuralComparable и IComparable (можно сравнивать)

- 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

элементы, в которых можно объявлять и из которых можно вызывать локальные функции

- . Методы, в частности методы итератора и асинхронные методы
- Методы доступа свойств
- Методы доступа событий
- Анонимные методы
- Лямбда-выражения
- Методы завершения
- Другие локальные функции

Модификаторы для локальной функции

- async
- unsafe
- static (в C# 8.0 и более поздних версий).

Статическая локальная функция не может сохранять локальные переменные или состояние экземпляра.

- extern (в C# 9.0 и более поздних версий).

Внешней локальной функцией должна быть static.

Index c#8

- Новый тип C#8

```
Index i = 2;

int number = i.Value;

Console.WriteLine(i + " " + i.IsFromEnd);//2 False
```

сколько элементов нужно отсчитать

булевое значение, показывающее,
нужно ли отсчитывать
от *конца* коллекции а не от начала

```
Index j = new Index(1, true);
var index = ^1;
```

берет первый элемент с конца

```
Index j = new Index(1, true);
var index = ^1;


var collection = new[] { 1, 2, 3, 4, 5 };
collection[index] = 0; // 1, 2, 3, 4, 0
foreach (int a in collection)
{
    Console.WriteLine(a);
}
```

Range C#8

- Линейный направленный по возрастанию диапазон индексов с шагом 1

```
int i1 = 5;
int i2 = 10;
var a = i1..i2; // Range(i1, i2)

var b = i1..; // Range(i1, new Index(0, true));

var c = ..i2; // Range(new Index(0, false), i2)

Range e = ..; // весь диапазон, от первого и до последнего элемента
```