



BASIC CODING IN C# 7.0

.NET LAB. MINSK. 2018

ANZHELIKA KRAVCHUK

Основы типов

Понятие типа

Тип – это именованная абстракция, предназначенная для повторного использования.

Языки программирования содержат синтаксические конструкции, предназначенные для создания типов (язык C# - классы, структуры, перечисления, интерфейсы, делегаты).

Каждая конструкция некоторым образом отображает принципы определения типов CLR. Тип CLR – это именованная абстракция, пригодная для повторного использования. Описание типов CLR находится в метаданных модуля CLR.

Имя типа CLR состоит из трех частей – имени сборки, необязательного префикса, обозначающего имя пространства имен и локального имени типа.

Определение членов типа

Определение типа CLR состоит из определения нулевого или большего количества членов типа (members).

От членов типа зависят способы использования и правила функционирования типа. Для каждого члена типа определен (возможно неявно) модификатор доступа, управляющий доступом к члену типа. Члены типа, доступные извне, обобщенно называются контрактом типа (type contract)

Кроме правил доступа к членам, можно определить, должен ли существовать экземпляр типа для доступа к его члену. Практически все члены можно разделить либо на члены экземпляра, либо на члены типа. Для обращения к члену экземпляра, должен существовать экземпляр типа

Определение членов типа

Существует три фундаментальных вида членов типа – поля, методы и вложенные типы

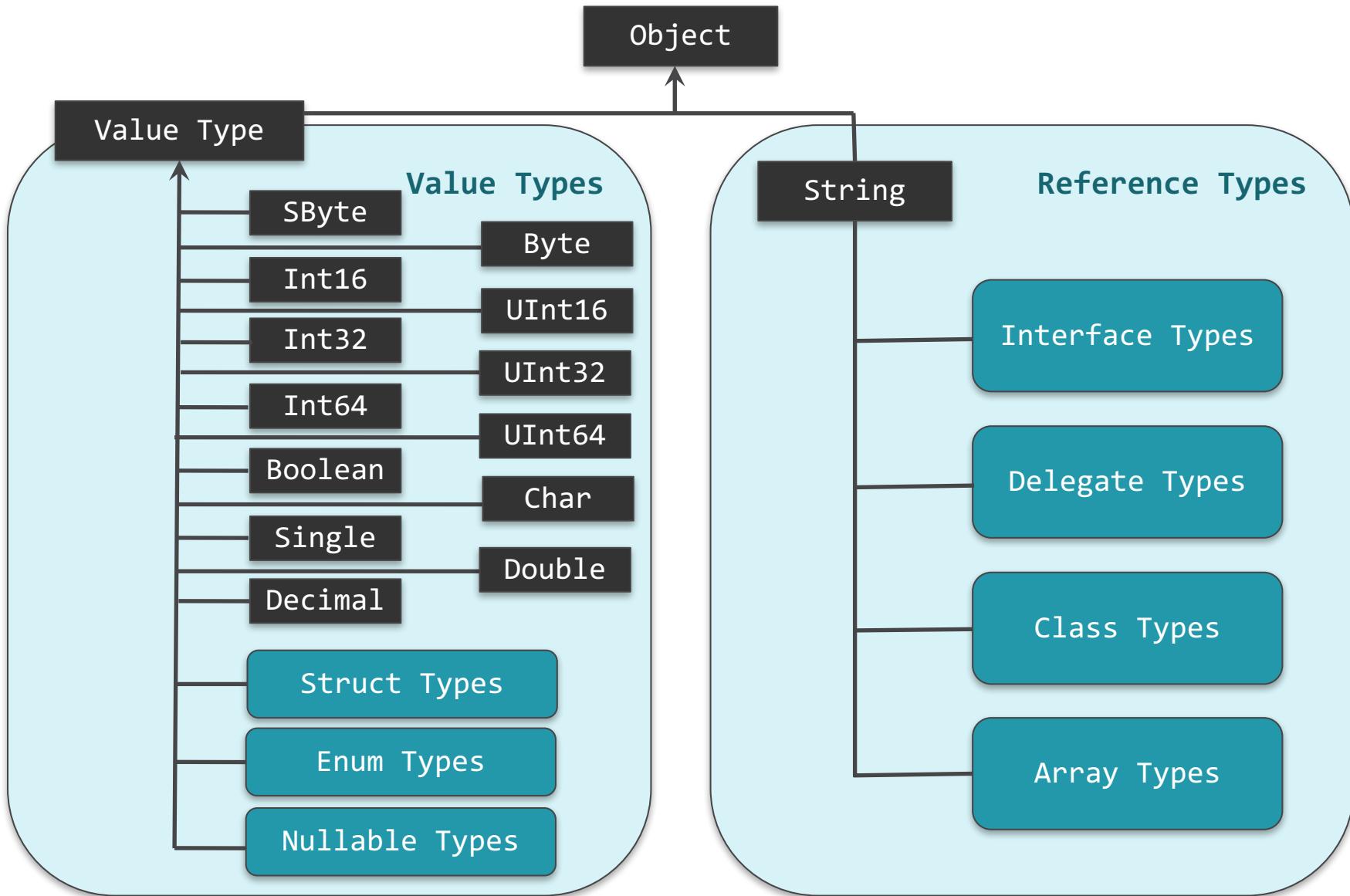
Поле – это именованная единица хранения данных, ассоциированная с типом

Метод – это именованная операция, которую можно вызвать и выполнить

Вложенный тип – это другой тип, определенный как часть реализации объявляющего типа

Другие виды членов типа – события, свойства и т.д. – это методы, расширенные с помощью метаданных!

Иерархия типов .NET Framework



Использование типов

Сами по себе типы используются довольно редко, полезными их делает возможность создания экземпляров. Экземпляр – это объект или значение, в зависимости от определения типа

Экземпляры значимых типов это значения, ссылочных типов – объекты

Каждый объект или каждое значение являются экземпляром только одного типа

Использование типов

Связь между типом и его экземпляром может быть явной или неявной

Объявление локальной переменной или поля значимого типа (например, System.Int32) приводит к выделению в памяти блока, связанного со своим типом только посредством кода, обрабатывающего этот блок. Среда CLR (компилятор и процедура проверки CLR) обеспечивают поддержку связи этого блока памяти с типом после загрузки кода

Каждый объект связан с типом явно. Поскольку объект доступен только посредством ссылки, фактический тип ссылки объекта может не совпадать с объявленным типом ссылки. В подобных ситуациях нужен механизм , явно связывающий объект с его типом – в CLR это заголовок объекта (object header)

Переменные ссылочного и значимого типов

```
int x1 = 42;
```

x1 :

42

System.Int32

```
int x2 = x1;
```

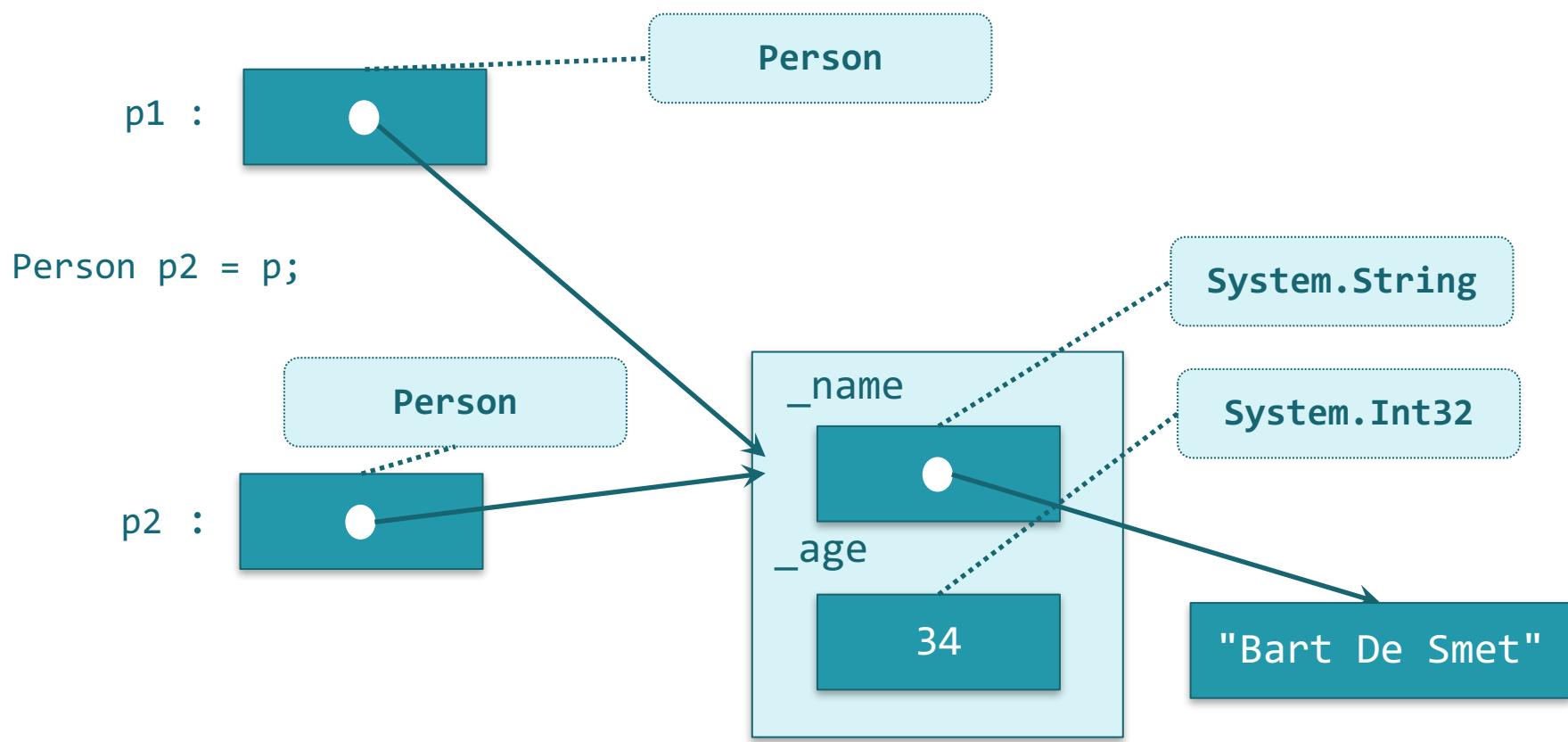
x2 :

42

System.Int32

Переменные ссылочного и значимого типов

```
Person p1 = new Person("Bart De Smet", 34);
```



Понятие переменной

Переменная представляет именованное место в памяти для хранения порции данных

Переменная характеризуется:

- Имя (Name)
- Адрес (Address)
- Тип данных (Data type)
- Значение (Value)
- Область видимости (Scope)
- Время жизни (Lifetime)

Объявление и присваивание переменных

Идентификатор может содержать только буквы, цифры и символы подчеркивания

Идентификатор должен начинаться с буквы или символа подчеркивания

Идентификатор не должен быть одним из ключевых слов, которые C# резервирует для собственного использования

При объявлении переменной для ее хранения должно быть зарезервировано место в памяти, размер которого определяется типом, поэтому при объявлении переменной необходимо указать тип хранимых данных

Объявление и присваивание переменных

Pascal Case

```
DataTypeName variableName;  
// or  
DataTypeName variableName1, variableName2;
```

Camel Case

```
int variableName = value;
```

После объявления переменной можно присвоить значение для его дальнейшего использования в приложении с помощью оператора присваивания

```
int numberOfEmployees;
```

```
numberOfEmployees = "Hello",
```

При объявлении переменной, пока ей не присвоено значение, она содержит случайное значение

Тип выражения при присваивании должен соответствовать типу переменной

Объявление и присваивание переменных

При объявлении переменных вместо указания явного типа данных можно использовать ключевое слово var (*неявная статическая типизация*)

```
var price = 20;
```

Неявную типизацию можно использовать для любых типов, включая массивы, обобщенные типы и пользовательские специальные типы

Неявная типизация применима только для локальных переменных в контексте какого-то метода или свойства

```
class ThisWillNeverCompile
{
    private var someInt = 10;
    public var MyMethod(var x, var y) { }
}
```

Объявление и присваивание переменных

Неявно типизированную локальную переменную можно возвращать вызывающему методу, при условии, что возвращаемый тип этого метода совпадает с типом, лежащим в основе определенных с помощью var данных

```
static int GetAnIntValue()
{
    var retVal = 9;
    return retVal;
}
```

```
var someObj = null; <
```

Локальным переменным, объявленным с помощью ключевого слова var, не допускается присваивать в качестве начального значения null

Область видимости переменной

```
if (length > 10)
{
    int area = length * length;  <----- Block scope
}
```

```
void ShowName()
{
    string name = "Bob";   <----- Procedure scope
}
```

```
private string message;  <----- Class scope

void SetString()
{
    message = "Hello World!";
}
```

Область видимости переменной

```
public class CreateMessage
{
    public string message = "Hello";
}
. . .
public DisplayMessage
{
    public void ShowMessage()
    {
        CreateMessage newMessage = new CreateMessage();
        MessageBox.Show(newMessage.message);
    }
}
```



Namespace scope

Примитивные типы C#. Целочисленные типы

C#	BCL тип	Диапазон
sbyte	System.SByte	-128 to 127
byte	System.Byte	0 to 255
short	System.Int16	-32,768 to 32,767
ushort	System.UInt16	0 to 65,535
int	System.Int32	-2,147,483,648 to 2,147,483,647
uint	System.UInt32	0 to 4,294,967,295
long	System.Int64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
ulong	System.UInt64	0 to 18,446,744,073,709,551,615
char	System.Char	0 to 65,535 (U+0000 to U+ffff)

Примитивные типы C#. Типы с плавающей точкой

C#	BCL тип	Диапазон	Точность
float	System.Single	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	23 bits (~7 decimal digits)
double	System.Double	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	52 bits (~15 decimal digits)

Примитивные типы C#. Тип Decimal

C#	BCL тип	Диапазон	Точность
decimal	System.Decimal	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9 \times 10^{28}$	28-29 digits

Примитивные типы C#. Тип Boolean

C#	BCL тип	Значения
bool	System.Boolean	false, true

Примитивные типы C#. Тип String

C#	BCL тип	Значения
string	System.String	Последовательность символов System.Char

Примитивные типы C#. Тип Object

C#	BCL тип	Диапазон
object	System.Object	

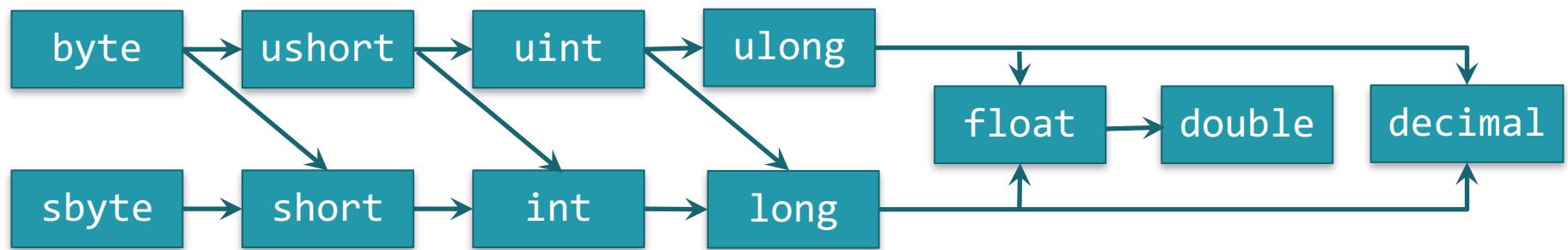
Примитивные типы C#. Тип dynamic

C#	BCL тип	Диапазон
dynamic	System.Object	

Преобразование типов данных

Неявное преобразование (implicit conversion)

Не требует особых синтаксических конструкций и осуществляется компилятором



Преобразование типов данных

Явное преобразование (explicit conversion), требует операции приведение (casting)

Требует, чтобы был написан код для выполнения преобразования, которое, в противном случае, может привести к потере информации или ошибке

```
DataType variableName1 = (DataType)variableName2
```

```
long l = 12345;
```

```
int i = l;
```

```
int i = (int)l; <----- casting
```

Константы и переменные только для чтения

Константы

- Используются только для хранения неизменяемых данных
- Объявляются с помощью ключевого слова const
- Значение можно инициализировать только во время разработки

```
const DataType CONSTANTNAME= Value;
```

```
const double PI = 3.14159;
```

```
int radius = 5;
```

```
double area = PI * radius * radius;
```

```
double circumference = 2 * PI * radius;
```



Upper Case

Константы и переменные только для чтения

Переменные только для чтения (read-only)

- Используются только для хранения неизменяемых данных
- Объявляются с помощью ключевого слова `readonly`
- Значение можно инициализировать во время выполнения

```
readonly DataType variableName = Value;
```

```
readonly string currentTime = DateTime.Now.ToString();
```

Выражения и операции в C#

Выражения. Операции. Арность. Ассоциативность. Приоритет

Выражения фундаментальная конструкция, используемая для вычисления и управления данными

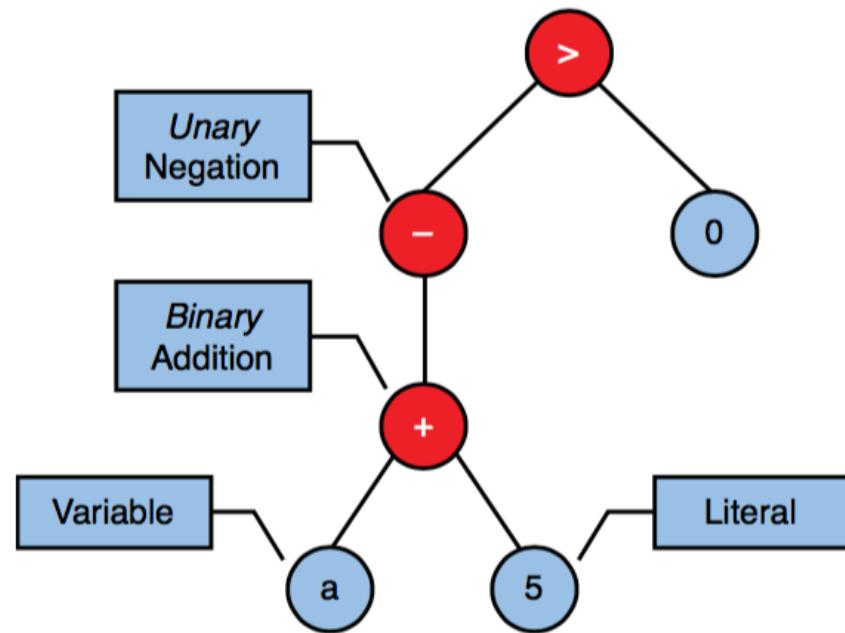
Выражения являются комбинацией operandов и операций

`a + 1`

`(a + b) / 2`

`"Answer: " + c.ToString()`

`b * System.Math.Tan(theta)`



`-(a + 5) > 0`

Основные операции

- Арифметические +, -, *, /, %
- Инкремент, декремент ++, --
- Сравнение ==, !=, <,>, <=, <=, is
- Логические &&, !, ||
- Индексация []
- Приведение (), as
- Присваивание =, +=, -=, *=, =,% = . . .
- Битовые сдвиг ~, |, &, ^, <<, >>
- Информация о типе SizeOf, TypeOf
- Конкатенация и удаление делегатов +, -
- Контроль за переполнением checked, unchecked
- Разыменования и получения адреса *, ->, [], &
- Условная ?:
- Поглощения ?? (null-coalescing)
- Условного null (null-conditional) ?.

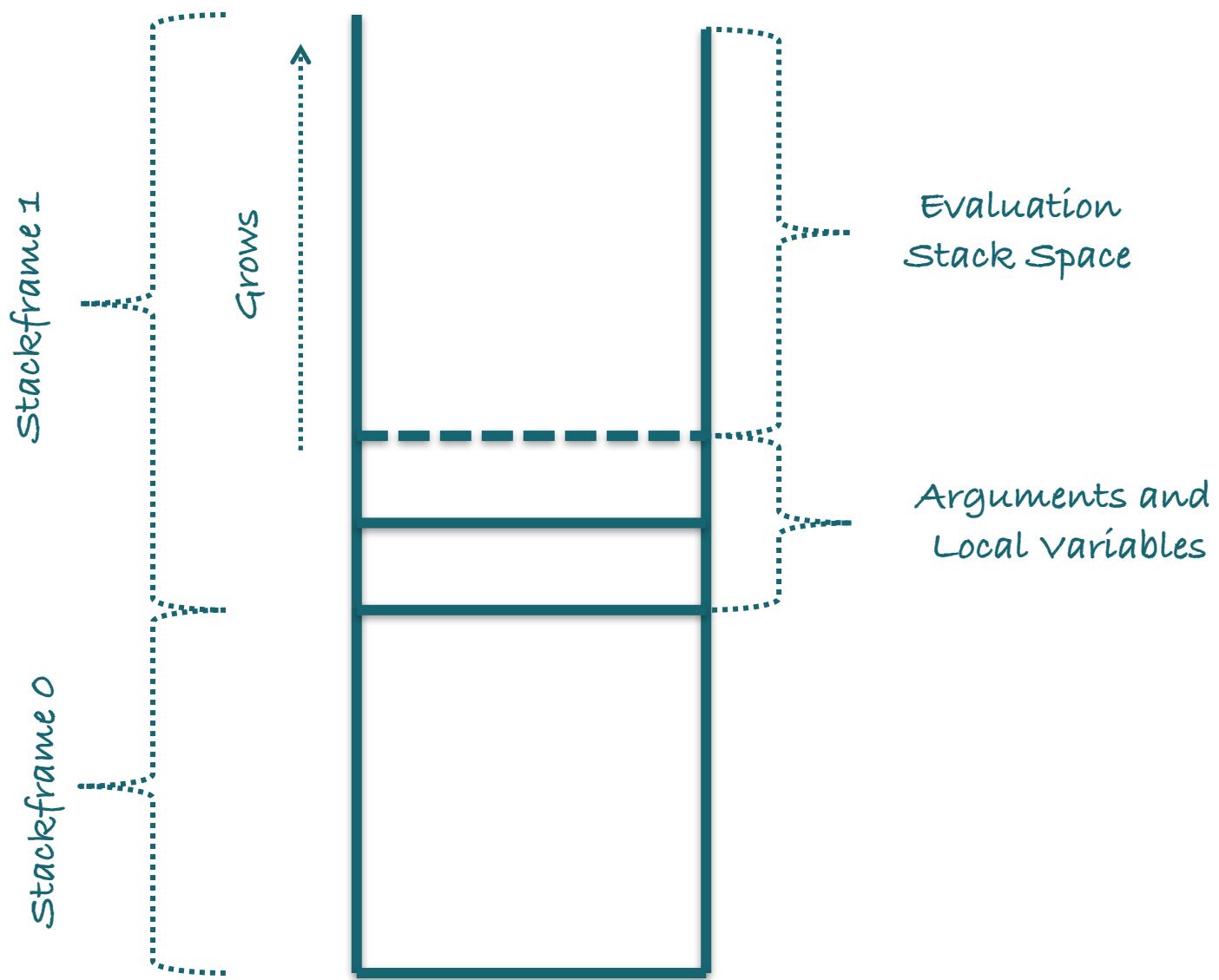
Стек вычислений (The Evaluation Stack)

Стек вычислений оценки является ключевой структурой приложений MSIL. Это мост между приложением и ячейками памяти. Он похож на обычный стековый фрейм, но есть значительные отличия.

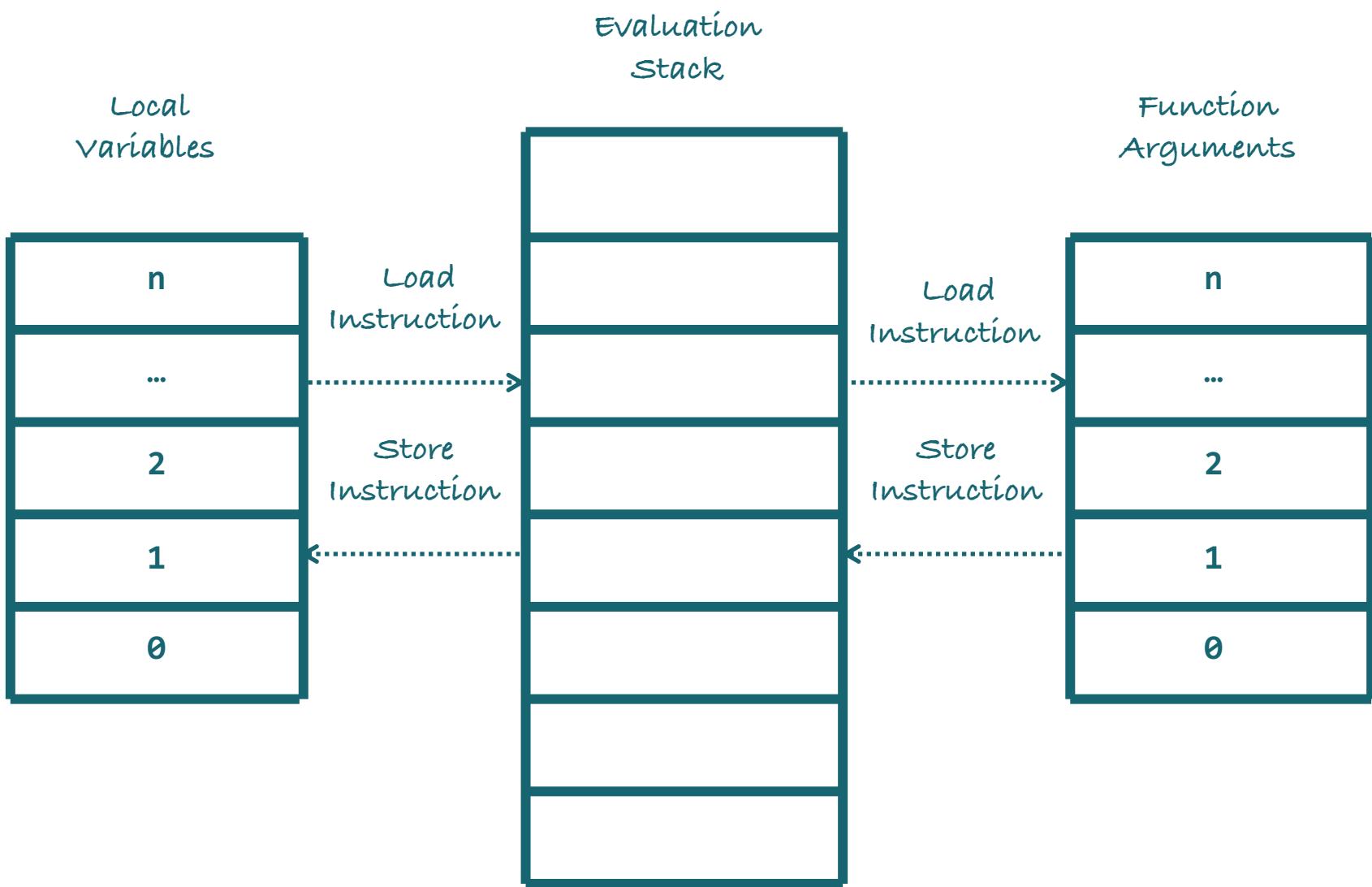
Стек вычислений - это средство просмотра приложения, и его можно использовать для просмотра параметров функции, локальных переменных, временных объектов и т. д. Традиционно параметры функций и локальные переменные помещаются в стек. В .NET эта информация хранится в отдельных репозиториях, в которых память зарезервирована для параметров функции и локальных переменных.

Нельзя напрямую обращаться к этим репозиториям. Доступ к параметрам или локальным переменным требует перемещения данных из памяти в слоты в стек вычислений с помощью команды `load`. И наоборот, при обновлении локальной переменной или параметра с содержимым в стек вычислений, используется команду `store`. Слоты в оценочном стеке составляют 4 или 8 байтов.

Стек вычислений



Стек вычислений



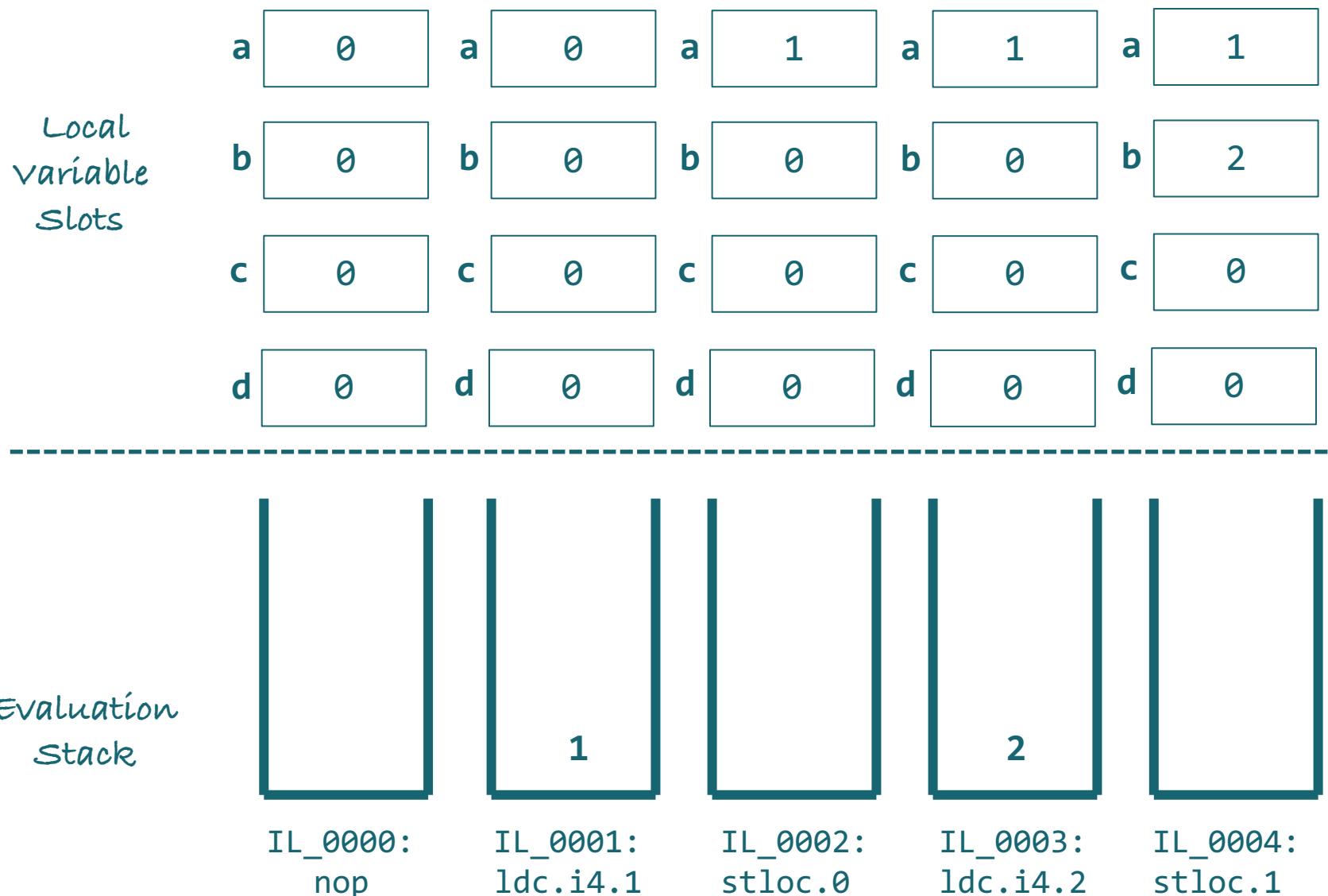
Стек вычислений

```
int a = 1;  
int b = 2;  
int c = 3;  
  
int d = a + b * c;
```

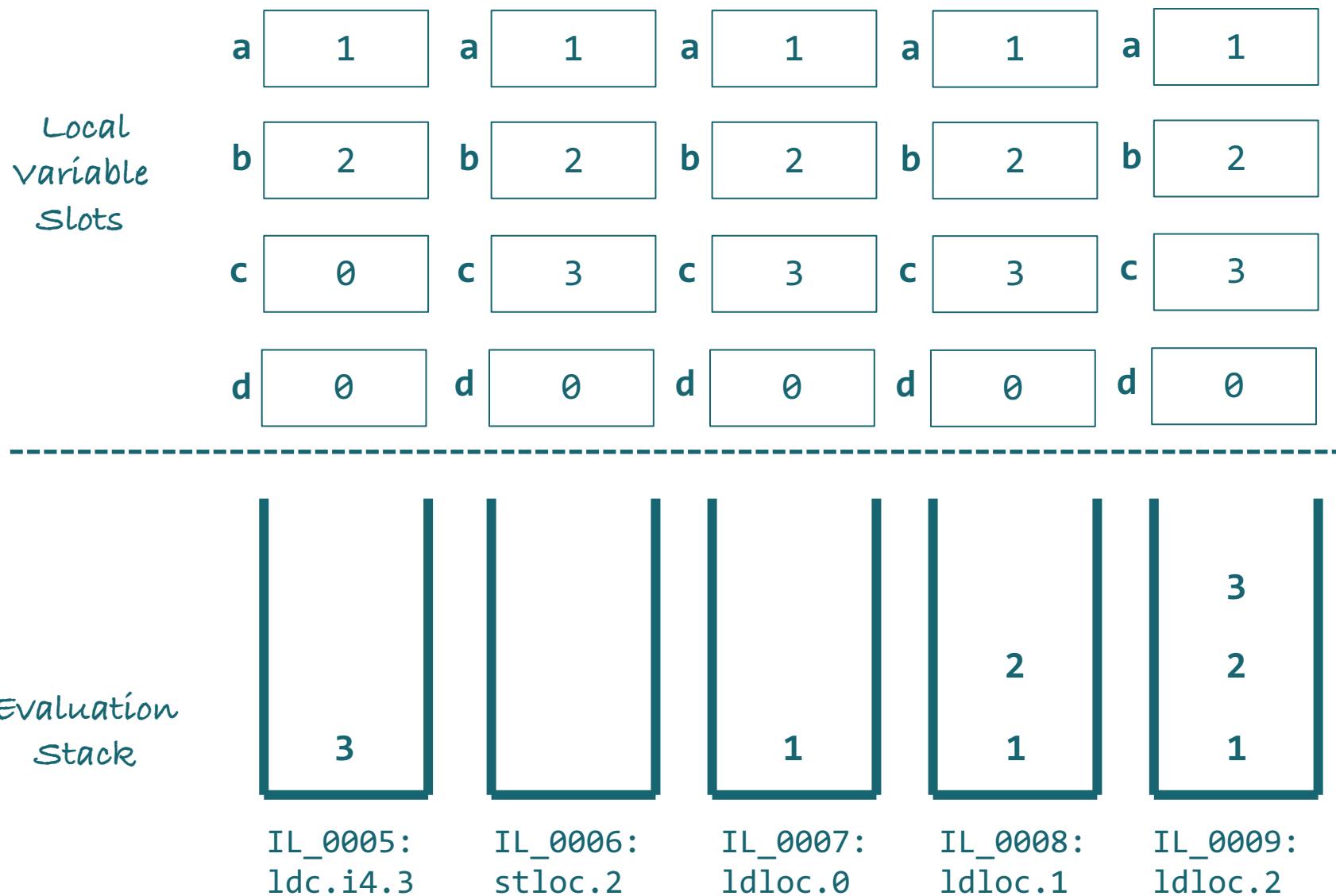
C# compiler

```
IL_0000: nop  
IL_0001: ldc.i4.1  
IL_0002: stloc.0      // a  
IL_0003: ldc.i4.2  
IL_0004: stloc.1      // b  
IL_0005: ldc.i4.3  
IL_0006: stloc.2      // c  
IL_0007: ldloc.0      // a  
IL_0008: ldloc.1      // b  
IL_0009: ldloc.2      // c  
IL_000A: mul  
IL_000B: add  
IL_000C: stloc.3      // d
```

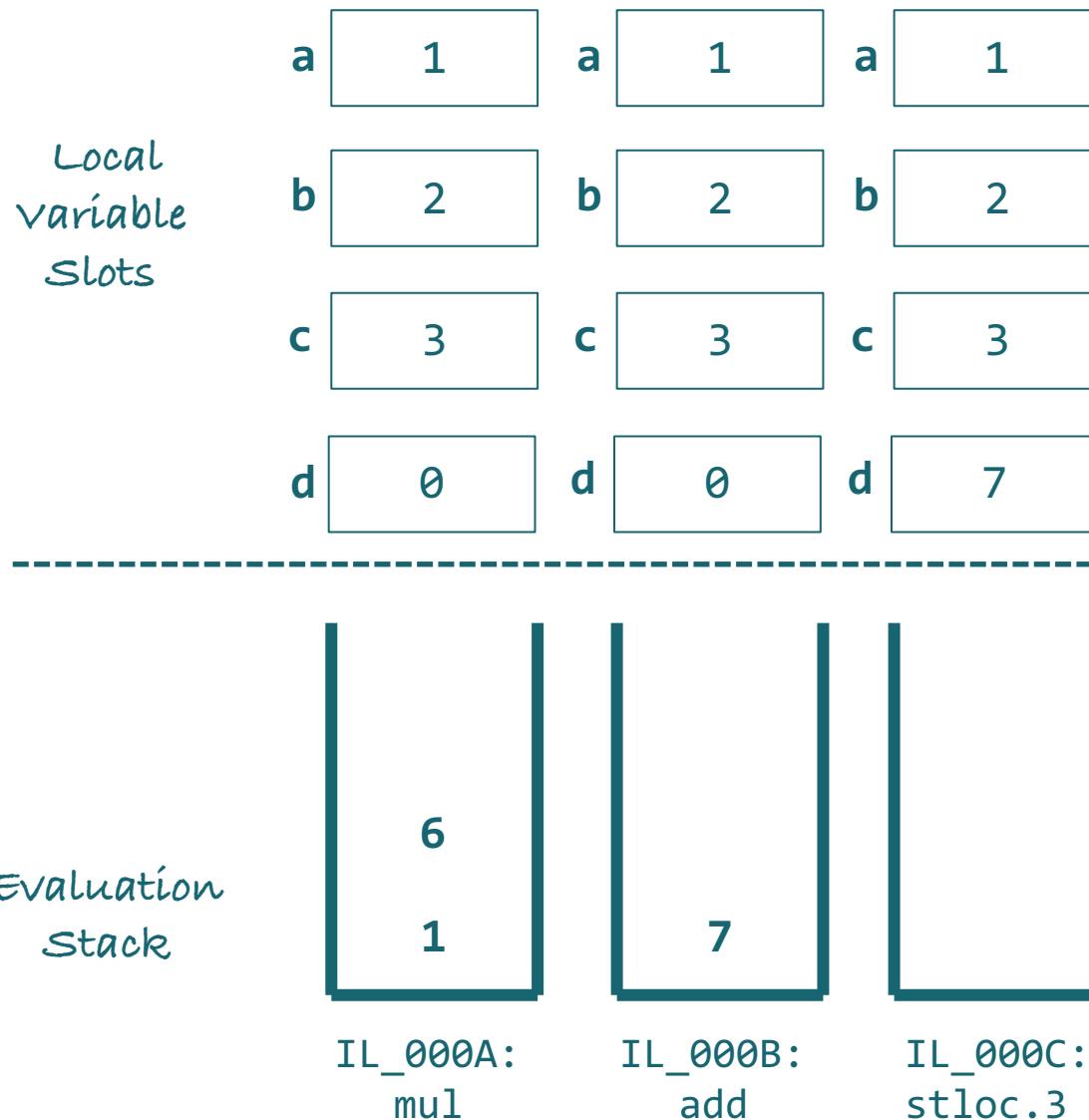
Стек вычислений



Стек вычислений



Стек вычислений



Стек вычислений

```
int x = 1;  
int y = x++;
```



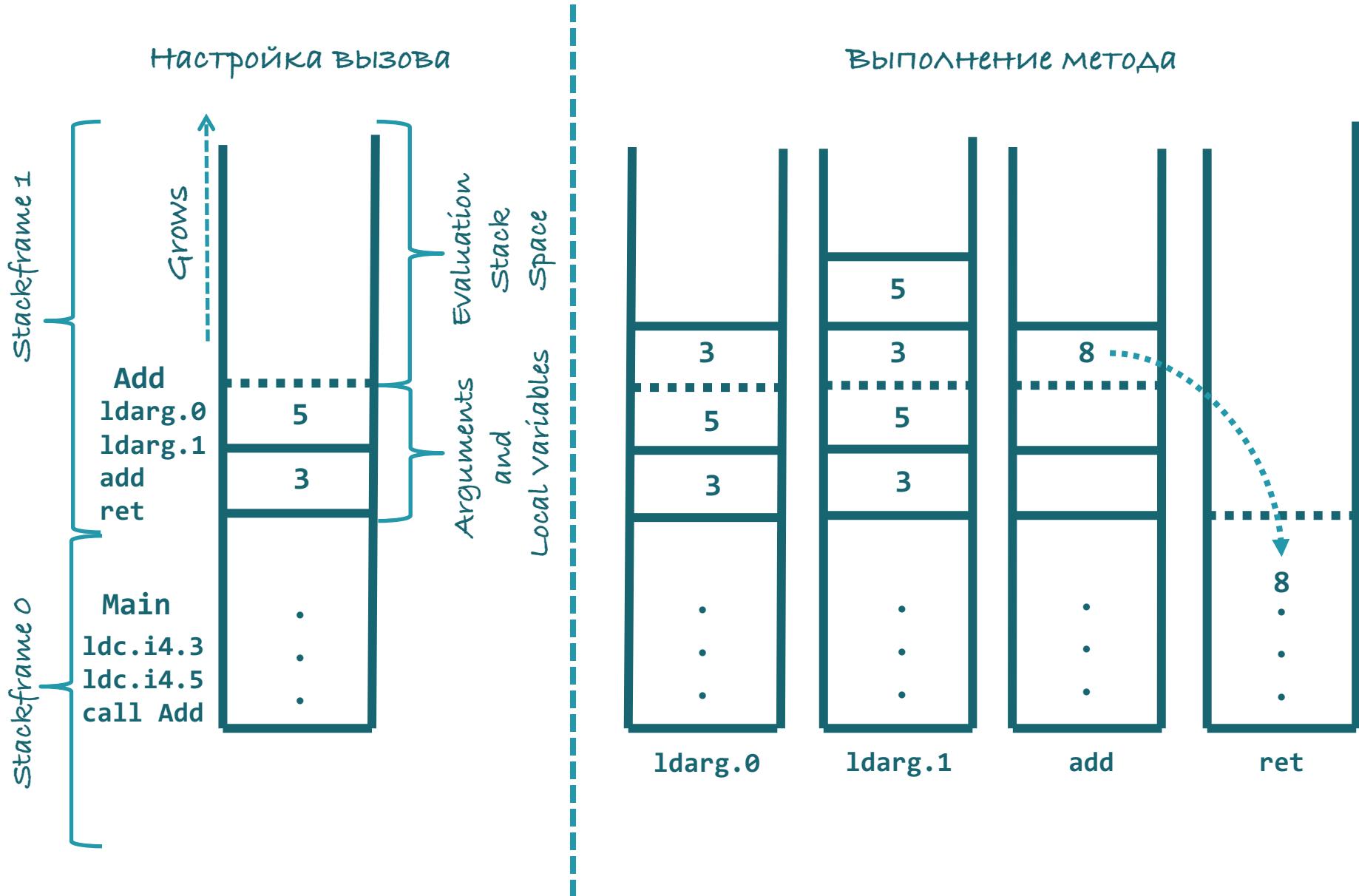
```
IL_0000: nop  
IL_0001: ldc.i4.1  
IL_0002: stloc.0  
IL_0003: ldloc.0  
IL_0004: dup  
IL_0005: ldc.i4.1  
IL_0006: add  
IL_0007: stloc.0  
IL_0008: stloc.1  
IL_0009: ret
```

```
int x = 1;  
int y = ++x;
```



```
IL_0000: nop  
IL_0001: ldc.i4.1  
IL_0002: stloc.0  
IL_0003: ldloc.0  
IL_0004: ldc.i4.1  
IL_0005: add  
IL_0006: dup  
IL_0007: stloc.0  
IL_0008: stloc.1  
IL_0009: ret
```

Стек вычислений



Управление потоком выполнения

Управление потоком выполнения

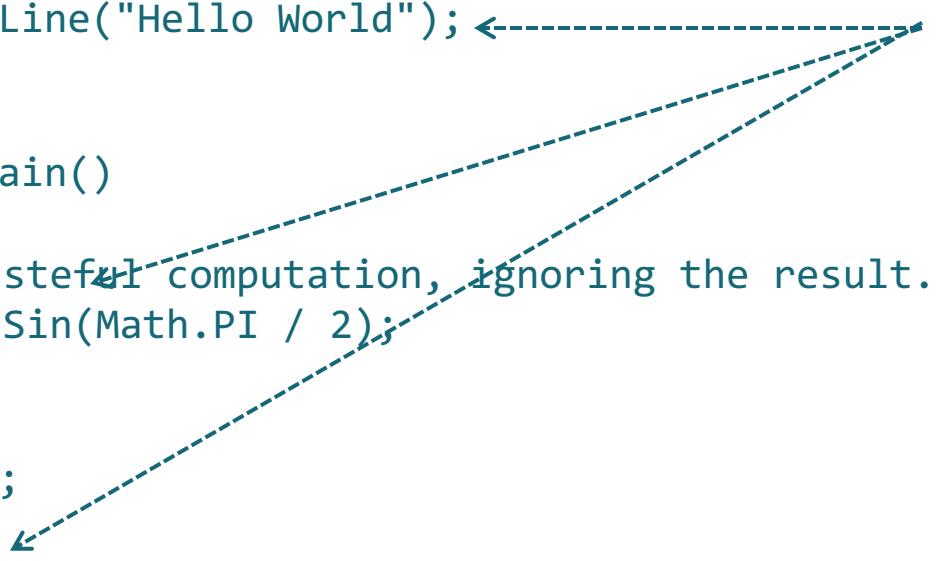
C# поддерживает множество утверждений, которые составляют таксономию следующим образом

- Expression statements
- Blocks
- Declaration statements
- Selection statements
- Iteration statements
- Jump statements
- Exception handling
- Resource management
- Locks
- Checked and unchecked contexts

Операторы выражения

Expression statements – позволяют подмножеству выражений, поддерживаемых языком, появляться самостоятельно, как правило, потому что они имеют полезные побочные эффекты, помимо создания значения. Включают различные формы присваивания, инкременты и декременты в постфиксной и префиксной форме, но также и вызовы методов.

```
Console.WriteLine("Hello World"); <----- Expression statements  
  
static void Main()  
{  
    // Wasteful computation, ignoring the result.  
    Math.Sin(Math.PI / 2);  
}  
  
name = "Bart";  
age = 25;  
age += 1;
```

A diagram illustrating the concept of expression statements. It consists of three parts of C# code. The first part is a single-line statement `Console.WriteLine("Hello World");`. A horizontal dashed arrow points from this line to the text `Expression statements` located to its right. The second part is a method body `static void Main(){}` containing a single-line comment `// Wasteful computation, ignoring the result.` followed by a call to `Math.Sin(Math.PI / 2);`. A diagonal dashed arrow originates from the word `computation` in the comment and points towards the same `Expression statements` text. The third part is a multi-line assignment block `name = "Bart"; age = 25; age += 1;`, which also has a diagonal dashed arrow originating from the word `assignment` in the first line and pointing towards the same `Expression statements` text.

Операторы блоки

Blocks - области кода, ограниченные фигурными скобками, - это способ группировать несколько операторов вместе, играют роль в определении переменных

```
if (user.Age >= 18)
{
    session.User = user;      ← Expression statements
    NavigateToHomepage(user.Homepage);
}
else
{
    LogInvalidAccessAttempt(); ← Expression statements
    ShowAccessDeniedPage();
}
```

Операторы объявления

Declaration statements - используются для объявления локальных переменных или констант путем сопоставления имени с идентификатором, тесно связаны с блоками из-за правил области видимости.

```
int age;           ← Expression statements  
var name; // This is invalid.  
var name = "Bart"; // Here we can infer System.String.  
const int triangleSideCount = 3; ←
```

Операторы выбора. Оператор if

Selection statements – предоставляют инструменты для ветвления потока выполнения на основе результата оценки выражения. Поток может быть переключен на основе результата или булева условия, которые могут использоваться для перехода в том или ином направлении

```
// Ignore when condition evaluates false.  
if (condition)  
    statement -if-true
```

← *one-way if*

```
// Also handle the false case.  
if (condition)  
    statement -if-true  
else  
    statement -if-false
```

← *either-or if*

Операторы выбора. Оператор if

```
if (n % 2 == 0)
{
    Console.WriteLine("Even");
}
else
{
    Console.WriteLine("Odd");
}
```

```
n % 2 == 0 ? "Even" : "Odd"
```

```
Type result = [condition] ? [true expression] : [false expression]
```

either-or if
form if-statement
vs
ternary statement

ternary statement

Операторы выбора. Оператор if

```
object foo = "Hello, world!";
```

```
if (foo is "Hello, world!") <-----  
{  
    Console.WriteLine("Hello, world!");  
}
```

```
if (foo is string s) <-----  
    Console.WriteLine(s);
```

Pattern matching C# 7.0

Операторы выбора. Оператор switch

```
switch ([expressionToCheck])
{
    case [test1]:
        ...
        [exitCaseStatement]
    case [test2]:
        ...
        [exitCaseStatement]
    default:
        ...
        [exitCaseStatement]
}
```

Каждый блок кода в операторе switch должен заканчиваться оператором, который явно завершает конструкцию ([exitCaseStatement]). Если опустить этот оператор, возникнет ошибка компиляции. В качестве таких операторов можно использовать:

- break;
- throw;
- goto case [testX];
- return;

Операторы выбора. Оператор switch

```
Control control = new TextBox();  
  
switch (control)  
{  
    case TextBox t:  
        Console.WriteLine(t.Multiline);  
        break;  
  
    case ComboBox c when c.DropDownStyle == ComboBoxStyle.DropDown:  
        Console.WriteLine(c.Items.Count);  
        break;  
  
    case ComboBox c:  
        Console.WriteLine(c.SelectedItem);  
        break;  
    default:  
        Console.WriteLine("Unknown");  
        break;  
  
    case null:  
        throw new ArgumentNullException(nameof(control));  
}
```

Pattern matching C# 7.0

Операторы циклов while и do

Iteration statements – обычно называются циклами; они выполняют содержащуюся инструкцию несколько раз на основе какого-либо условия или для выполнения заданного фрагмента кода для каждого элемента в последовательности данных.

```
while ([condition])
{
    [Code to loop]
}
```

```
do
{
    [Code to loop]
} while
```

```
while (!reader.EndOfStream)
{
    Console.WriteLine(reader.ReadLine());
}
```

```
char k;
do
{
    Console.WriteLine("Press x to exit");
    k = Console.ReadKey().KeyChar;
} while (k != 'x');
```

Операторы циклов. C-Style циклы for

```
for ([counterVariable = startingValue]; [condition]; [counterModification])
{
    [Code to loop]
}

for (int i = 0; i < 10; i++)
{
    // Code to loop, which can use i.
}
. . .
for (int i = 0; i < 10; i += 2)
{
    // Code to loop, which can use i.
}
. . .
int j;
for (j = 0; j < 10; j++)
{
    // Code to loop, which can use j.
}
// j is also available here
```

Операторы циклов. Итерирование по коллекции – цикл foreach

```
foreach (itemType iterationVariable in collection)
{
    [Code to loop]
}

string[] messages = GetMessagesFromSomewhere();
foreach (string message in messages)
{
    Console.WriteLine(message);
}
```

Операторы break, continue, goto

Jump statements – способ явно передать управление, что может означать различные вещи: можете перейти к следующей итерации цикла или полностью выйти из цикла, вернуться из метода, выбросить исключение и т. д.

```
int[] oldNumbers = { 1, 2, 3, 4, 5, 6, 7, 8 };
int count = 0;
while (oldNumbers.Length > count)
{
    if (oldNumbers[count] == 5)
    {
        break;
    }
    count++;
}
```

Операторы break, continue, goto

```
int[] oldNumbers = { 1, 2, 3, 4, 5, 6, 7, 8 };
int count = 0;
while (oldNumbers.Length > count)
{
    if (oldNumbers[count] == 5)
    {
        continue;
    }
    count++;
}
```

Обработка исключений

Exception handling – выполняется блочно-управляемым образом, когда область кода, называемая защищенным блоком, имеет связанные блоки обработчика исключений для типов исключений, которые она готова обрабатывать. Помимо обработки исключения, может быть указан блок, который выполняется независимо от результата защищенного блока (finally).

Управление ресурсами

Resource management – обеспечивает правильный подход к использованию ресурсов независимо от того, что происходит во время выполнения кода. Оператор using C# предоставляет средство для использования структурированного подхода при работе с ресурсами путем назначения ресурса блока, гарантируя, что ресурс очищается независимо от того, как элемент управления покидает этот блок.

Блокировка

Locks – способ координации выполнения параллельного кода. Выполнение операции над объектом с использованием блокировки (lock), гарантирует, что никакой другой код не сможет выполняться до тех пор, пока удерживается блокировка. Это позволяет структурировать выполнение операций, которые касаются объекта, предотвращая несогласованные состояния.

Проверяемый и непроверяемый контексты

Checked and unchecked contexts – проверяемый и непроверяемый контексты для управления арифметическим переполнением. Существуют в виде синтаксиса выражения и синтаксиса оператора на основе блока.

Массивы

Понятие массива

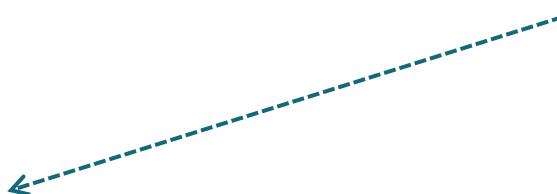
Массив представляет собой набор объектов, которые сгруппированы вместе и управляются как единое целое. Массивы имеют следующие характеристики:

- каждый элемент в массиве содержит значение
- индексируются с нуля
- нижняя граница массива индекс его первого элемента
- могут быть одномерными, многомерными или неправоугольные
- ранг массива это число измерений в массиве

Создание и инициализация массивов

```
int[] arrayName;  
. . .  
int[] list;  
list = new int[20];  
. . .  
int[] list = new int[20];  
. . .  
int[] list = new int[5] { 1, 2, 3, 4, 5 };  
int[] list = new int[] { 1, 2, 3, 4, 5 };  
int[] list = new[] { 1, 2, 3, 4, 5 };  
int[] list = { 1, 2, 3, 4, 5 };
```

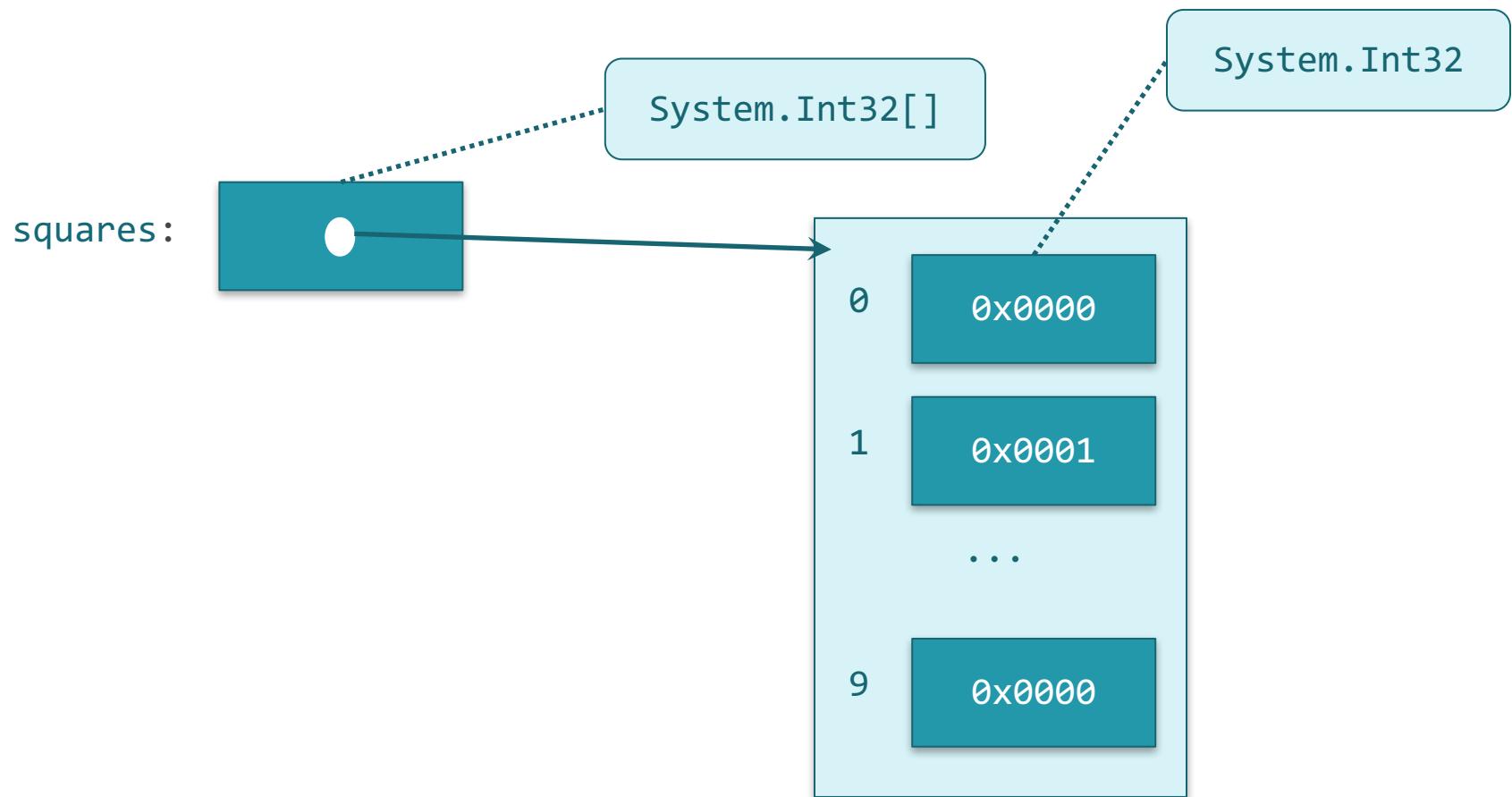
Одномерные (*single, sz*)
массивы



Если не инициализировать элементы массива, компилятор C# инициализирует их автоматически при его создании с помощью ключевого слова new значениями по умолчанию для его базового типа

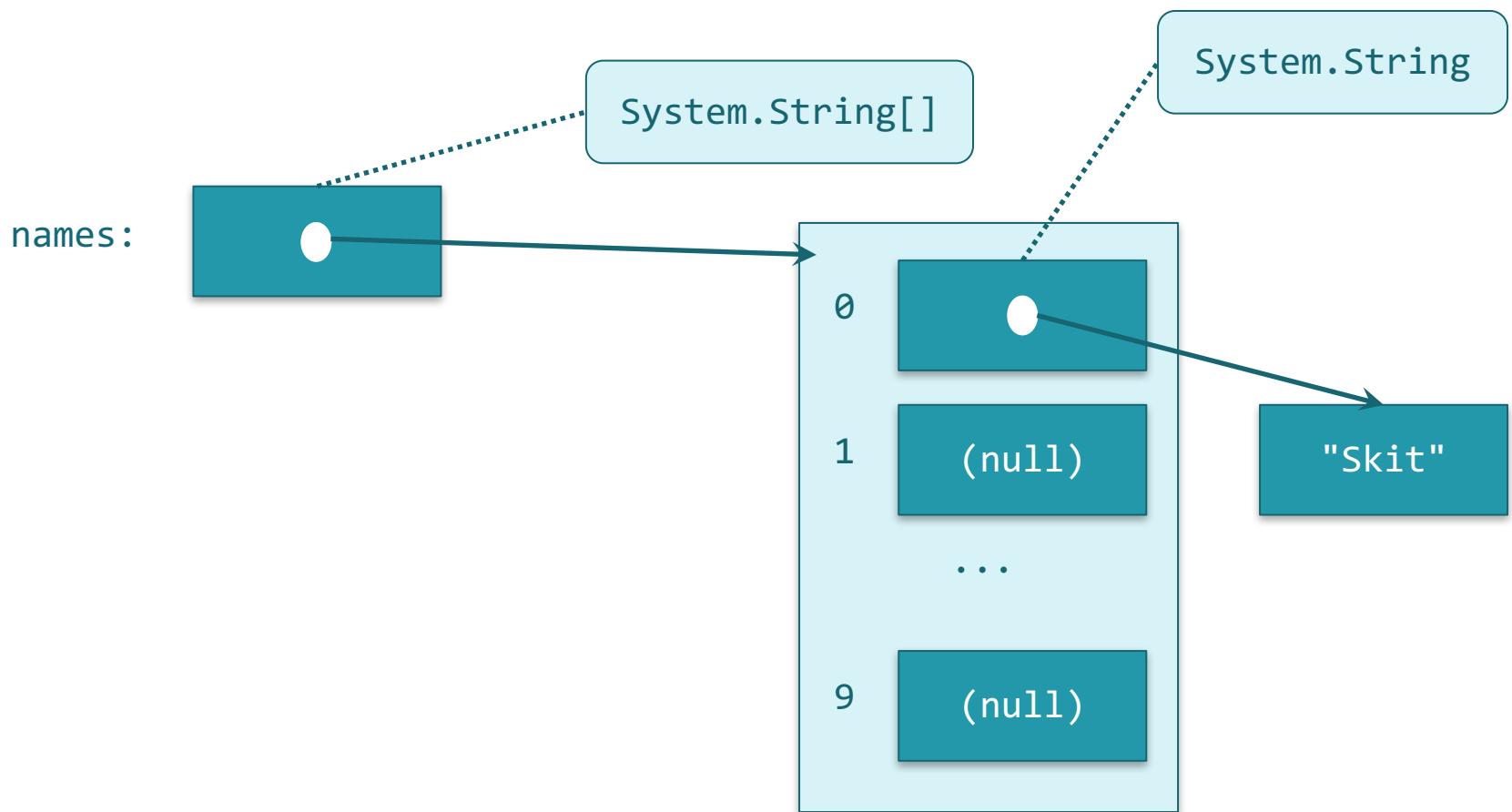
Одномерные массивы

```
int[] squares = new int[10];  
squares[1] = 1;
```



Одномерные массивы

```
string[] names = new string[10];
names[0] = "Skit";
```



Многомерные массивы

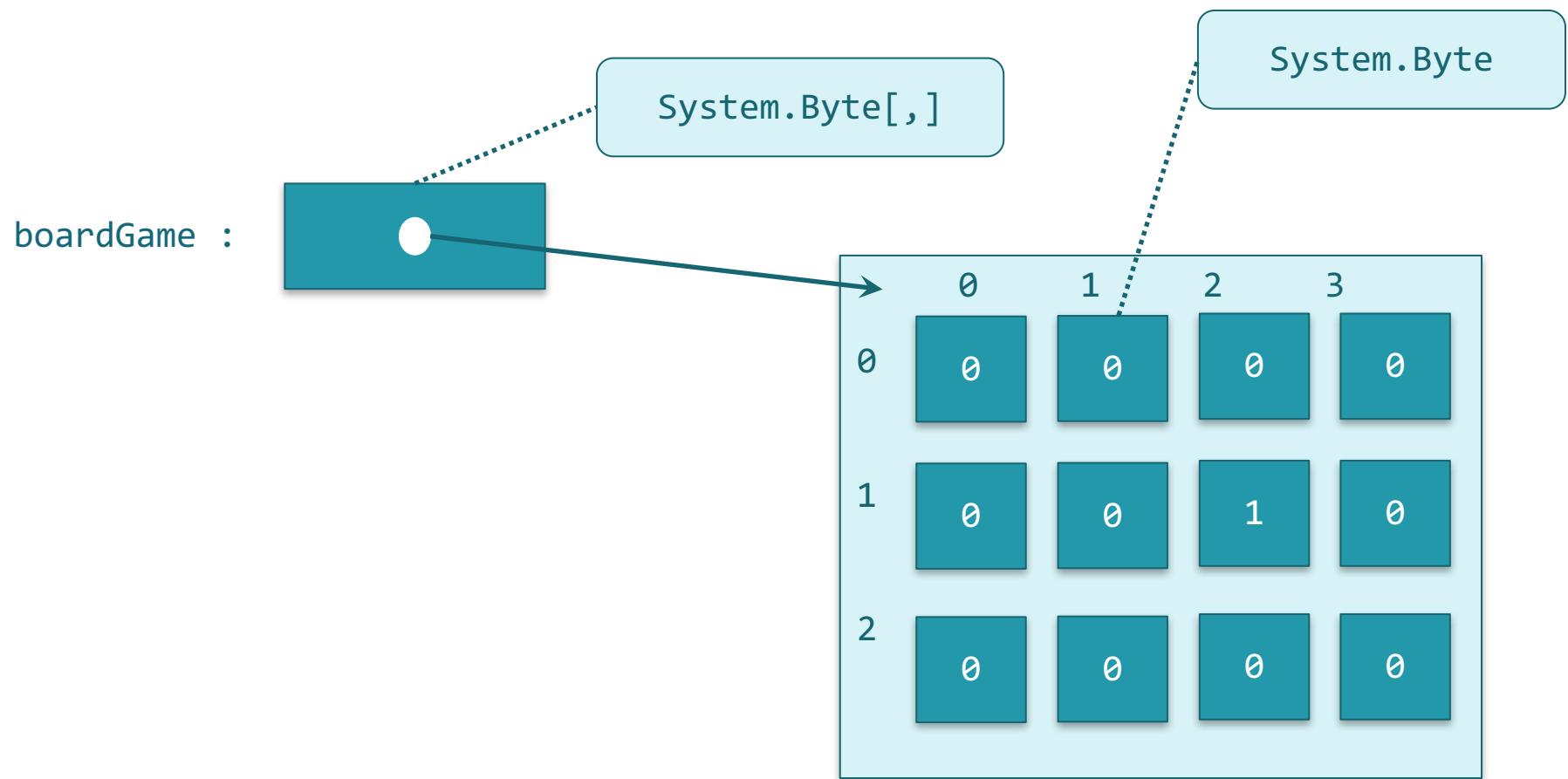
```
int[,] table; // two-dimensional array  
table = new int[10, 2];  
.  
.  
int[, ,] cube = new int[3, 2, 5]; // three-dimensional array
```

Многомерные (multiple)
массивы

```
Type[ , , ... ] arrayName = new Type[ Size1, Size2 , . . . ];
```

Многомерные массивы

```
byte[,] boardGame = new byte[3,4];  
boardGame[1,2] = 1;
```



Зубчатые массивы (jagged arrays)

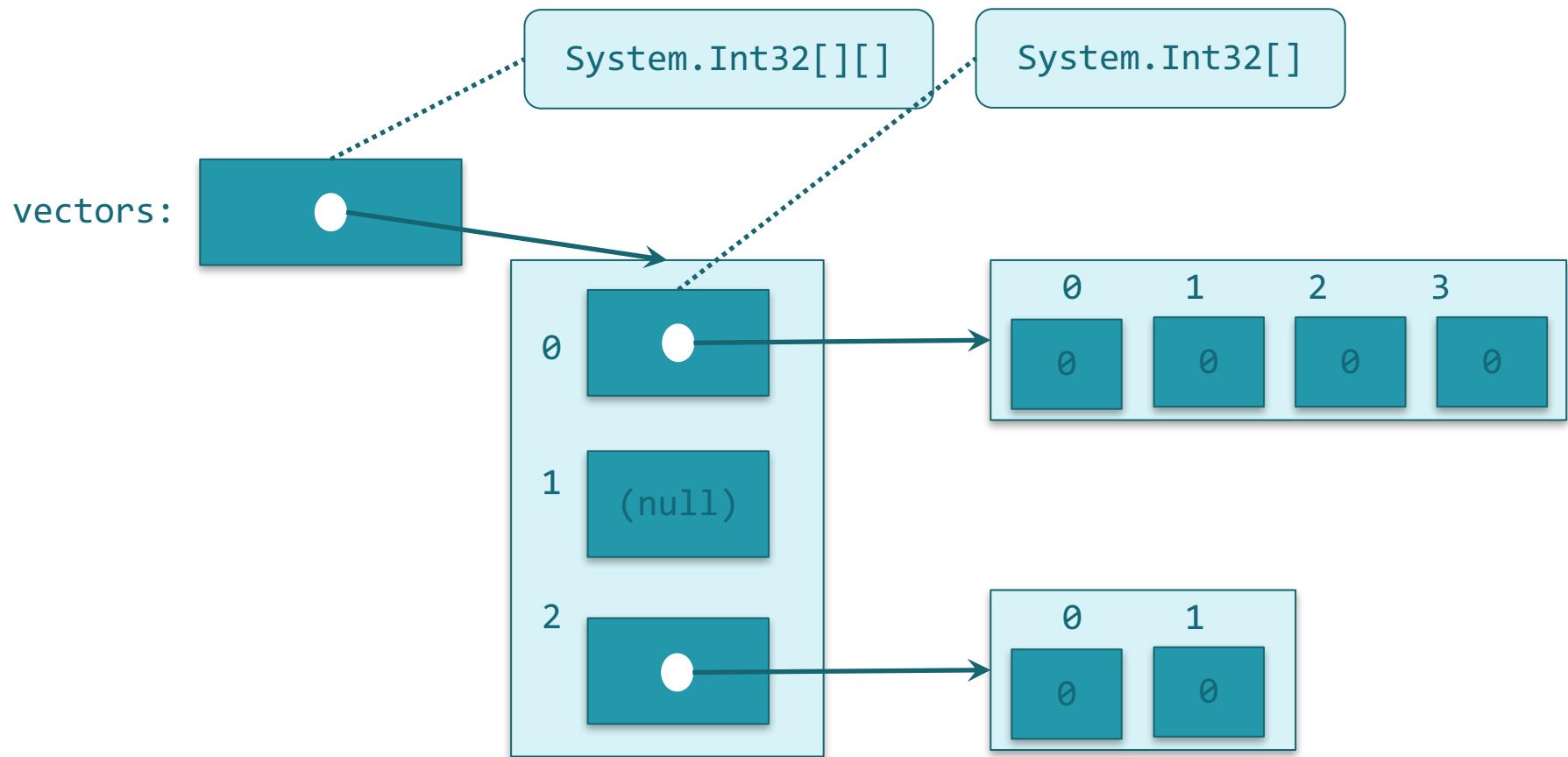
Jagged array

```
Type [][] jaggedArray = new Type[10][]; ←  
jaggedArray[0] = new Type[5]; // Can specify different sizes  
jaggedArray[1] = new Type[7];  
...  
jaggedArray[9] = new Type[21];
```

```
int[,] jaggedArray = new int[3][,]  
{  
    new int[,] {{1, 3}, {5, 7}},  
    new int[,] {{0, 2}, {4, 6}, {8, 10}},  
    new int[,] {{11, 22}, {99, 88}, {0, 9}}  
};
```

Зубчатые массивы

```
int[,] vectors = new int[3][];
vectors[0] = new int[4];
vectors[2] = new int[2];
```



Неявно типизированные массивы

```
var mixed = new[] { 1, DateTime.Now, true, false, 1.2 };

var a = new[] { 1, 10, 100, 1000 }; // int[]

var b = new[] { "hello", null, "world" }; // string[]

var d = new[]
{
    new[] {"Luca", "Mads", "Luke", "Dinesh"},
    new[] {"Karen", "Suma", "Frances"}
} // jagged array of strings

var c = new[]
{
    new[] {1, 2, 3, 4},
    new[] {5, 6, 7, 8}
};

};
```

Ковариантность массивов

```
string[] strings = new string[3]{ "one", "two", "three" };
object[] objects = new object[3];
objects = strings;
string[] stringsAgain = new string[3];
stringsAgain = (string[])objects;
```

Ковариантность

только для ссылочных
типов

```
int[] integers = new int[3] { 1, 2, 3 };
object[] objects = new object[3];
objects = integers;
```

СТЕ

```
int[] integers = new int[5];
object[] objects = new object[integers.Length];
Array.Copy(integers, objects, integers.Length);
```

Передача и возврат массивов

- Массив передается в метод всегда по ссылке, а метод может модифицировать элементы в массиве
- Отдельные методы могут возвращать ссылку на массив
- Если метод создает и инициализирует массив, возвращение ссылки на массив не вызывает проблем; если же нужно, чтобы метод возвращал ссылку на внутренний массив, ассоциированный с полем, сначала необходимо решить, вправе ли вызывающая программа иметь доступ к этому массиву

Символы, строки и работа с текстом

Символы

Метод	Действие
GetUnicodeCategory	Метод возвращает элементы перечисления UnicodeCategory, описывающего категорию символа
IsDigit	Возвращает true, если символ является десятичной цифрой
IsLetter	Возвращает true, если символ является буквой
IsPunctuation	Возвращает true, если символ является знаком препинания
IsSeparator	Возвращает true, если символ является разделителем
IsLower	Возвращает true, если символ – это буква в нижнем регистре
IsUpper	Возвращает true, если символ – это буква в верхнем регистре
ToLower	Приводит символ к нижнему регистру
ToUpper	Приводит символ к верхнему регистру
IsControl	Возвращает true, если символ является управляемым
IsLetterOrDigit	Возвращает true, если символ является буквой или цифрой
IsNumber	Возвращает true, если символ является десятичной или шестнадцатеричной цифрой

Символы

```
double d = Char.GetNumericValue('\u0033');
Console.WriteLine(d.ToString());//3
d = Char.GetNumericValue('\u00bc'); // '\u00bc' is the “vulgar fraction
one quarter ('¼'”)
Console.WriteLine(d.ToString());//0.25
d = Char.GetNumericValue('A'); // 'A' is the “Latin capital letter A”
Console.WriteLine(d.ToString());//-1
```

```
char c;
int n;
c = (char)65;
Console.WriteLine(c); //A
n = (int)c;
Console.WriteLine(n); //65
c = Convert.ToChar(65);
Console.WriteLine(c); //A
n = Convert.ToInt32(c);
Console.WriteLine(n); //65
```

Создание строки. Строковые литералы

```
string s = new string("Hi there.");
```

```
String s = new String("Hi there.");
```

```
string s = "Hi there.;"
```

```
String s = "Hi there.;"
```

```
string s = new string(' ', 20);
```

```
string s = new string(new char[]{ 'a', 'b', 'c', 'd', 'e' }));
```

```
string s = "Hi\r\nthere.;"
```

```
string s = "Hi" + Environment.NewLine + "there.;"
```

```
string file = "C:\\Windows\\System32\\Notepad.exe";
```

```
string file = @"C:\\Windows\\System32\\Notepad.exe";
```

verbatim strings

Строки. Члены класса

Член	Описание
Compare (статический метод)	Сравнение двух строк в лексикографическом (алфавитном) порядке. Разные реализации метода позволяют сравнивать строки с учетом, или без учета регистра
CompareTo (экземплярный метод)	Сравнение текущего экземпляра строки с другой строкой
Concat (статический метод)	Слияние произвольного числа строк
Copy (статический метод)	Создание копии строки
Empty (статическое поле)	Открытое статическое поле, представляющее пустую строку
Format (статический метод)	Форматирование строки в соответствии с заданным форматом

Строки. Члены класса

Член	Описание
IndexOf, LastIndexOf (экземплярные методы)	Определение индекса первого или, соответственно, последнего вхождения подстроки в данной строке
IndexOfAny, LastIndexOfAny (экземплярные методы)	Определение индекса первого или, соответственно, последнего вхождения любого символа из подстроки в данной строке
Insert (экземплярный методы)	Вставка подстроки в заданную позицию
Join (статический метод)	Слияние массива строк в единую строку. Между элементами массива вставляются разделители
Length (свойство)	Возвращает длину строки

Строки. Члены класса

Член	Описание
PadLeft, PadRigth (экземплярные методы)	Выравнивают строки по левому или, соответственно, правому краю путем вставки нужного числа пробелов в начале, или в конце строки
Remove (экземплярный метод)	Удаление подстроки из заданной позиции
Replace (экземплярный метод)	Замена всех вхождений заданной подстроки, или символа новыми подстрокой, или символом
Split (экземплярный метод)	Разделяет строку на элементы, используя разные разделители. Результаты помещаются в массив строк
EndWith EndWith (экземплярные методы)	Возвращают true или false в зависимости от того, начинается, или заканчивается строка заданной подстрокой

Строки. Члены класса

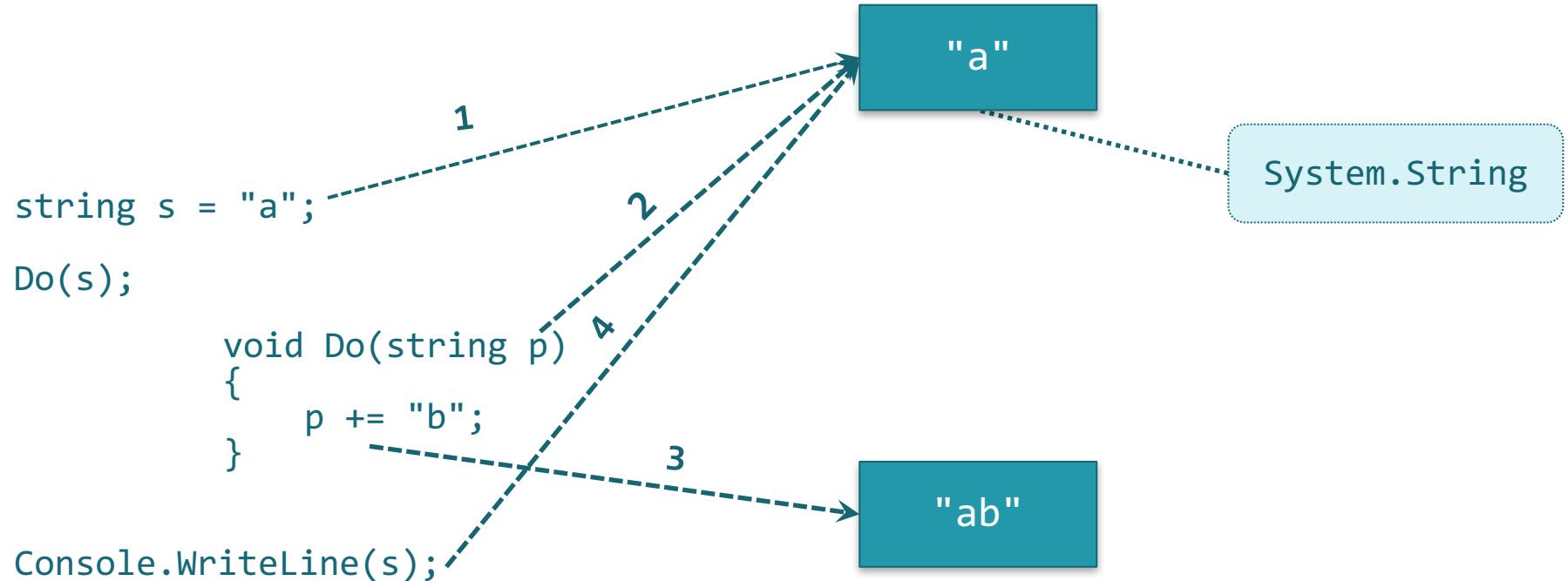
Член	Описание
Substring (экземплярный метод)	Выделение подстроки, начиная с заданной позиции
ToCharArray (экземплярный метод)	Преобразует строку в массив символов
ToLower, ToUpper (экземплярные методы)	Преобразование строки к нижнему или, соответственно, к верхнему регистру
Trim, TrimStart, TrimEnd (экземплярные методы)	Удаление пробелов в начале и конце строки, или только с начала, или только с конца соответственно

Строка как неизменяемый (immutable) тип

```
string s = "Это текстовая строка, состоящая из семи слов.";  
  
Console.WriteLine($"Длина строки {s.Length}");  
Console.WriteLine($"Наличие подстроки \"текст\" {s.Contains("текст")});  
Console.WriteLine($"Вставка \"{s.Insert(4, "большая")}\");  
Console.WriteLine($"Длина строки {s.Length} символов");  
Console.WriteLine($"Удаление:{s.Remove(4, 10)}");  
Console.WriteLine($"Замена: \"{s.Replace("семи", "нескольких")}\");  
Console.WriteLine($"Подстрока: \"{s.Substring(4, 5)}");  
Console.WriteLine($"В верхний регистр: {s.ToUpper()});  
Console.WriteLine($"Вхождение \"текст\": {s.IndexOf("текст")});
```

Тип `string` является неизменяемым (immutable) типом данных, таким образом, методы и операции, модифицирующие содержимое строк, на самом деле создают новые строки, копируя при необходимости содержимое старых

Строка как неизменяемый тип



Форматные строки

Для того, чтобы разрабатываемые классы были дружественными к пользователю, они должны предлагать средства для отображения своих строковых представлений в любом из форматов, которые могут понадобиться пользователю

В исполняющей среде .NET определен стандартный способ достижения этого — интерфейс `IFormattable`

```
decimal d = 12.05667M;  
int i = 5;  
string message = string.Format("Value d = {0,8:C}, and i = {1}", d, i);
```

Количество символов, занимаемое представлением элемента, снабженное префиксом-запятой

индекс элемента

спецификатор формата предваряется двоеточием

Интерполяционные строки (C# 6.0)

```
string name = "Chuck";
string surname = "Norris";
string message = string.Format("The man is {0} {1}", name, surname);
string messageAgain = $"The man is {name} {surname}";
Console.WriteLine(message);
Console.WriteLine(messageAgain);
```

← Strings interpolation

```
var numbers = new int[] { 1, 2, 3, 4, 5 };
Console.WriteLine($"There are {numbers.Length} numbers, and their average
is {numbers.Average()}");
```

← ←

```
var dateOfBirth = DateTime.Now;
Console.WriteLine($"It's {dateOfBirth:yyyy-MM-dd}");
```

Спецификаторы формата

Спецификатор	Применяется к	Значение	Пример
C	Числовым типам	Символ местной валюты	\$835.50 (США) £835.50 (Великобритания) 835.50р. (Россия)
D	Только к целочисленным типам	Обычное целое	835
E	Числовым типам	Экспоненциальная нотация	8.35E+002
F	Числовым типам	С фиксированной десятичной точкой	835.50
G	Числовым типам	Обычные числа	835.5
N	Числовым типам	Формат чисел, принятый в данной местности	4,384.50 (Великобритания/США) 4 384,50 (континентальная Европа)
P	Числовым типам	Процентная нотация	835,000.00%
X	Только к целочисленным типам	Шестнадцатеричный формат	1a1f

Спецификаторы формата

```
CultureInfo ci = new CultureInfo("en-us");
double floating = 10761.937554;
Console.WriteLine("C: {0}", floating.ToString("C", ci));
Console.WriteLine("E: {0}", floating.ToString("E03", ci));
Console.WriteLine("F: {0}", floating.ToString("F04", ci));
Console.WriteLine("G: {0}", floating.ToString("G", ci));
Console.WriteLine("N: {0}", floating.ToString("N03", ci));
Console.WriteLine("P: {0},(floating / 10000).ToString("P02",
ci));
Console.WriteLine("R: {0}", floating.ToString("R", ci));
Console.WriteLine();
```

C: \$10,761.94
E: 1.076E+004
F: 10761.9376
G: 10761.937554
N: 10,761.938
P: 107.62 %
R: 10761.937554


```
CultureInfo ci = new CultureInfo("ru-ru");
double floating = 10761.937554;
Console.WriteLine("C: {0}", floating.ToString("C", ci));
Console.WriteLine("E: {0}", floating.ToString("E03", ci));
Console.WriteLine("F: {0}", floating.ToString("F04", ci));
Console.WriteLine("G: {0}", floating.ToString("G", ci));
Console.WriteLine("N: {0}", floating.ToString("N03", ci));
Console.WriteLine("P: {0},(floating / 10000).ToString("P02",
ci));
Console.WriteLine("R: {0}", floating.ToString("R", ci));
Console.WriteLine();
```

C: 10 761,94p.
E: 1,076E+004
F: 10761,9376
G: 10761,937554
N: 10 761,938
P: 107,62%
R: 10761,937554

Строковый тип **StringBuilder**

Для операций со строками и символами в библиотеке классов .NET Framework существует тип **TextStringBuilder** из пространства имен **System.Text**

```
StringBuilder sb = new StringBuilder();
```

Строковый тип `StringBuilder`

```
public sealed class StringBuilder : ISerializable // .NET 2.0
{
    // Fields
    private const string CapacityField = "Capacity";
    internal const int DefaultCapacity = 0x10;
    internal IntPtr m_currentThread;
    internal int m_MaxCapacity;
    internal volatile string m_StringValue;
    private const string MaxCapacityField = "m_MaxCapacity";
    private const string StringValueField = "m_StringValue";
    private const string ThreadIDField = "m_currentThread";
    . . .
}
```

Строковый тип `StringBuilder`

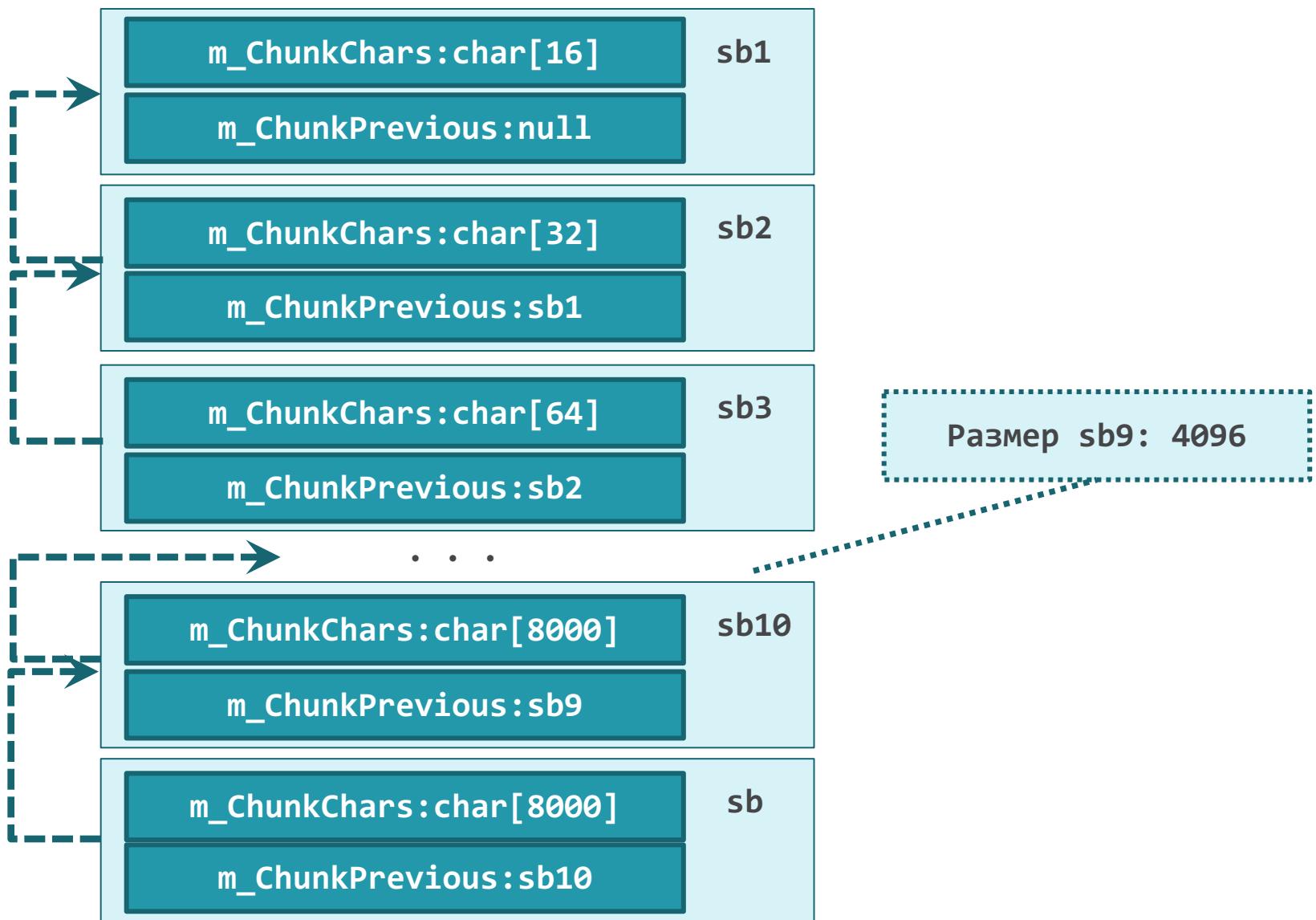
```
public sealed class StringBuilder : ISerializable // .NET 4.0
{
    // Fields
    private const string CapacityField = "Capacity";
    internal const int DefaultCapacity = 0x10;
    internal char[] m_ChunkChars;
    internal int m_ChunkLength;
    internal int m_ChunkOffset;
    internal StringBuilder m_ChunkPrevious;
    internal int m_MaxCapacity;
    private const string MaxCapacityField = "m_MaxCapacity";
    internal const int MaxChunkSize = 0x1f40; //8000
    private const string StringValueField = "m_StringValue";
    private const string ThreadIDField = "m_currentThread";
    . . .
}
```

Строковый тип `StringBuilder`

```
static void Main(string[] args)
{
    var sb = new StringBuilder();
    Append(sb, 20000);
    sb.Clear();
}

private static void Append(StringBuilder sb, int numberOfChars)
{
    for (int i = 0; i < numberOfChars; i++)
    {
        sb.Append("F");
    }
}
```

Строковый тип StringBuilder



Строковый тип **StringBuilder**

Член	Описание
MaxCapacity (свойство)	Возвращает наибольшее количество символов
Capacity (свойство)	Получает/устанавливает размер массива символов
EnsureCapacity (метод)	Гарантирует, что размер массива символов будет не меньше, чем значение параметра, передаваемого этому методу
Length (свойство)	Возвращает количество символов в строке
ToString (метод)	Версия без параметров возвращает объект String, представляющий поле с массивом символов объекта StringBuilder
AppendFormat (метод)	Добавляет заданные объекты в массив символов, увеличивая его при необходимости

Строковый тип `StringBuilder`

Член	Описание
<code>Replace</code> (метод)	Заменяет один символ или строку символов в массиве символов
<code>Remove</code> (метод)	Удаляет диапазон символов из массива символов
<code>Equals</code> (метод)	Возвращает <code>true</code> , только если объекты <code>StringBuilder</code> имеют одну и ту же максимальную емкость, емкость и одинаковые символы в массиве
<code>CopyTo</code> (метод)	Копирует подмножество символов <code>StringBuilder</code> в массив <code>Char</code>

Регулярные выражения

Для упрощения решения задач по обработке символьной информации в пространстве имен `System.Text.RegularExpressions` определены классы для работы со строками, основанные на регулярных выражениях, `Regex`, `Match` и `MatchCollection`

Регулярное выражение – это шаблон, согласно которому выполняется поиск соответствующего фрагмента текста

Использование регулярных выражений обеспечивает:

- проверку строки на соответствие шаблону
- поиск в тексте по заданному шаблону
- разбиение текста на фрагменты

Регулярные выражения

Набор символов	Описание	Пример
.	Любой символ, кроме \n	Выражение c.t соответствует фрагментам: cat, cut, c#t, c{t и т.д.
[]	Любой одиночный символ из записанной внутри скобок. Допускается использование диапазонов символов, для задания которого используется символ «-»	Выражение c[au]t соответствует фрагментам: cat, cut, cit. Выражение c[a-c]t соответствует фрагментам: cat, cbt, cct
[^]	Любой одиночный символ, не входящий в последовательность, записанную внутри скобок. Допускается использование диапазонов символов	Выражение c[^au]t соответствует фрагментам: cbt, cct, c2t и т.д. Выражение c[^a-c]t соответствует фрагментам: cdt, cet, c%t и т.д.

Регулярные выражения

Набор символов	Описание	Пример
\w	Любой алфавитно-цифровой символ, а также символ подчеркивания	Выражение <code>c\wt</code> соответствует фрагментам: <code>cbt</code> , <code>cct</code> , <code>c2t</code> , <code>c_t</code> и т. д., но не соответствует фрагментам <code>c%t</code> , <code>c{t</code> и т. д.
\W	Любой символ не удовлетворяющий \w	Выражение <code>c\Wt</code> соответствует фрагментам: <code>c%t</code> , <code>c{t</code> , <code>c.t</code> и т.д., но не соответствует фрагментам <code>cbt</code> , <code>cct</code> , <code>c2t</code> и т.д.
\s	Любой пробельный символ (пробел, табуляция и переход на новую строку)	Выражение <code>\s\w\w\w\s</code> соответствует любому слову из трех букв, окруженному пробельными символами

Регулярные выражения

Набор символов	Описание	Пример
\S	Любой не пробельный символ	Выражение <code>\s\S\S\s</code> соответствует любым трем непробельным символам, окруженным пробельными
\b	Граница слова	Выражение <code>\B\d\d\d\B</code> соответствует любым трем цифрам, входящим в состав слова так, что ни справа ни слева от них нет конца слова
\d	Любая десятичная цифра	Выражение <code>c\dt</code> соответствует фрагментам: c1t, c2t, c3t и т.д.

Регулярные выражения

Повторители	Описание	Пример
*	Ноль, или более повторений предыдущего элемента	Выражение ca^*t соответствует фрагментам: ct, cat, caat, caaat и т.д.
+	Одно, или более повторений предыдущего элемента	Выражение $ca+t$ соответствует фрагментам: cat, caat, caaat и т.д.
?	Не более одного повторения предыдущего элемента	Выражение $ca?t$ соответствует фрагментам: ct, cat
{n}	Ровно n повторений предыдущего элемента	Выражение $ca\{3\}t$ соответствует фрагменту: caaat. Выражение $(cat)\{2\}$ соответствует фрагменту: catcat

Регулярные выражения

Повторители	Описание	Пример
{n,}	По крайней мере n повторений предыдущего элемента	Выражение $ca\{3,\}t$ соответствует фрагментам: caaat, caaaat, caaaaaat и т.д. Выражение $(cat)\{2,\}$ соответствует фрагментам: catcat, catcatcat и т.д.
{n, m}	От n до m повторений предыдущего элемента	Выражение $ca\{2,4\}t$ соответствует фрагментам: saat, caaat, caaaaat

Регулярные выражения

- Слово USA – @ " USA"
- Номер телефона в формате xxx-xx-xx – @"\d\d\d-\d\d-\d\d" или @"\d{3}(-\d\d){2}"
- Целое число со знаком, или без знака – @"[+-]?\d+"
- Время в формате чч.мм, или чч:мм – @"([01]\d)|(2[0-4])[.:][0-5]\d"

Регулярные выражения

Редактор [Expresso](#) подходит в качестве учебного пособия для начинающего пользователя регулярных выражений, а также является полнофункциональной средой разработки для опытного программиста или веб-дизайнер с обширными знаниями о регулярных выражениях

Регулярные выражения

Класс Regex

- IsMatch
- Match
- Matches
- Replace
- Split

```
Regex r = new Regex("собака", RegexOptions.IgnoreCase);
string text1 = "Кот в доме, собака в конуре.";
string text2 = "Котик в доме, собачка в конуре.";
Console.WriteLine(r.IsMatch(text1)); //True
Console.WriteLine(r.IsMatch(text2)); //False
```

```
Regex r = new Regex(@"\d{2,3}(-\d\d){2}");
string text1 = "tel:123-45-67";
string text2 = "tel:no";
string text3 = "tel:12-34-56";
Console.WriteLine(r.IsMatch(text1));//True
Console.WriteLine(r.IsMatch(text2));//False
Console.WriteLine(r.IsMatch(text3));//True
```

Регулярные выражения

Класс Match

- Success
- Length
- Index
- NextMatch

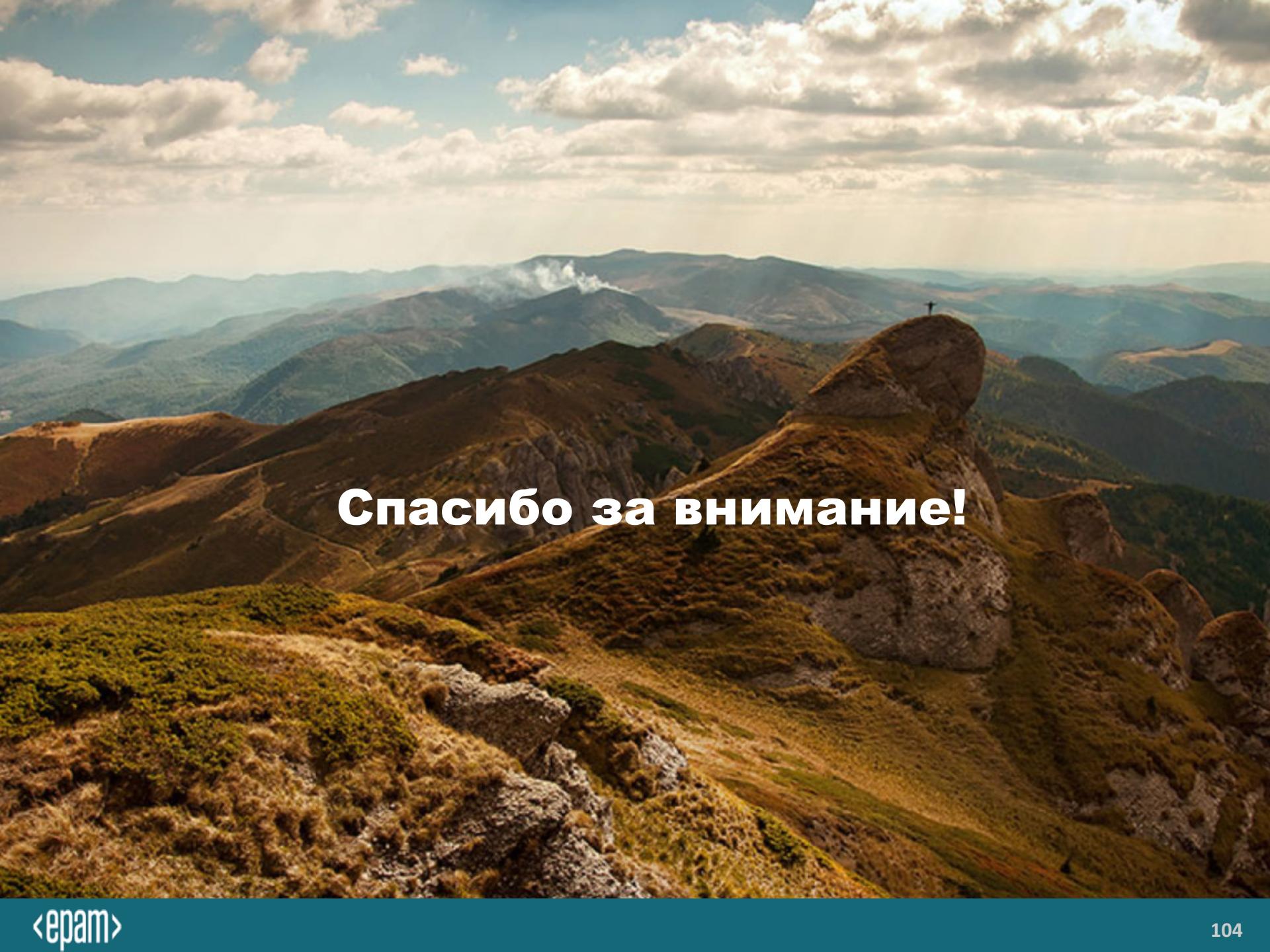
```
Regex r = new Regex(@"[-+]?[0-9]+");
string text = @"5*10=50 -80/40=-2";
Match teg = r.Match(text);
int sum = 0;
while (teg.Success)
{
    Console.WriteLine("{0} ", teg.Value);
    sum += int.Parse(teg.ToString());
    teg = teg.NextMatch();
}
Console.WriteLine("\nsum={0}", sum);
```

Регулярные выражения

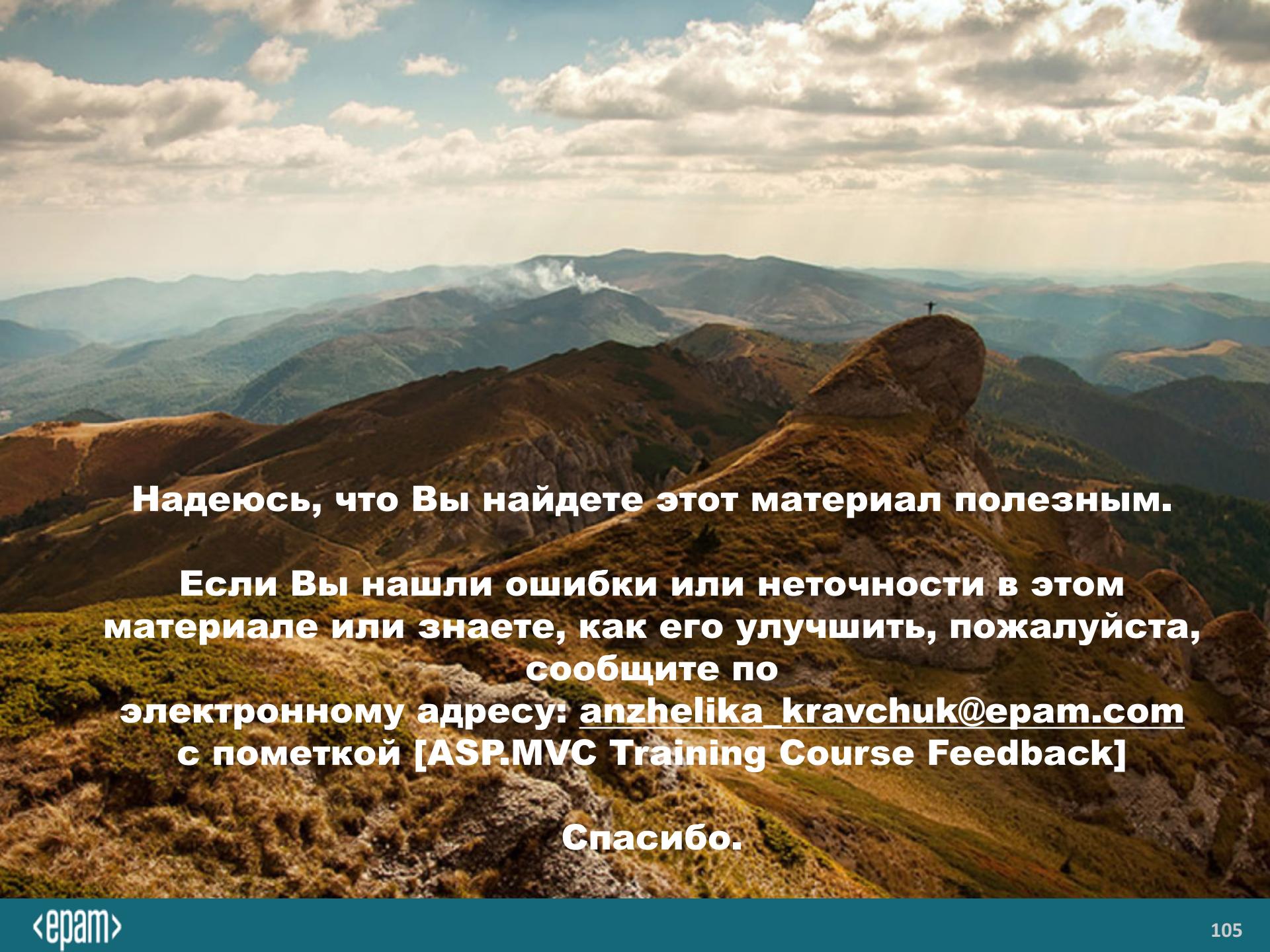
Класс MatchCollection

Метод Matches класса Regex возвращает объект класса MatchCollection – коллекцию только для чтения всех фрагментов заданной строки, совпавших с шаблоном

```
Regex r = new Regex(@"[-+]?[0-9]+");
string text = @"5*10=50 -80/40=-2";
MatchCollection teg = r.Matches(text);
int sum = 0;
foreach (Match temp in teg)
{
    sum += int.Parse(temp.ToString());
}
Console.WriteLine("\nsum={0}", sum);
```

A wide-angle photograph of a mountain range under a dramatic sky. In the foreground, rocky terrain and green slopes are visible. A lone figure stands on a prominent peak in the middle ground. The background features multiple layers of mountains, with a plume of white smoke or steam rising from one of the peaks in the distance.

Спасибо за внимание!



Надеюсь, что Вы найдете этот материал полезным.

**Если Вы нашли ошибки или неточности в этом
материале или знаете, как его улучшить, пожалуйста,
сообщите по**

**электронному адресу: anzhelika_kravchuk@epam.com
с пометкой [ASP.MVC Training Course Feedback]**

Спасибо.