



# BASIC CODING IN C# 7.0

.NET & JS LAB. 2017

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

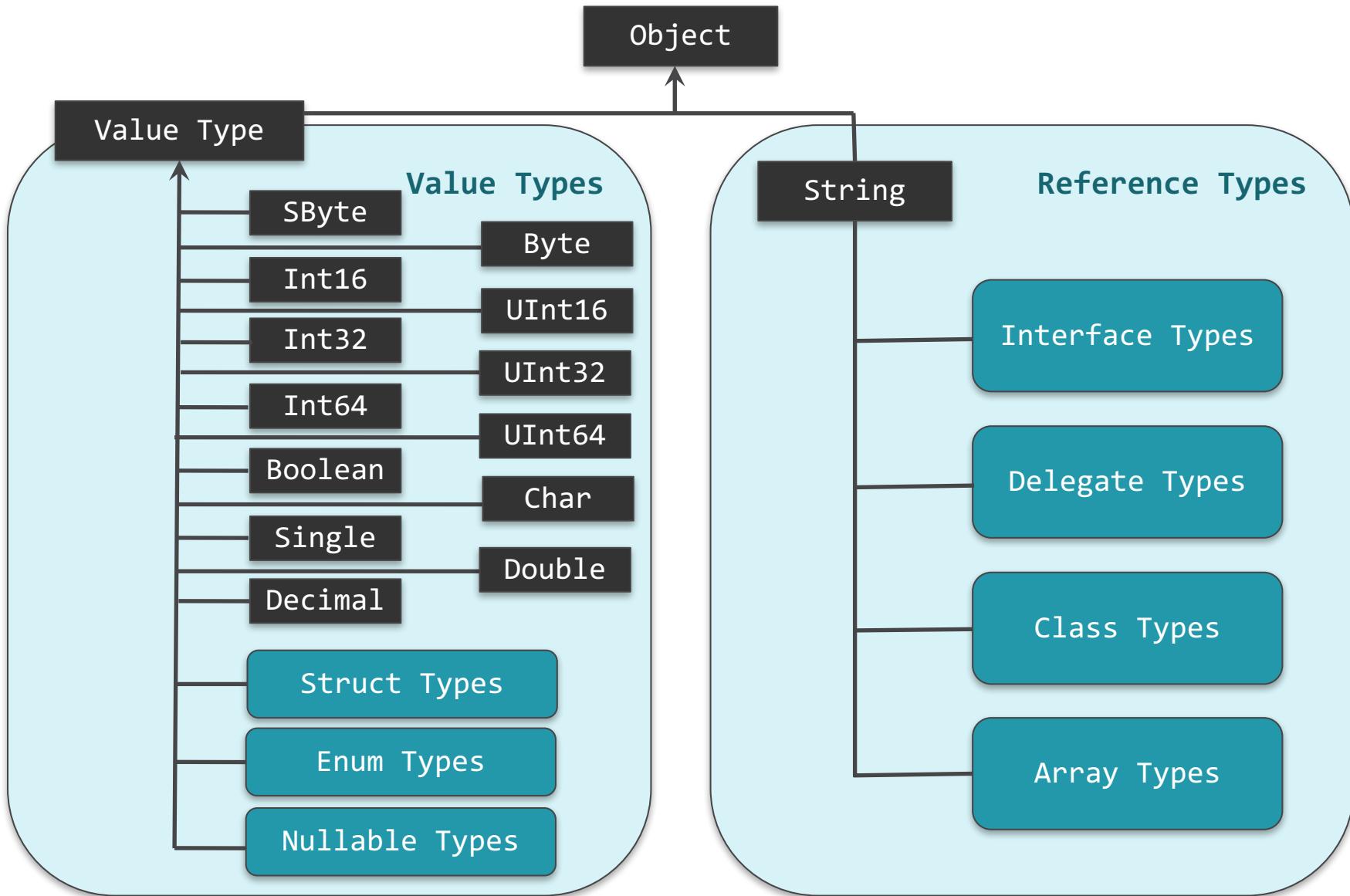
Поле – это именованная единица хранения данных, ассоциированная с типом

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;
```

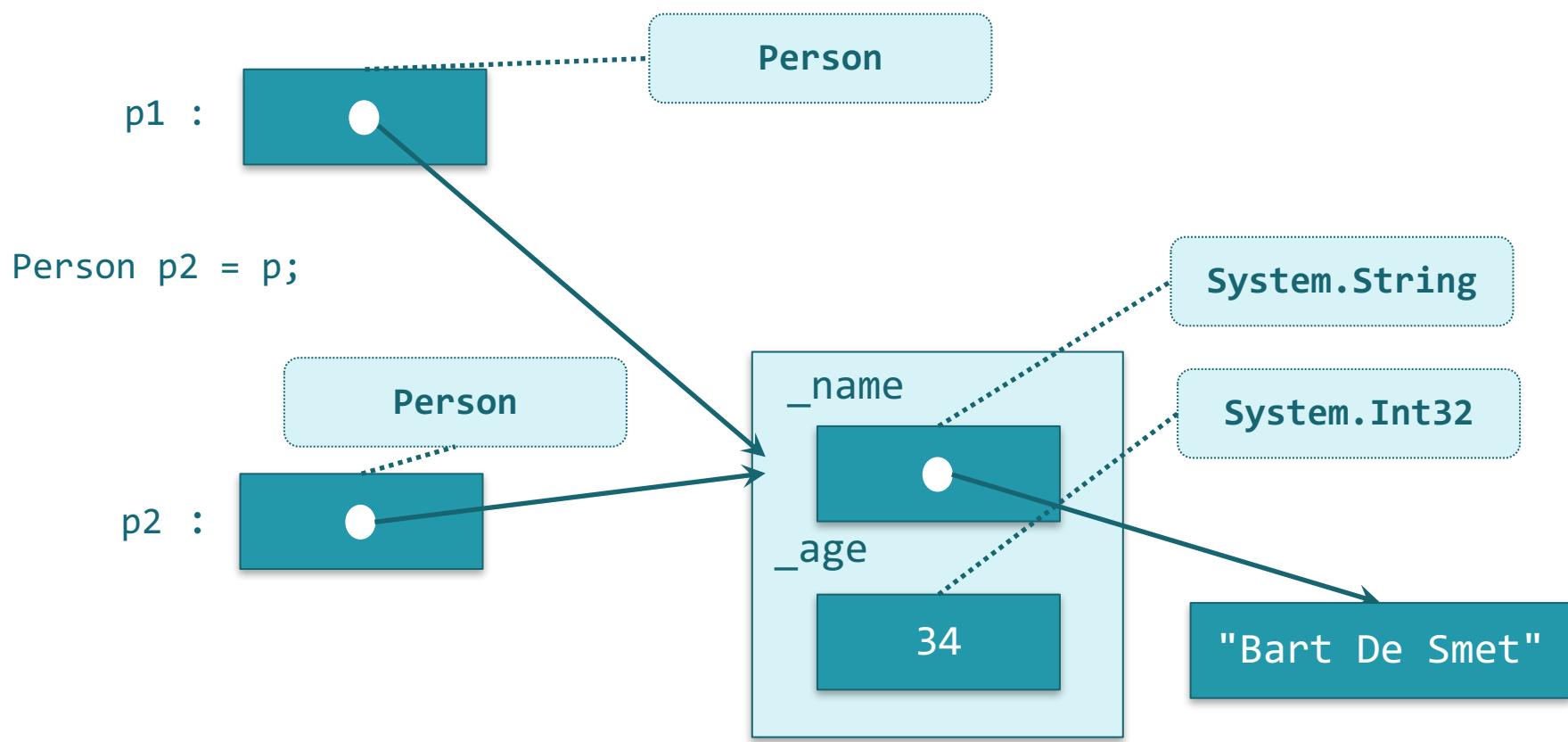
x2 :

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 (*строгая типизация*)

# 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);
    }
}
```



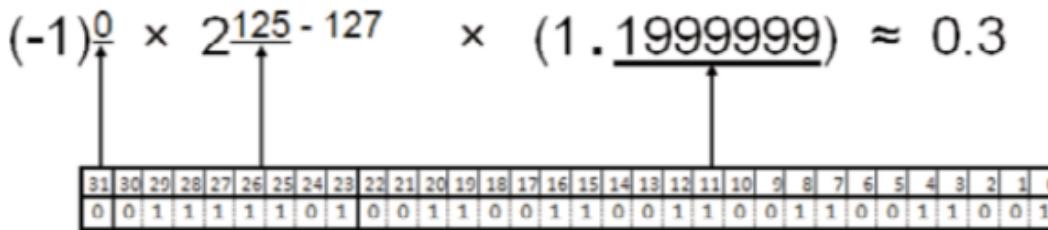
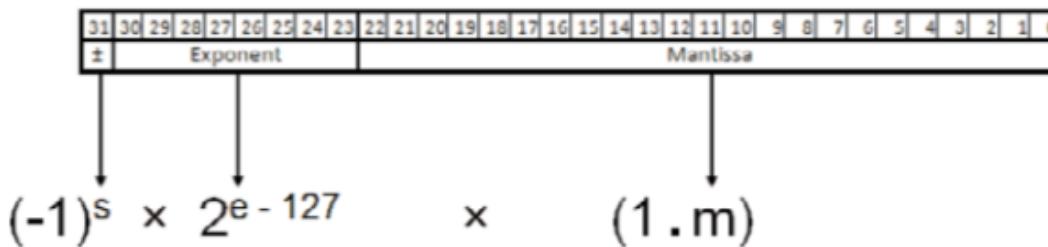
Namespace scope

# 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	Диапазон
bool	System.Boolean	false, true

# Примитивные типы C#. Тип String

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

# Примитивные типы C#. Тип Object

C#	BCL тип	Диапазон
object	System.Object	

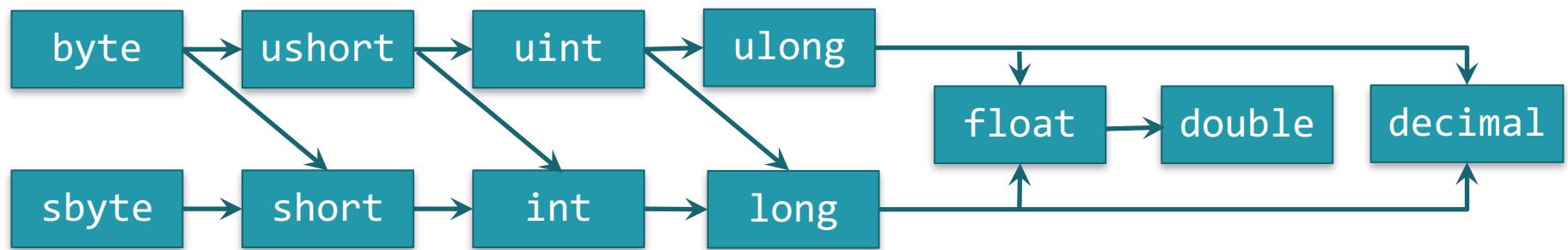
# Примитивные типы C#. Тип dynamic

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

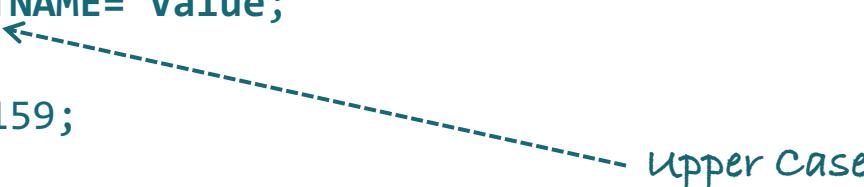
```
const DataType CONSTANTNAME= Value;
```

```
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

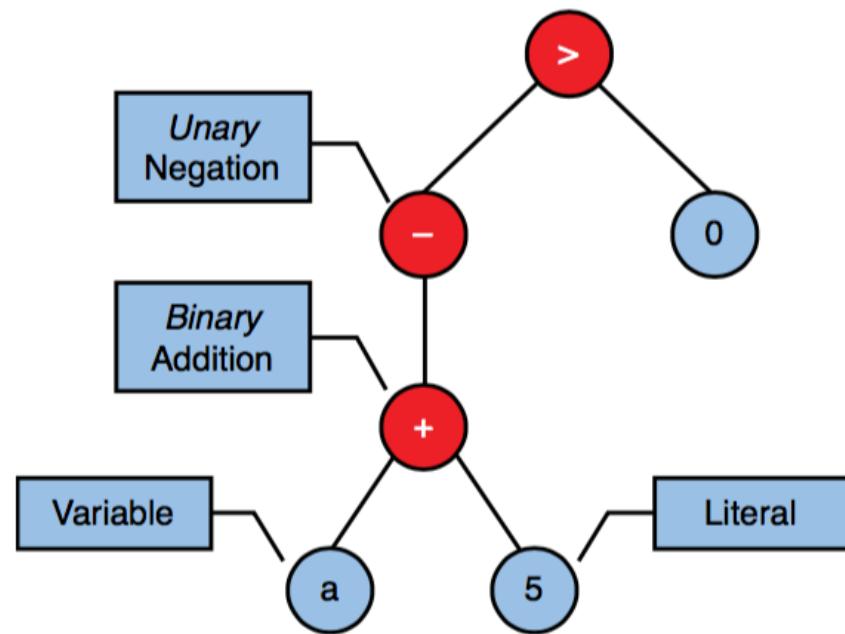
Выражения являются комбинацией operandов и операций

`a + 1`

`(a + b) / 2`

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

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



`-(a + 5) > 0`

# Операции.

Category	Operators
Primary	x.y f(x) a[x] x++ x— new typeof default checked unchecked delegate
Unary	+ , ! ~ ++x —x (T)x
Multiplicative	* / %
Additive	+ -
Shift	<< >>
Relational, type test	< > <= >= is as
Equality	== !=
Logical AND	&
Logical XOR	^
Logical OR	
Conditional AND	&&
Conditional OR	
Null coalescing	??
Conditional	:?
Assignment, lambda	= *= /= %= += -= <<= >>= &= ^=  = =>

# Операции.

Table 2-3. Basic arithmetic operators

Name	Example
Identity (unary plus)	<code>+x</code>
Negation (unary minus)	<code>-x</code>
Post-increment	<code>x++</code>
Post-decrement	<code>x--</code>
Pre-increment	<code>++x</code>
Pre-decrement	<code>--x</code>
Multiplication	<code>x * y</code>
Division	<code>x / y</code>

Table 2-6. Relational operators

Name	Example
Less than	<code>x &lt; y</code>
Greater than	<code>x &gt; y</code>
Less than or equal	<code>x &lt;= y</code>
Greater than or equal	<code>x &gt;= y</code>
Equal	<code>x == y</code>

Table 2-4. Binary integer operators

Name	Example
Bitwise negation	<code>~x</code>
Bitwise AND	<code>x &amp; y</code>
Bitwise OR	<code>x   y</code>
Bitwise XOR	<code>x ^ y</code>
Shift left	<code>x &lt;&lt; y</code>
Shift right	<code>x &gt;&gt; y</code>

Table 2-5. Operators for bool

Name	Example
Logical negation (also known as NOT)	<code>!x</code>
Conditional AND	<code>x &amp;&amp; y</code>
Conditional OR	<code>x    y</code>

# Операции.

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

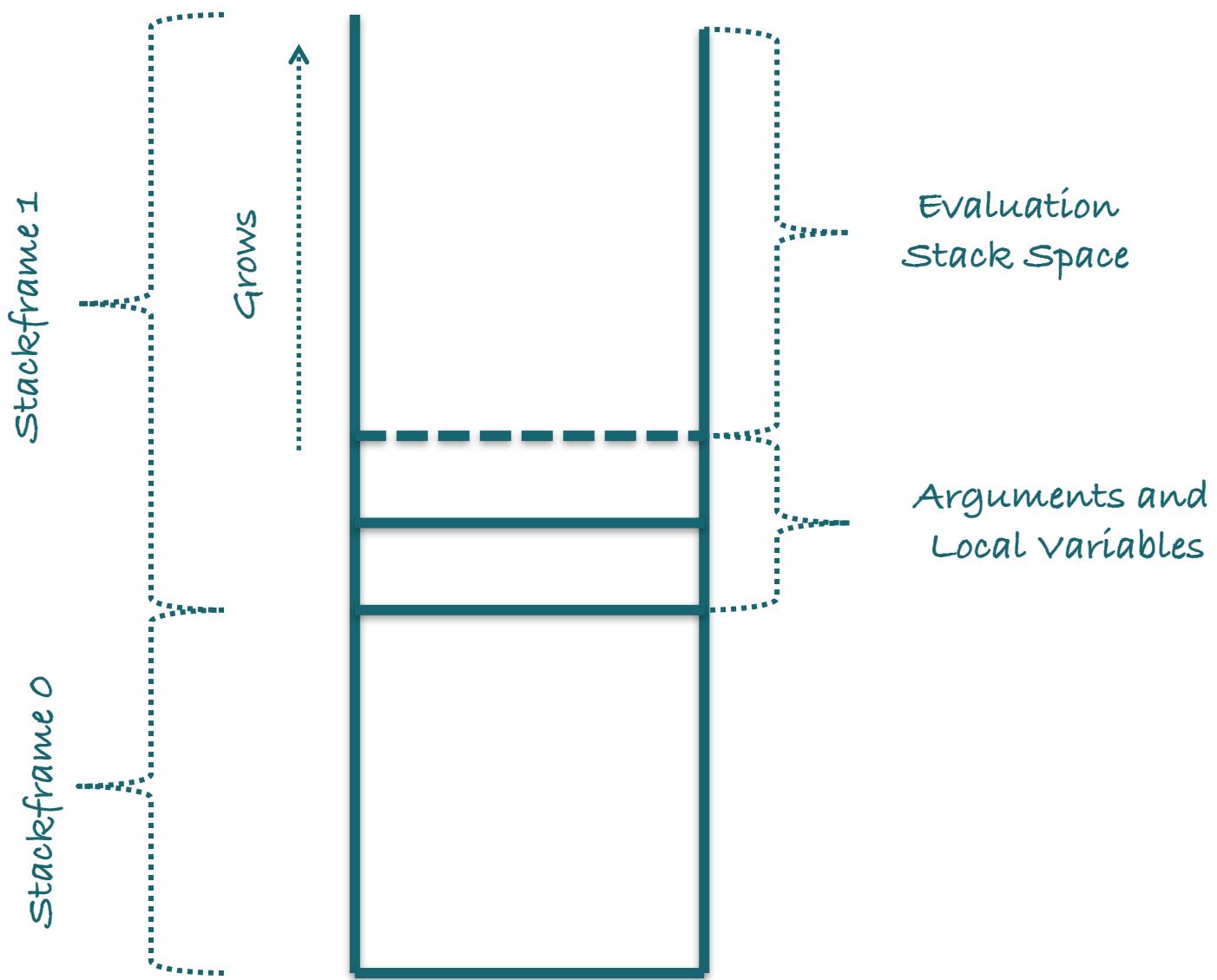
# Стек вычислений (The Evaluation Stack )

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

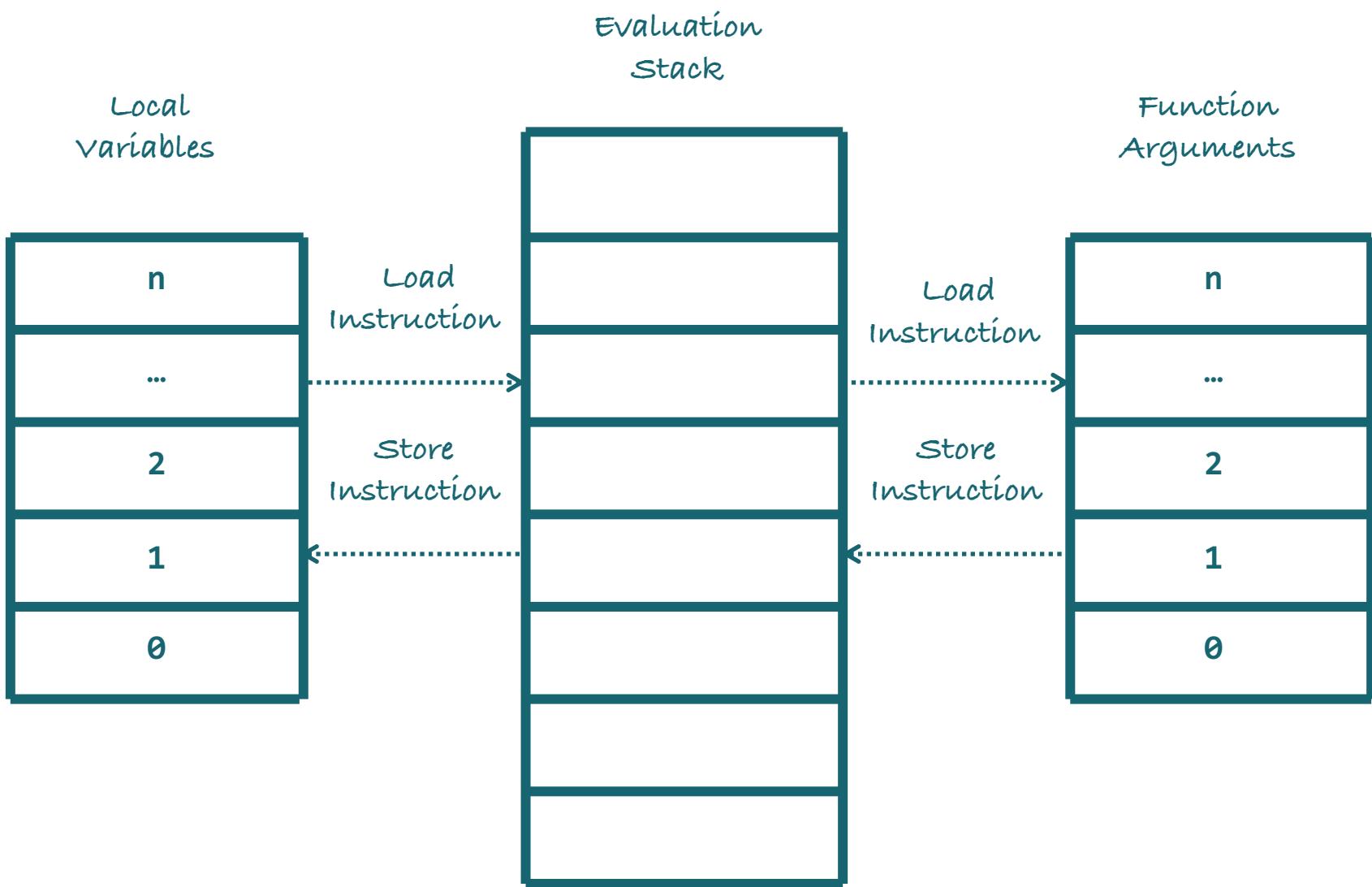
Стек вычислений - это средство просмотра приложения, и его можно использовать для просмотра параметров функции, локальных переменных, временных объектов и т. д. Традиционно параметры функций и локальные переменные помещаются в стек. В .NET эта информация хранится в отдельных репозиториях, в которых память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



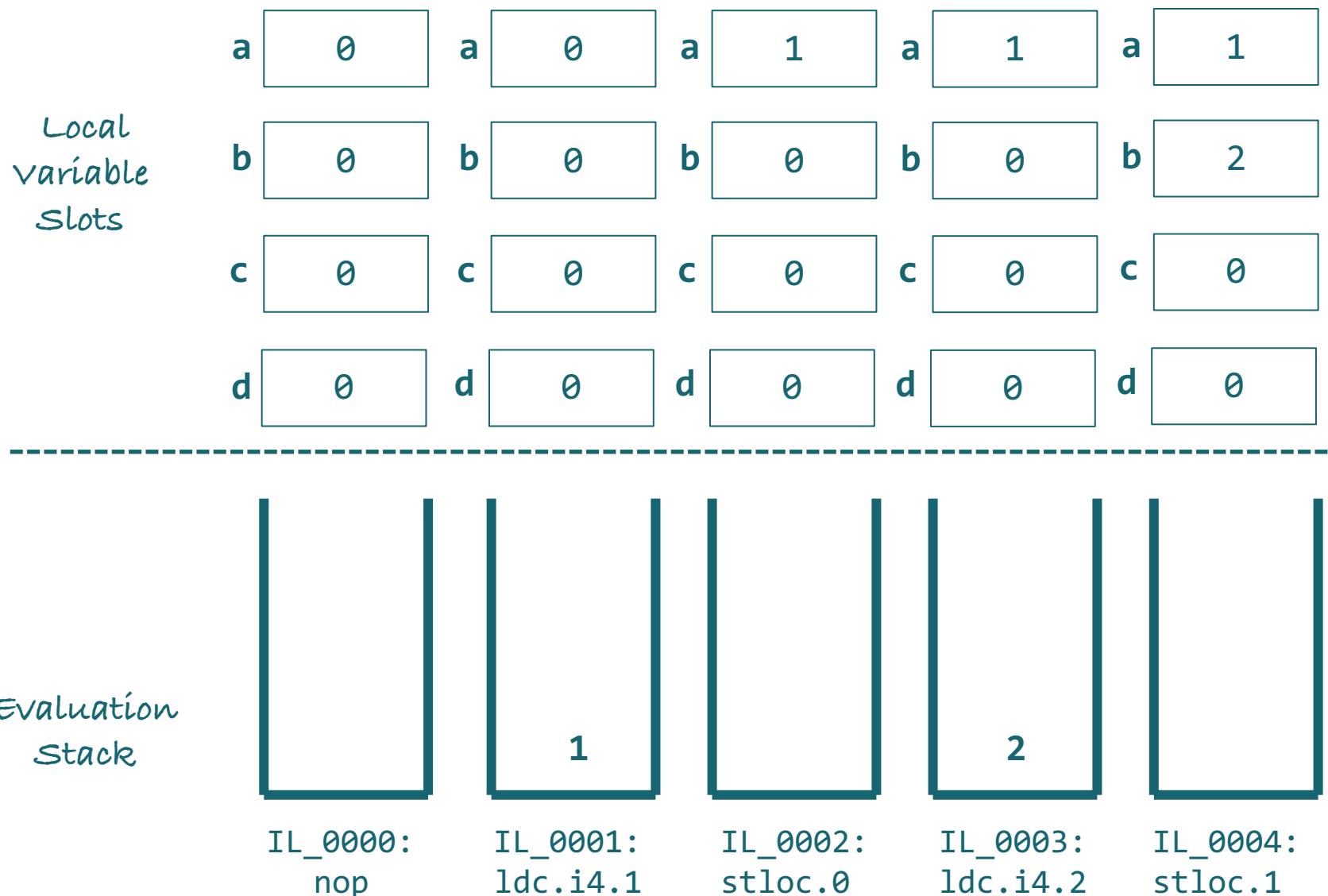
# Стек вычислений

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

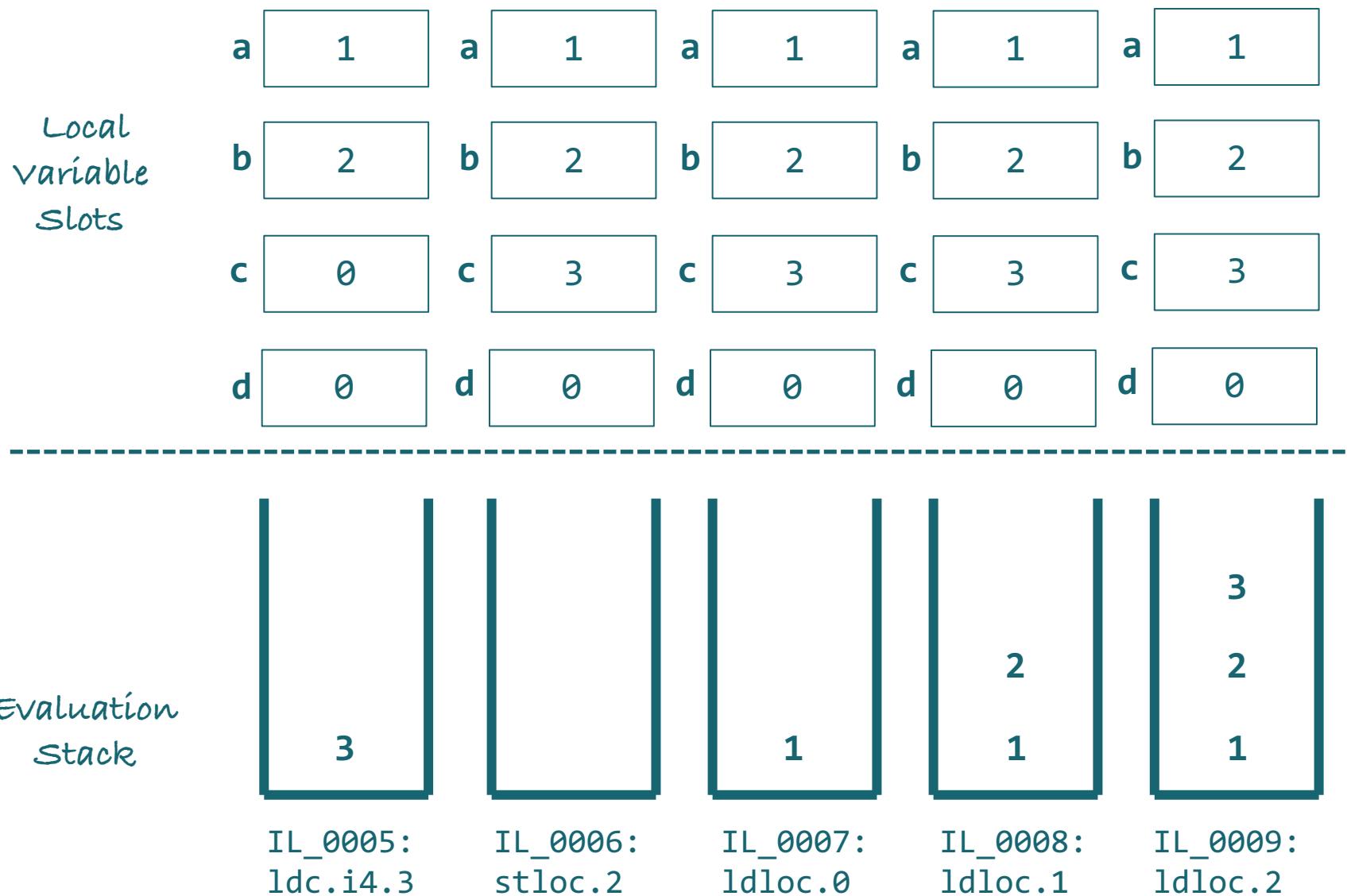
*C# compiler*

```
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
```

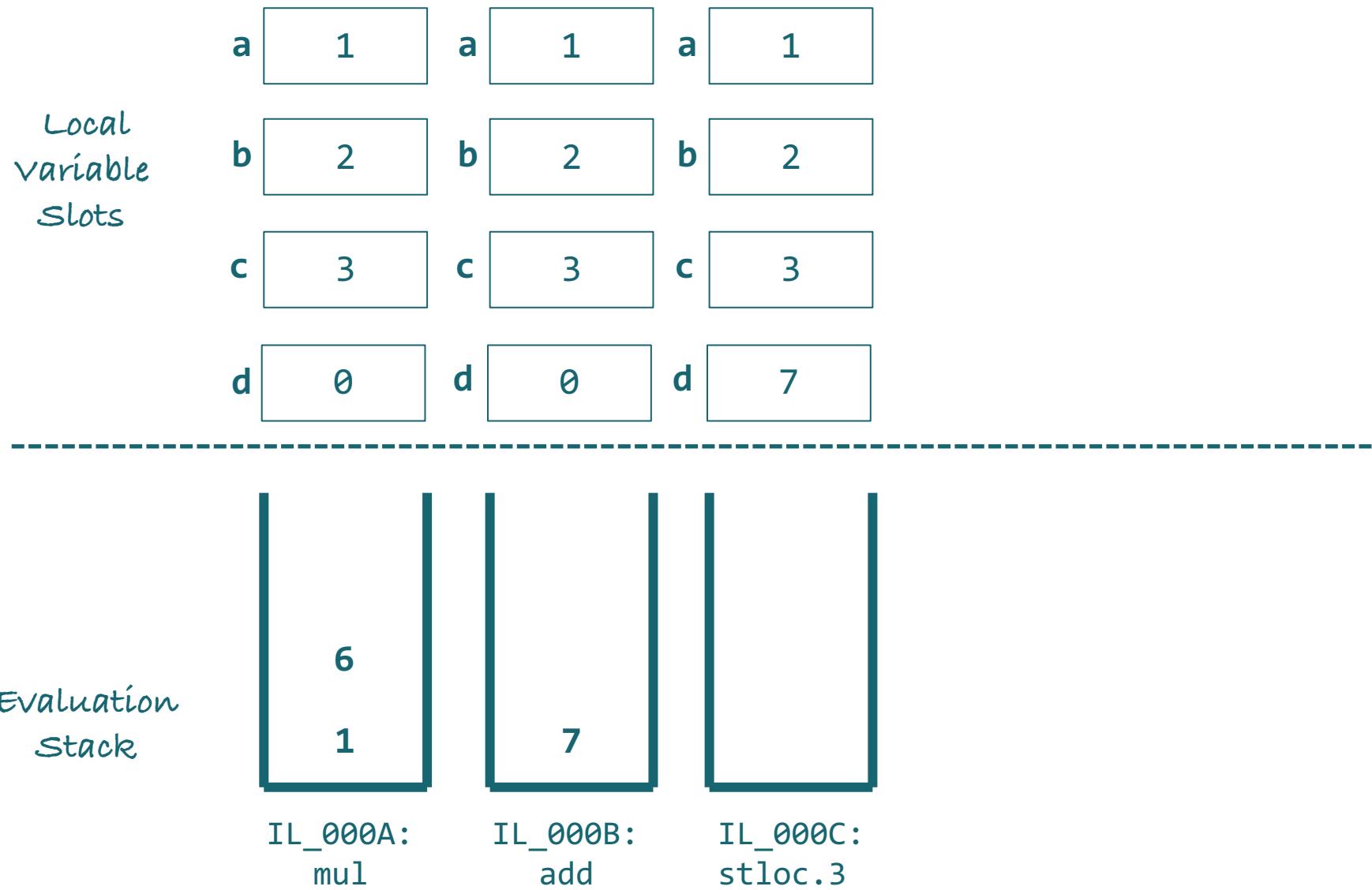
# Стек вычислений



# Стек вычислений



# Стек вычислений



# Стек вычислений

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



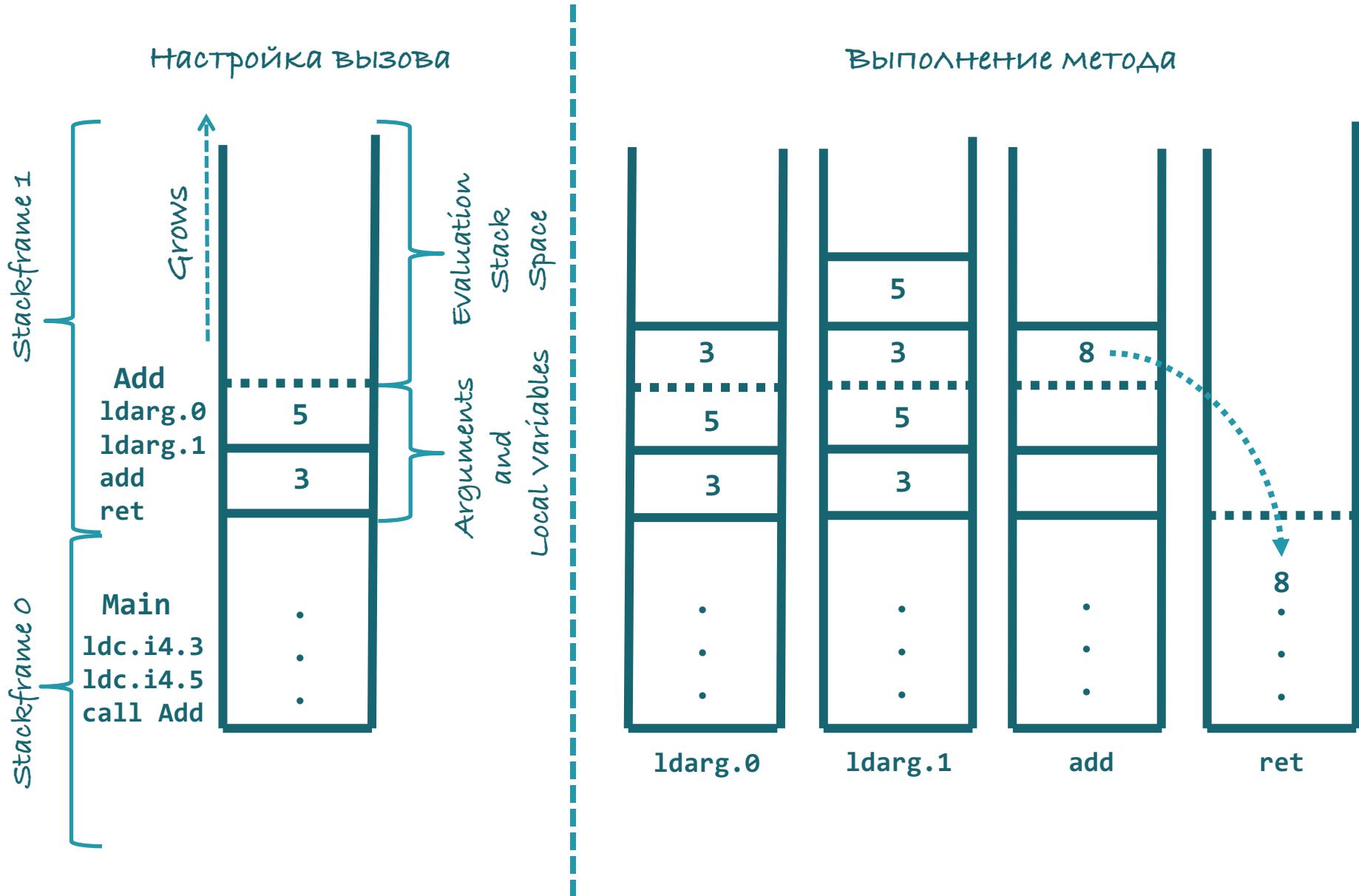
```
IL_0000:  nop  
IL_0001:  ldc.i4.1  
IL_0002:  stloc.0  
IL_0003:  ldloc.0  
IL_0004:  dup  
IL_0005:  ldc.i4.1  
IL_0006:  add  
IL_0007:  stloc.0  
IL_0008:  stloc.1  
IL_0009:  ret
```

```
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

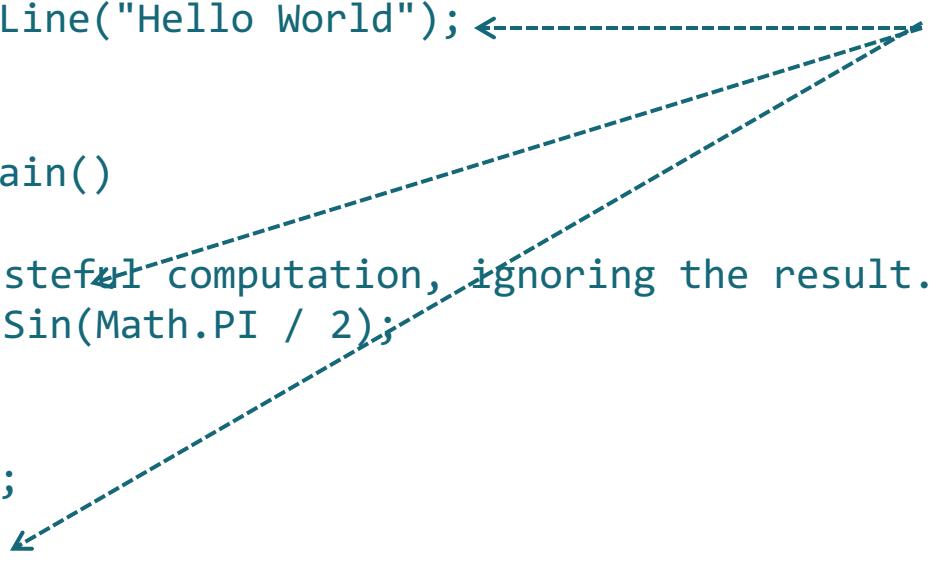
C# поддерживает множество утверждений, которые составляют таксономию следующим образом

- 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 consisting of three dashed arrows originating from specific code snippets and pointing towards the 'Expression statements' text. One arrow points from the 'Console.WriteLine' call, another from the trigonometric calculation in the Main method, and a third from the assignment and increment operation at the bottom.

# 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(); ←
    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

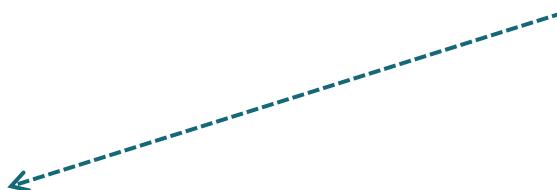
Массив представляет собой набор объектов, которые сгруппированы вместе и управляются как единое целое. Массивы имеют следующие характеристики:

- 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 };
```

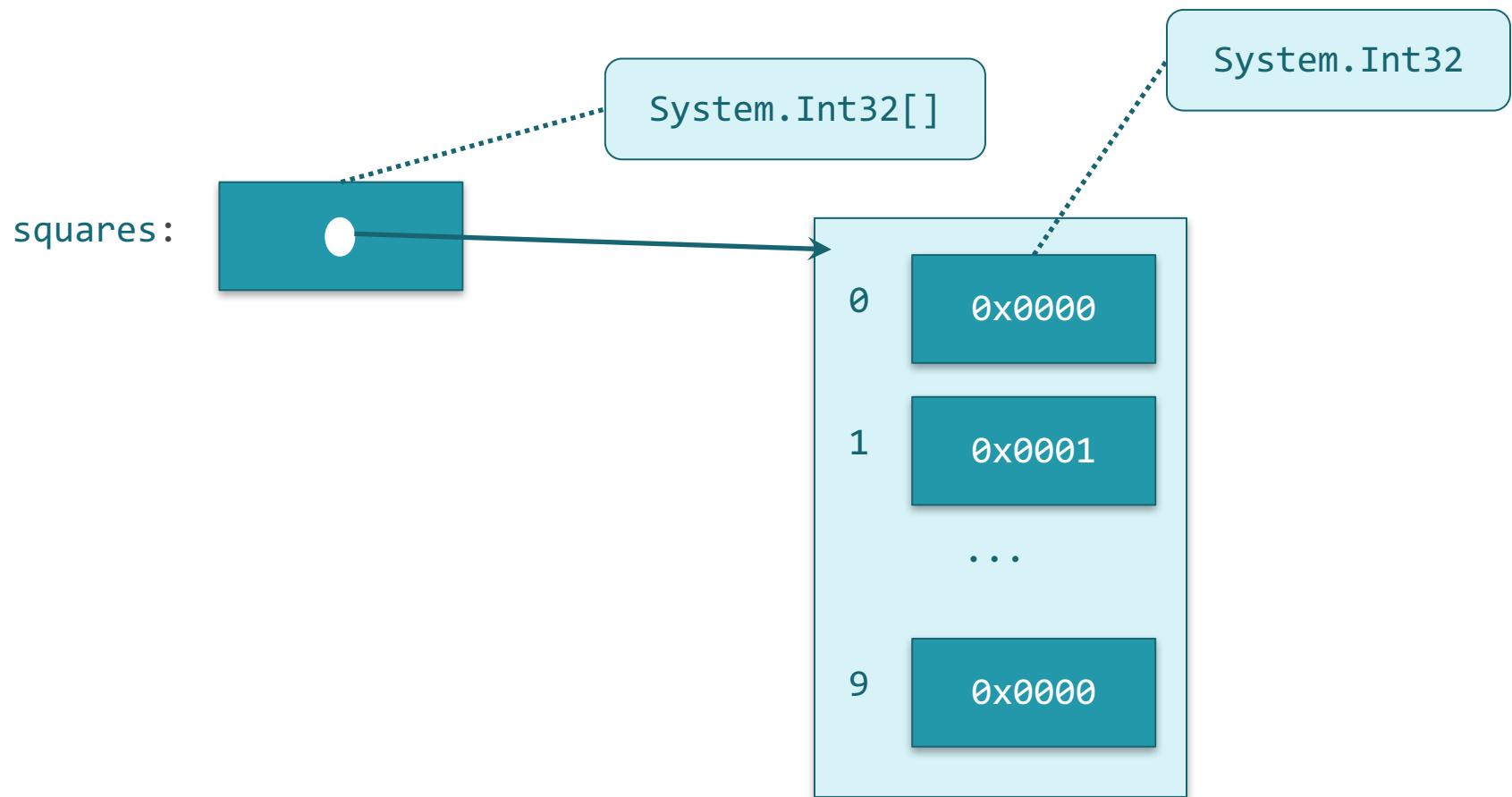
Одномерные (single, sz)  
массивы



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

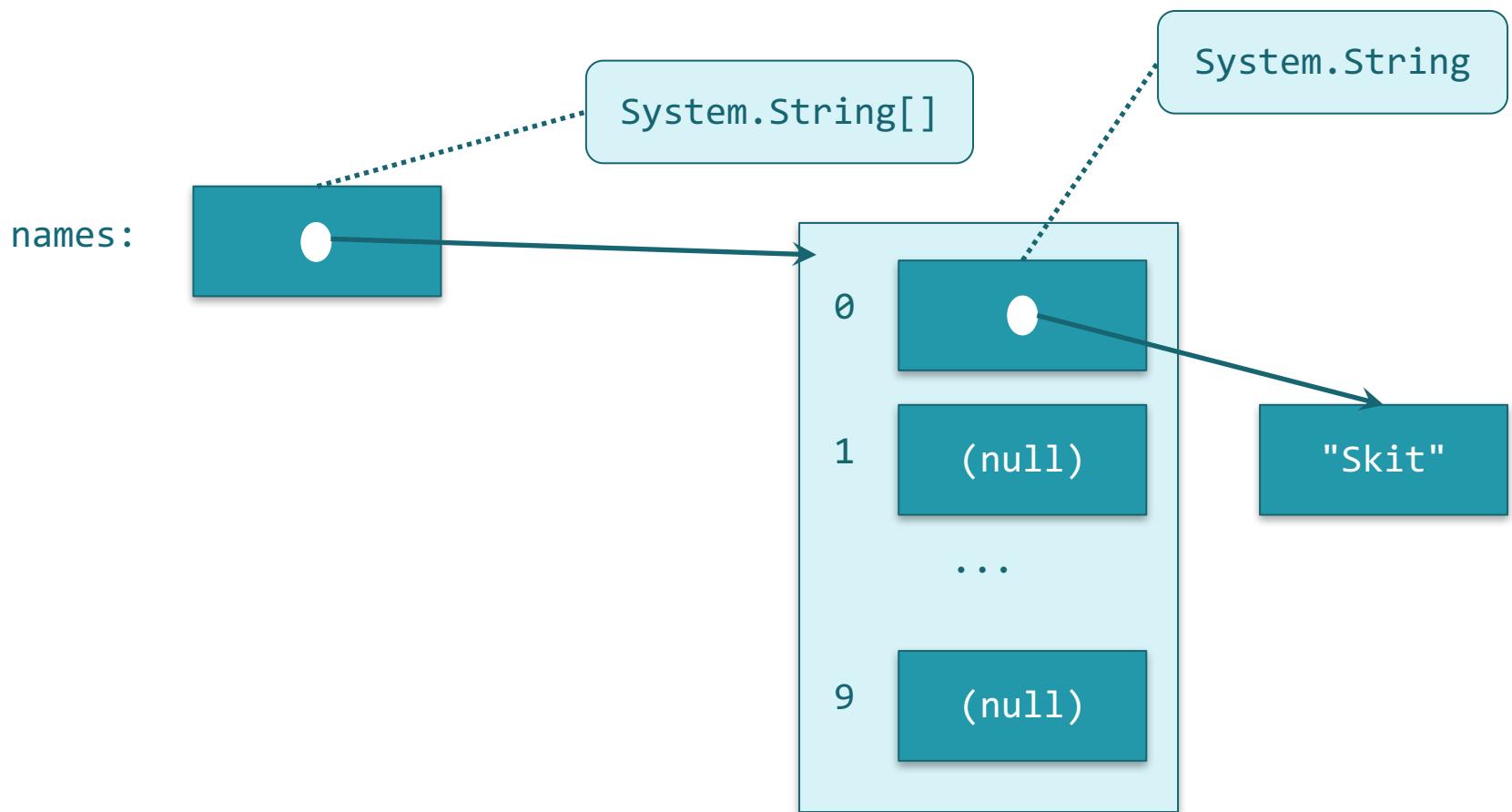
# Одномерные массивы

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



# Одномерные массивы

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



# Многомерные массивы

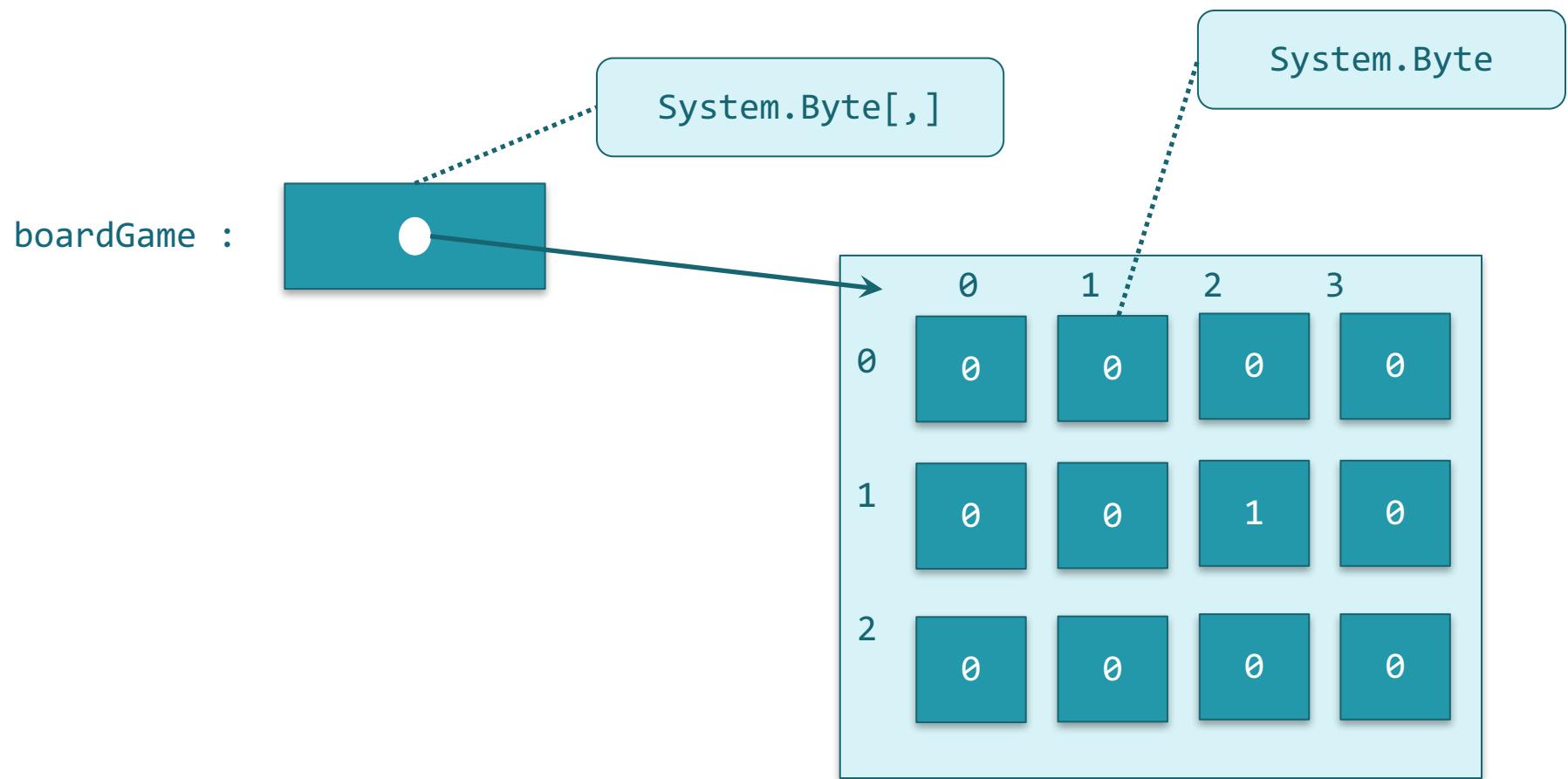
```
int[,] table; // two-dimensional array  
table = new int[10, 2];  
.  
.  
int[, , ] cube = new int[3, 2, 5]; // three-dimensional array
```

Многомерные (multiple)  
массивы

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

# Многомерные массивы

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



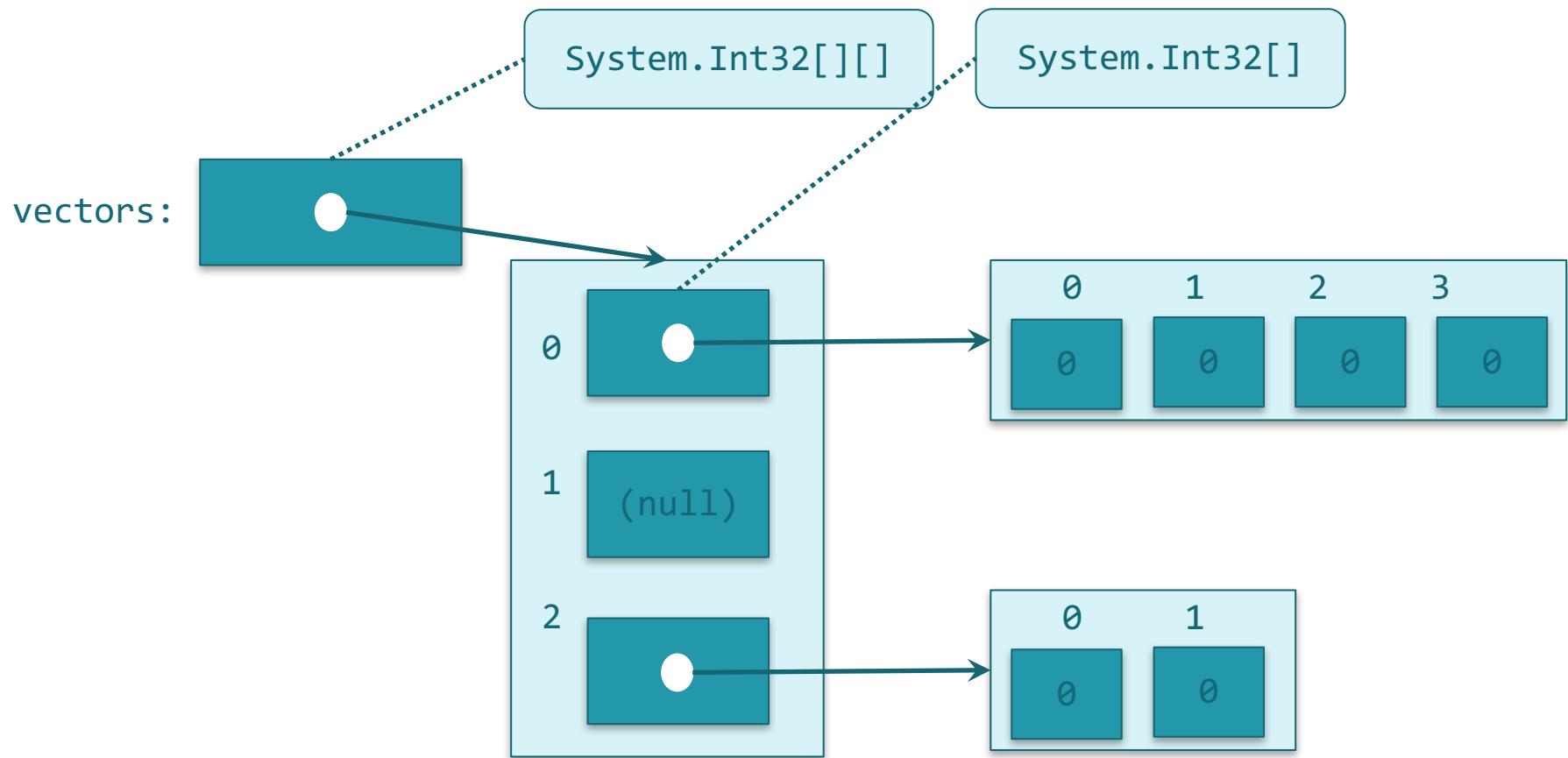
# Зубчатые массивы (jagged arrays)

```
Type [][] jaggedArray = new Type[10][]; ← Jagged array
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

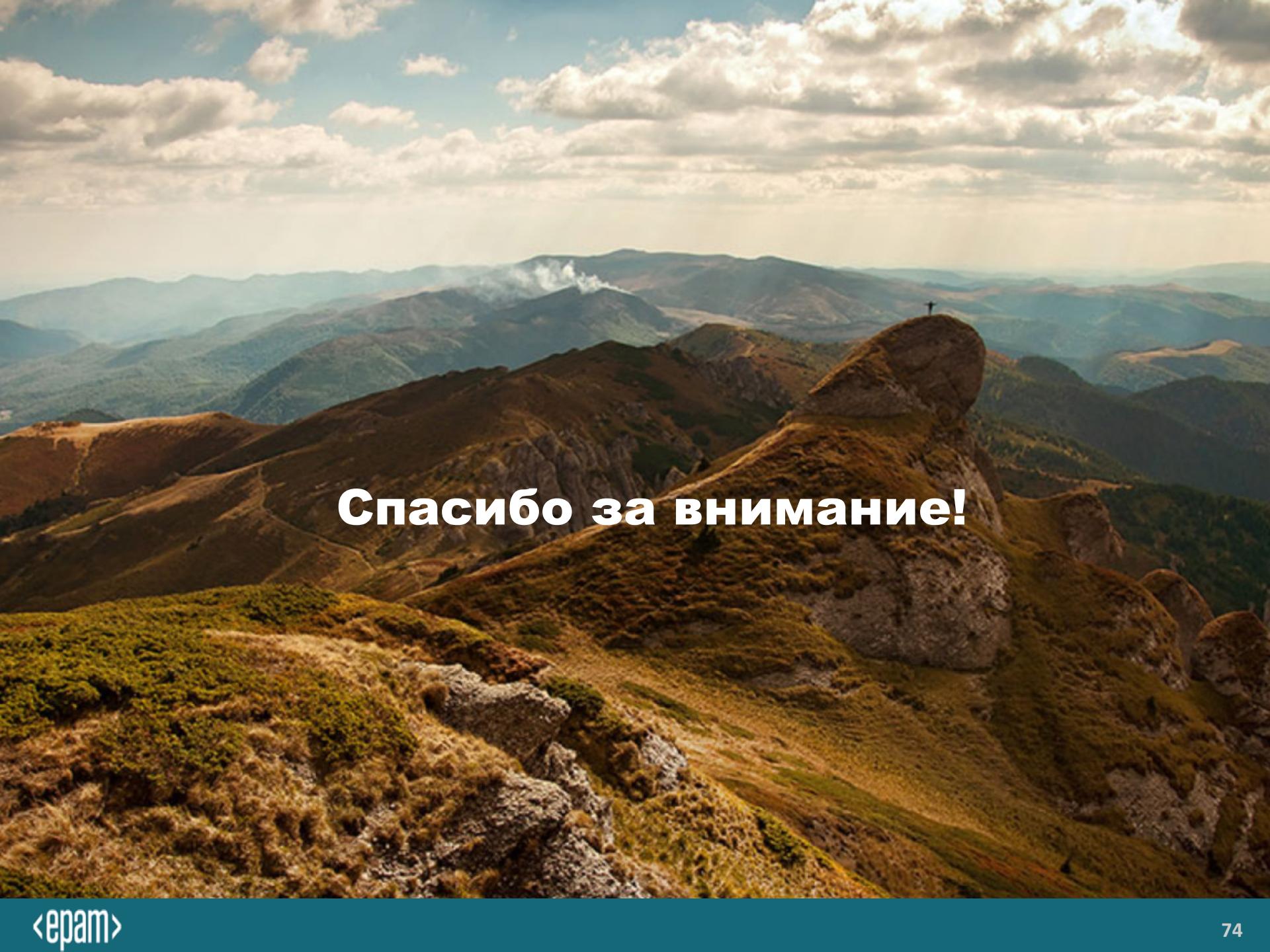
```
int[] integers = new int[3] { 1, 2, 3 };
object[] objects = new object[3];
objects = integers;
```

СТЕ

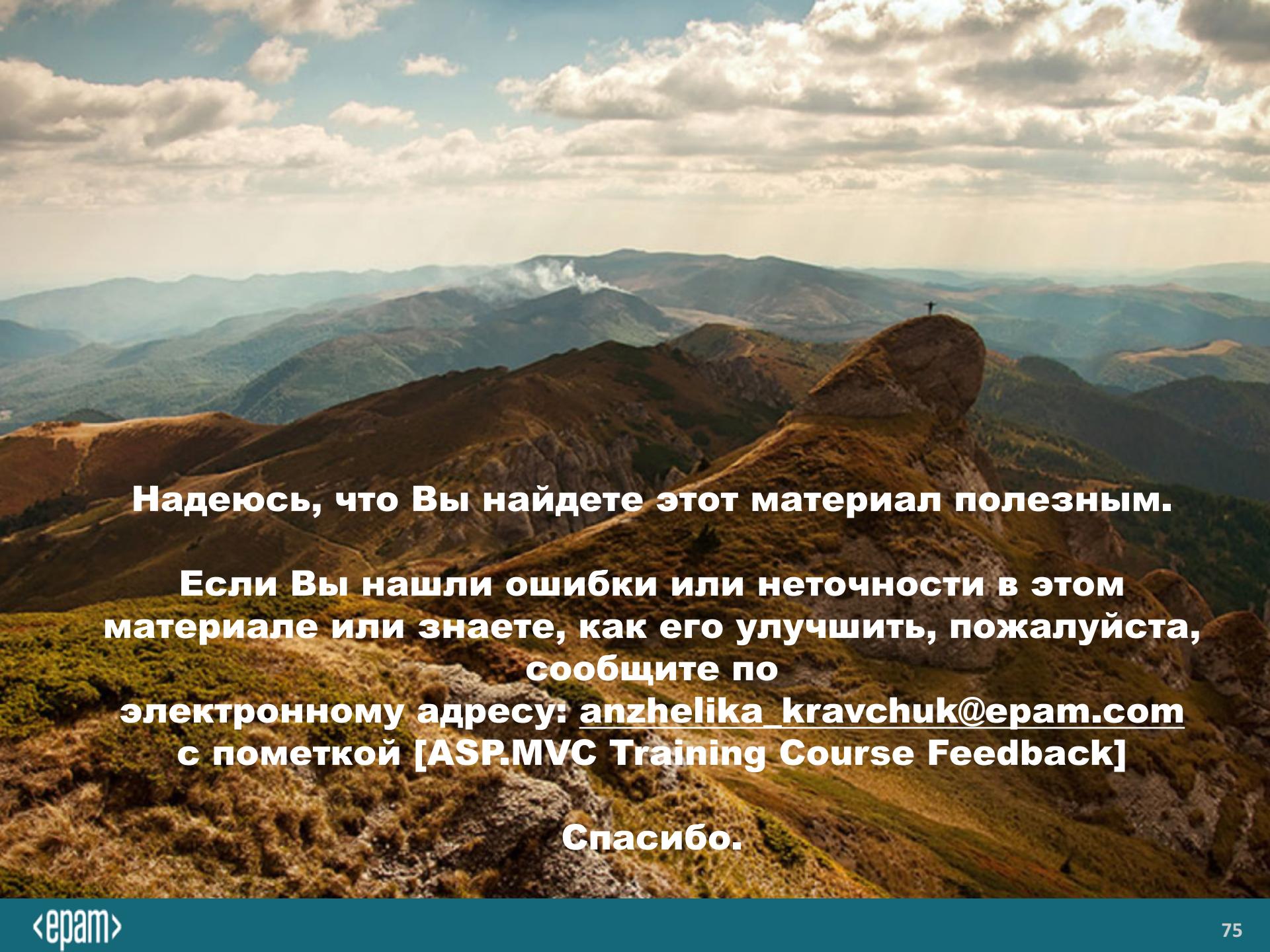
```
int[] integers = new int[5];
object[] objects = new object[integers.Length];
Array.Copy(integers, objects, integers.Length);
```

## 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

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 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](mailto:anzhelika_kravchuk@epam.com)  
с пометкой [ASP.MVC Training Course Feedback]**

**Спасибо.**