



METHODS IN DETAILS

.NET LAB, 2018

Anzhelika KRAVCHUK

Определение метода

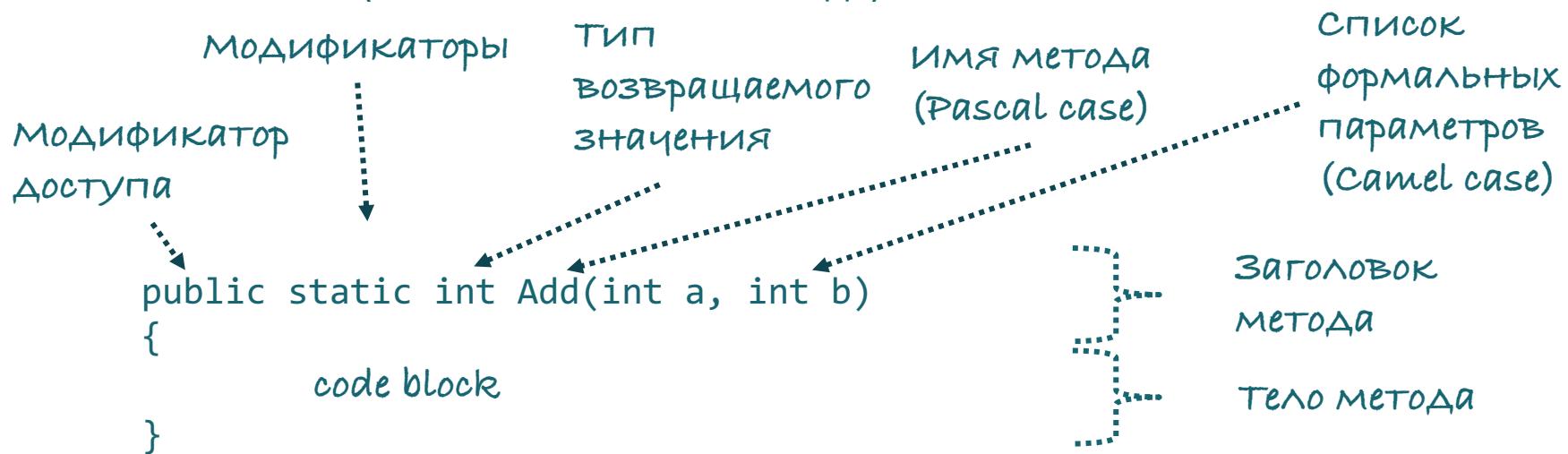
Методы это имплементация поведения типа

Метод - это член класса, который содержит блок кода, который представляет действие:

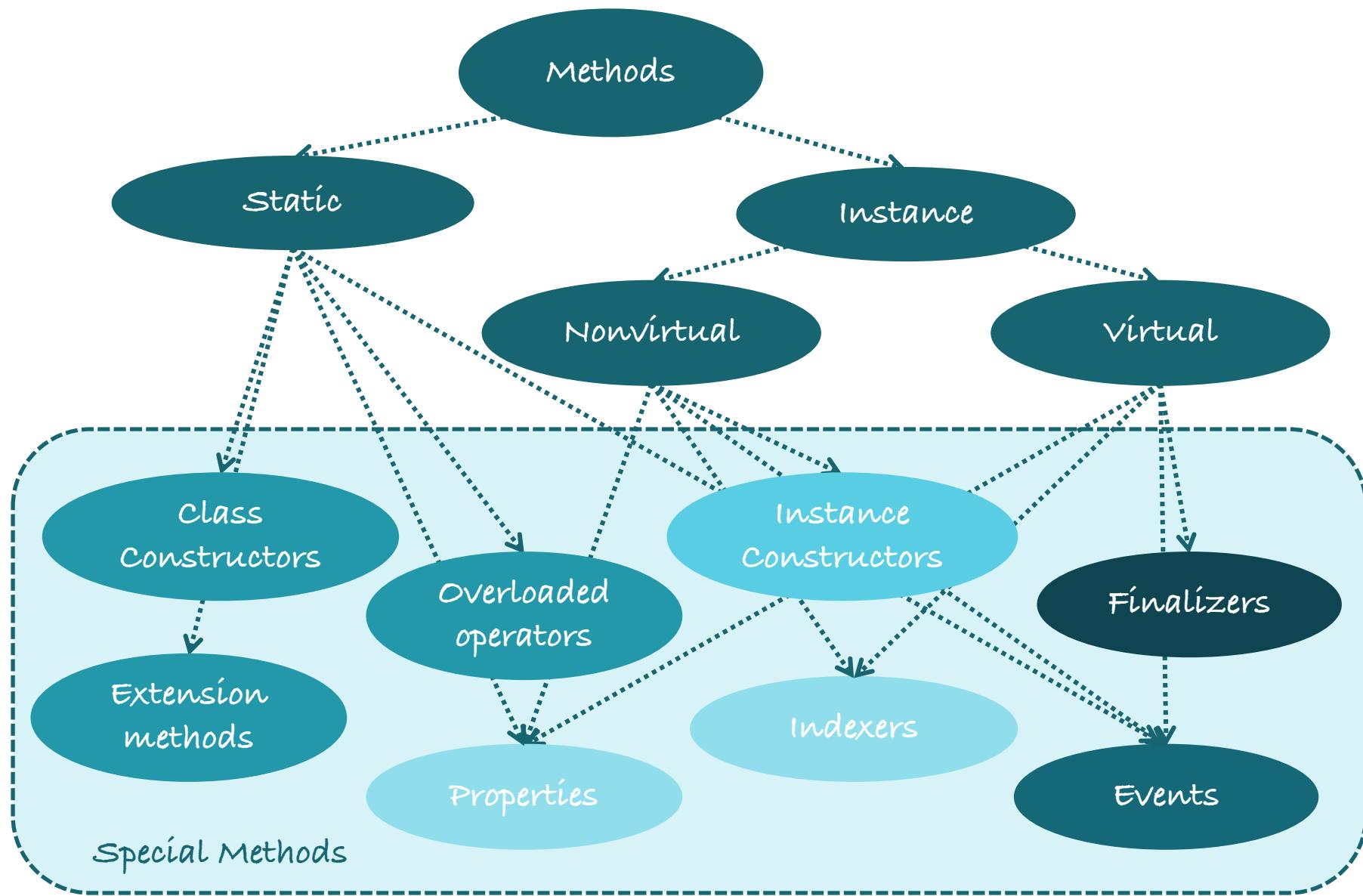
- весь исполняемый код принадлежит методу
- любое C# -приложение должно иметь хотя бы один метод

Объявление метода состоит из двух частей:

- заголовок метода
- необязательный блок кода реализации, определяющий действия, которые может выполнять тип (называемый телом метода).



Типы методов



Модификаторы метода

Статический модификатор

`static`

Модификаторы доступа

`public internal private protected`

Модификаторы наследования

`new virtual abstract override sealed`

Модификаторы небезопасного кода

`unsafe extern`

Модификатор частичного метода

`partial`

Модификатор асинхронного кода

`async`

Именование методов

- Использовать при именовании метода глаголы или словосочетания, что помогает понять назначение кода
- Использовать соглашение именования методов «Pascal case» (Pascal naming convention) независимо от модификаторов доступа.

Параметры и аргументы метода

Параметры указываются в месте объявления метода (в списке параметров метода), тогда как аргументы указываются в месте вызова (при вызове метода).

именуются в соответствии с
соглашением именования «Camel case»

```
void SomeMethod(int intData) {...} ← параметры
```

```
void SomeMethod(int moreIntData, float floatData) {...}
```

```
SomeMethod(10); ← аргументы
```

```
float f = 54.321F;
```

```
SomeMethod(100, f);
```

```
string Concat(string str0, string str1, string str2)
```

```
bool TryParse(string s, out int result)
```

```
int Exchange(ref int location1, int value)
```

```
void WriteLine(string format, params object[] arg)
```

допустимые
заголовки методов с
различным
количеством
параметров и
некоторыми
модификаторами

Возвращаемое значение метода

Тип возвращаемого метода может быть либо типом, либо `void`. Он указывает на то, что вызывающий код получит (при нормальных обстоятельствах) в результате вызова метода.

```
public static int Div(int n, int d)
{
    if (d == 0)
        throw new ArgumentOutOfRangeException(nameof(d));

    return n / d;
}
```

Когда указан тип возвращаемого значения, все ветки выполнения в теле метода должны достигать точки, в которой ключевое слово `return` используется для передачи результата вызывающему коду.

Метод может генерировать исключение

Методы-выражения (Expression-bodied methods). C# 6.0

```
class Customer
{
    public string First { get; }
    public string Last { get; }

    public static implicit operator string(Customer p) =>
        p.First + " " + p.Last;

    public void Print() => Console.WriteLine(First + " " + Last);

    public string Name => First + " " + Last;
    public Customer this[long id] => store.LookupCustomer(id);
}
```

Метод, который содержит единственное выражение, может быть записан более кратко в качестве метода выражения. Стрелка "`=>`" заменяет фигурные скобки и ключевое слово `return`:

```
p.First + " " + p.Last;
```

Expression-bodied синтаксис может быть использован с свойствах и индексаторах

Expression-bodied синтаксис можно использовать также в методах с типом возвращаемого значения `void`

Методы-выражения (Expression-bodied methods). C# 7.0

С #7.0 добавляет возможность использования
аксессоров, конструкторов и финализаторов в
список методов, которые могут использоваться
как expression-bodied методы

```
class Person
{
    private static ConcurrentDictionary<int, string> names = new
        ConcurrentDictionary<int, string>();
    private int id = GetId();

    public Person(string name) => names.TryAdd(id, name);

    ~Person() => names.TryRemove(id, out *); // :)

    public string Name
    {
        get => names[id];
        set => names[id] = value;
    }
}
```

Выброс исключений C# 7.0

В C# 7.0 разрешено выбрасывать исключения в указанных местах кода

```
class Person
{
    public string Name { get; }
    public Person(string name) => Name = name ??
                                throw new ArgumentNullException(name);
    public string GetFirstName()
    {
        var parts = Name.Split(' ');
        return (parts.Length > 0) ? parts[0] :
                                throw new InvalidOperationException("No name!");
    }

    public string GetLastName() => throw new NotImplementedException();
}
```

Вызов метода

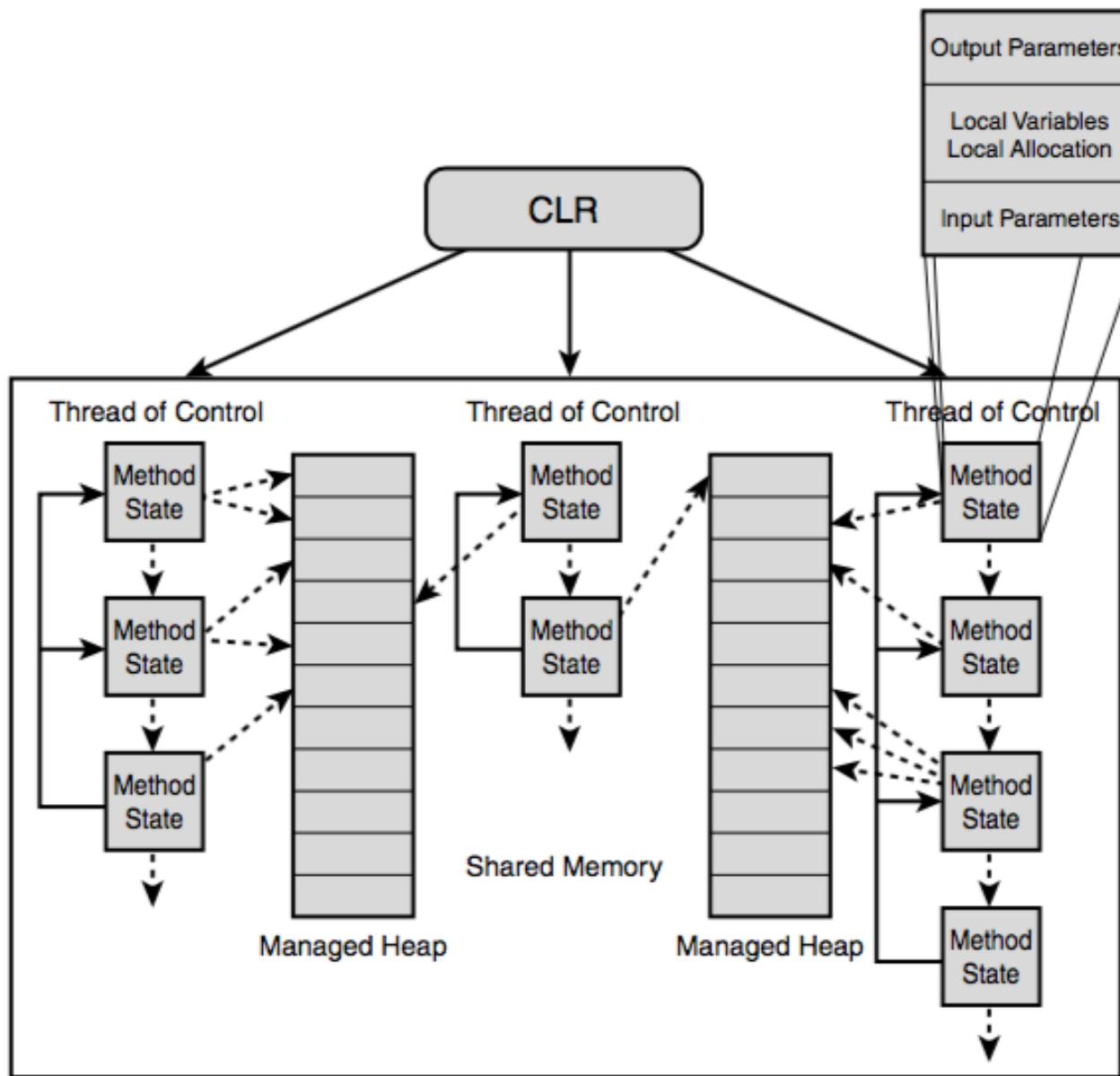
Для вызова метода необходимо:

- указать имя метода
- предоставить в скобках аргументы для соответствующих параметров метода
- если метод возвращает значение, необходимо указать, как использовать это значение

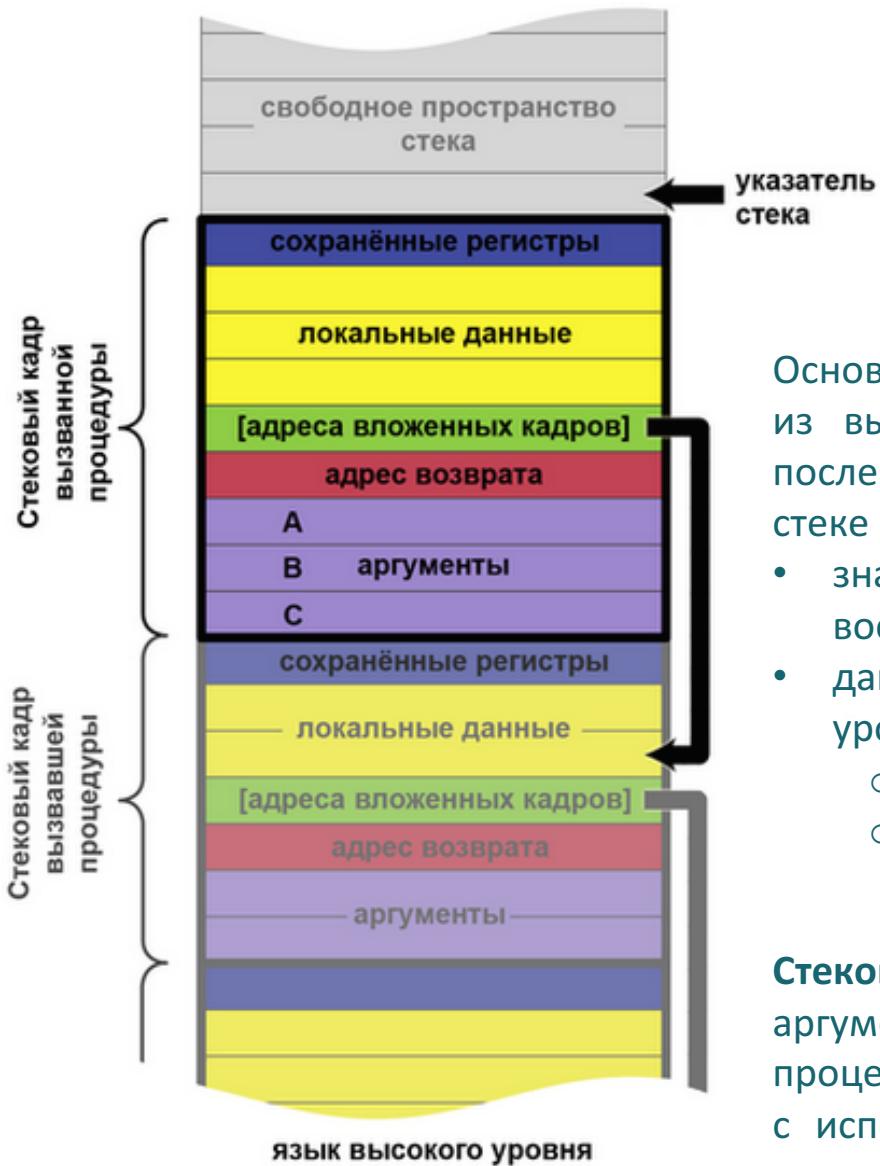
```
int i = 1;  
int j = 2;           .....  
int result = Sum(i++, i+j); ←  
  
int Sum(int first, int second)  
{  
    return first + second;  
}
```

Аргументы метода вычисляются в строгом порядке слева направо. Это может иметь значение, если изменение значения одного аргумента влияет на значение другого

Упрощенное представление о состоянии CLR (*Machine state under the CLR*)



Стек вызовов. Стековый фрейм



Стек вызовов – LIFO-стек, хранящий информацию для возврата управления из подпрограмм (процедур) в программу (или подпрограмму, при вложенных или рекурсивных вызовах) и/или для возврата в программу из обработчика прерывания (в том числе при переключении задач в многозадачной среде).

Основное назначение – отслеживать место, куда каждая из вызванных процедур должна вернуть управление после своего завершения. Кроме адресов возврата в стеке могут сохраняться другие данные

- значения регистров с их последующим восстановлением
- данные стекового кадра (фрейма) языков высокого уровня:
 - аргументы, переданные в функцию
 - локальные переменные – временные данные функции

Стековый кадр (фрейм) – механизм передачи аргументов и выделения временной памяти (в процедурах языков программирования высокого уровня) с использованием системного стека; ячейка памяти в стеке.

Выполнение метода CLR

```
public static class GreetingClass
{
    public static string GreetingMethod(string userName)
    {
        if (userName == null)
        {
            throw new ArgumentNullException(nameof(userName));
        }

        if (userName == string.Empty)
        {
            return "Hello, anonim!";
        }

        return $"Hello, {userName}!";
    }
}
```

Выполнение метода CLR. Стек виртуальной машины

The size of the evaluation stack

```
.method public hidebysig static string GreetingMethod(string userName) cil managed
{
    // Code size          61 (0x3d)
    .maxstack 2
    .locals init ([0] bool V_0,
                 [1] bool V_1,
                 [2] string V_2)
    IL_0000: nop
    IL_0001: ldarg.0
    IL_0002: ldnull
    IL_0003: ceq
    IL_0005: stloc.0
    IL_0006: ldloc.0
    IL_0007: brfalse.s IL_0015
    IL_0009: nop
    IL_000a: ldstr      "userName"
    IL_000f: newobj     instance void [mscorlib]System.ArgumentNullException::ctor(string)
    IL_0014: throw
    IL_0015: ldarg.0
    IL_0016: ldsfld     string [mscorlib]System.String::Empty
    IL_001b: call        bool [mscorlib]System.String::op_Equality(string, string)

    IL_0020: stloc.1
    IL_0021: ldloc.1
    IL_0022: brfalse.s IL_002d
    IL_0024: nop
    IL_0025: ldstr      "Hello, anonim!"
    IL_002a: stloc.2
    IL_002b: br.s       IL_003b
    IL_002d: ldstr      "Hello, {0}!"
    IL_0032: ldarg.0
    IL_0033: call        string [mscorlib]System.String::Format(string, object)

    IL_0038: stloc.2
    IL_0039: br.s       IL_003b
    IL_003b: ldloc.2
    IL_003c: ret
} // end of method GreetingClass::GreetingMethod
```

Instructions

A description of the locals array

Exception Handling Array

Signature

Intermediate language (IL)

Промежуточный язык (IL) - это объектно-ориентированный язык программирования, предназначенный для использования компиляторами для .NET Framework перед статической или динамической компиляцией в машинный код.

IL используется .NET Framework для генерации машинно-независимого кода как результат компиляции исходного кода, написанного на любом языке программирования .NET.

IL - это язык ассемблера на основе стека, который преобразуется в байтовый код во время выполнения виртуальной машины. Он определяется спецификацией общей языковой инфраструктуры (CLI).

IL также известен как Microsoft intermediate language (MSIL) (MSIL) или common intermediate language (CIL).

https://en.wikipedia.org/wiki/List_of_CIL_instructions

[Inside Microsoft .NET IL Assembler](#)

[Expert .NET 2.0 IL Assembler](#)

[Compiling for the .NET Common Language Runtime \(CLR\)](#)

Основные черты MSIL

Компиляторы, ориентированные на платформу .NET, должны генерировать код на MSIL, являющимся языком ассемблера некоторой виртуальной машины.

Основные черты архитектуры виртуальной машины MSIL

- машина является стековой, причем стек является статически типизированным – в каждой точке программы JIT-compiler должен иметь возможность статически определить типы содержимого всех ячеек стека;
- стек используется, как правило, только для хранения промежуточных результатов вычисления;
- большинство команд MSIL получают свои аргументы в стеке, удаляют их из стека и помещают вместо них результат(ы) вычисления.
- машина является объектно-ориентированной: структура MSIL отражает разбиение кода на классы, методы и т.п.

Специфика хранения переменных в MSIL

Существует несколько вариантов хранения переменных в MSIL:

- в статической области памяти, существующей все время выполнения программы;
- в локальной области, которая выделяется при входе в метод;
- внутри объекта, размещенного в куче.

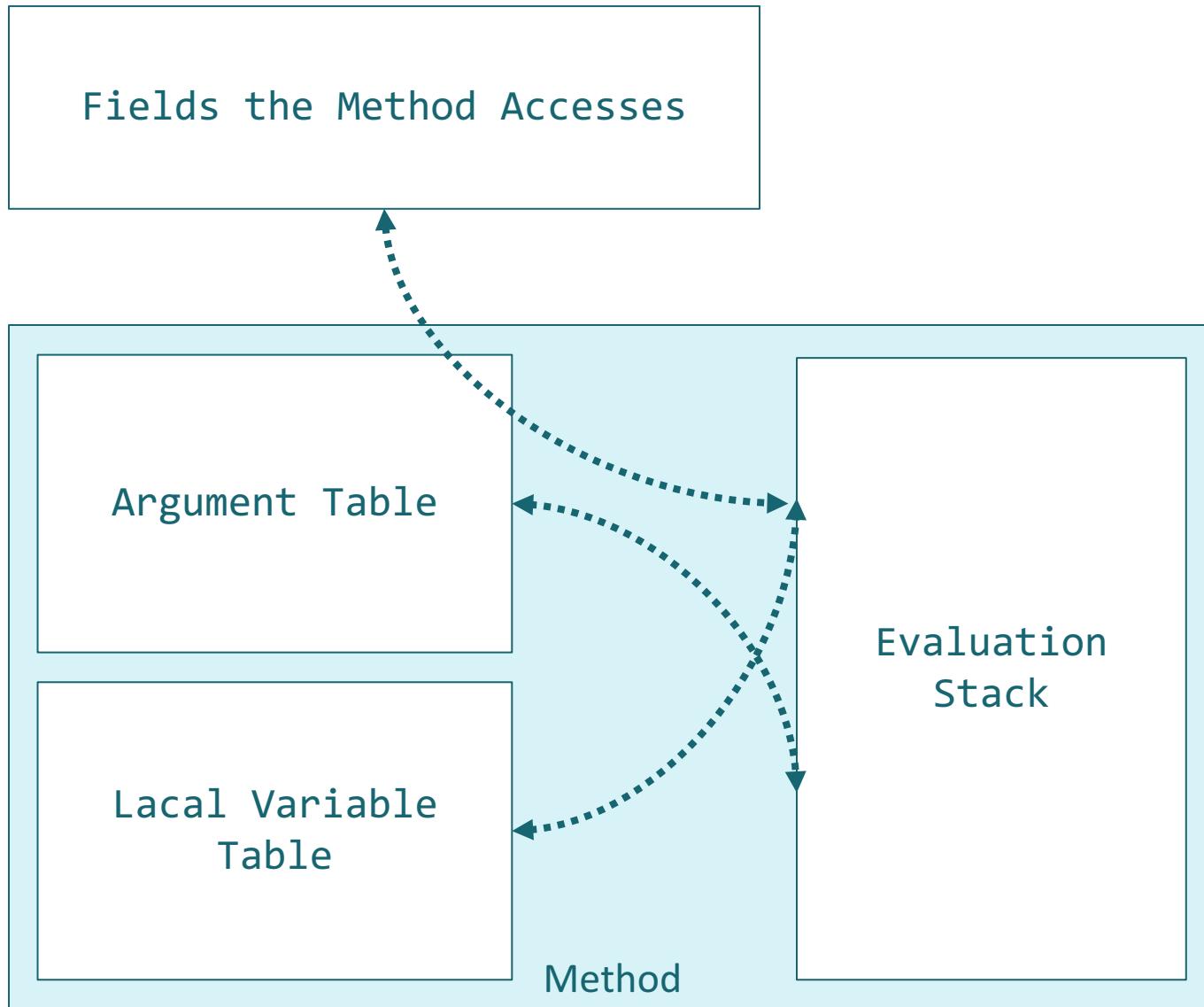
Вычислительный стек (The Evaluation Stack)

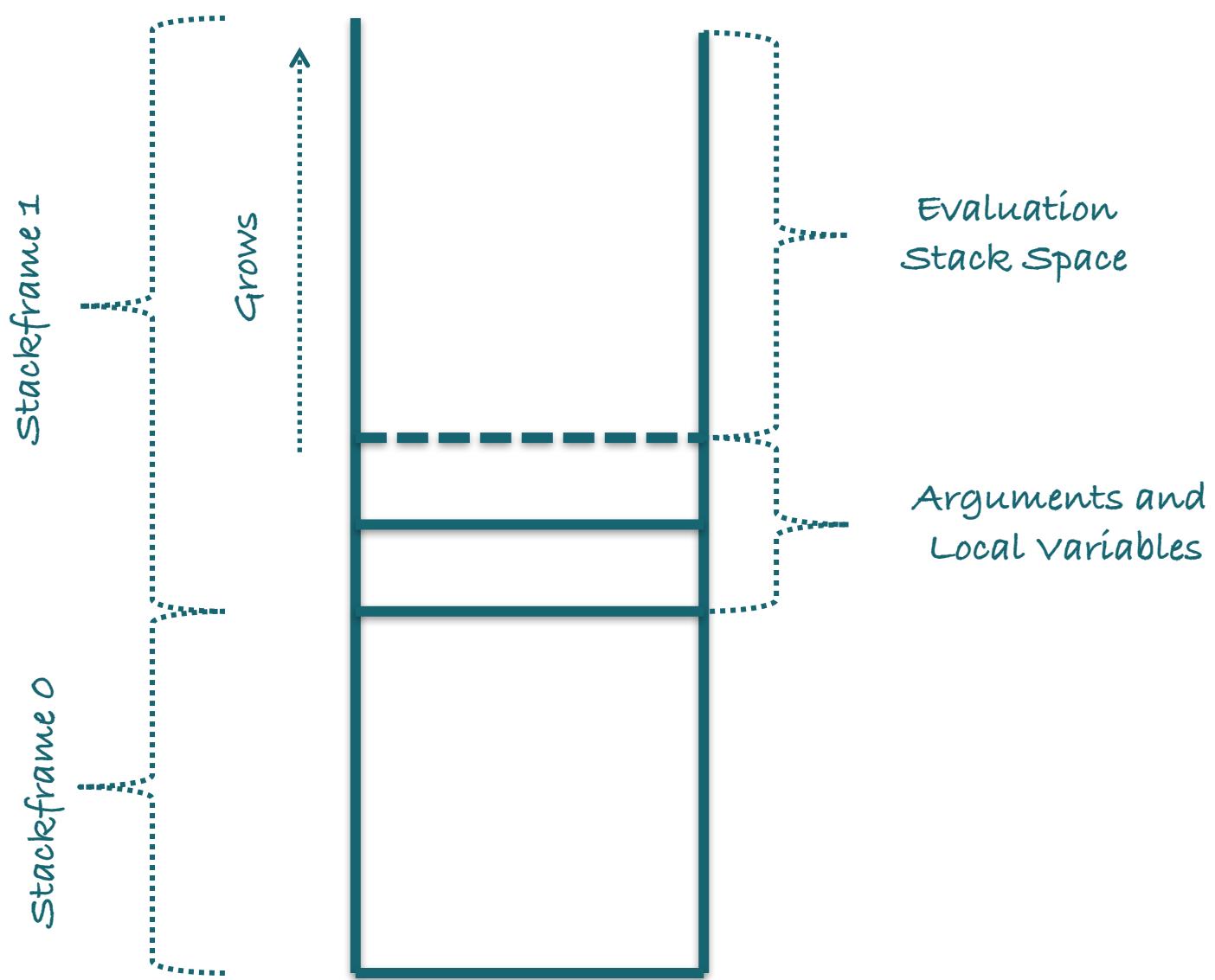
Вычислительный стек является ключевой структурой приложений MSIL. Это мост между приложением и ячейками памяти. Он похож на обычную стековый фрейм, но есть значительные отличия.

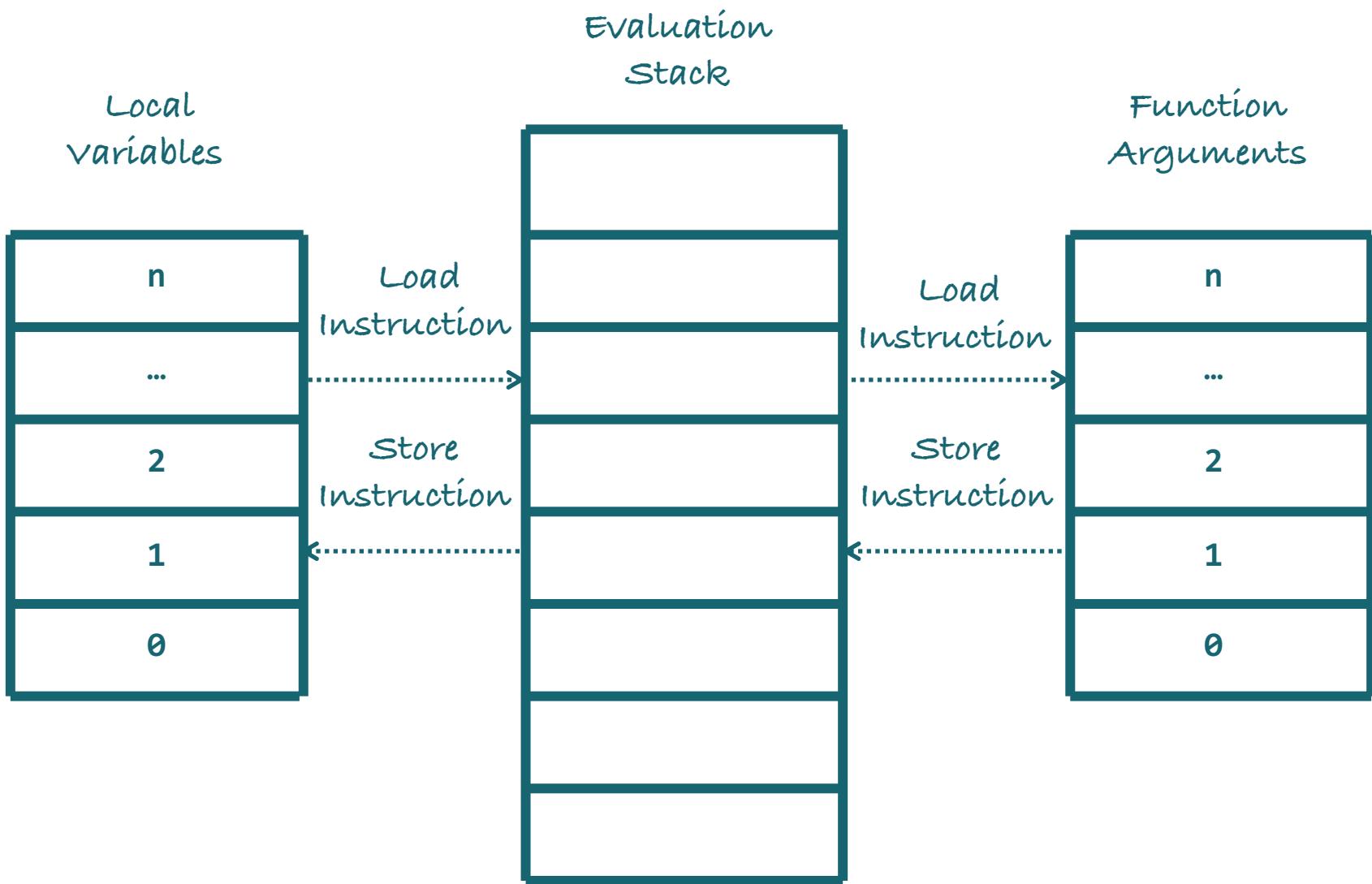
Вычислительный стек - это средство просмотра приложения, и его можно использовать для просмотра параметров функции, локальных переменных, временных объектов и т. д. Традиционно параметры функции и локальные переменные помещаются в стек. В .NET эта информация хранится в отдельных репозиториях, в которых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

Напрямую обратиться к этим репозиториям нельзя. Доступ к параметрам или локальным переменным требует перемещения данных из памяти в слоты вычислительного стека с помощью команды `load`. И наоборот, обновление содержимого локальной переменной или параметра новым содержимым в вычислительном стеке, требует использования используя команды `store`. Слоты в вычислительном стеке составляют 4 или 8 байтов.

Вычислительный стек (The Evaluation Stack)







Команды загрузки и выгрузки в MSIL

- `ldc<число>` – загрузка константы
- `ldstr<строка>` – загрузка строковой константы
- `ldsflda<поле>` – загрузка адреса статического поля
- `ldloca <#переменной>` – загрузка адреса локальной переменной
- `ldflda<поле>` – загрузка адреса поля объекта
- `ldind` – косвенная загрузка, берет адрес из стека и помещает на его место значение, размещенное по этому адресу

- `stind` берет из стека адрес значения и само значение и записывает значение по выбранному адресу
- команды `stloc`, `stfld`, `stsfld` эквивалентны командам `ldxxxx`

Арифметические команды MSIL

- Арифметические команды MSIL выполняют вычисления в стеке вычислений (evaluation stack)
- Вычислительные команды (`add`, `mul`, `sub`...) существуют в знаковом и беззнаковом (.u) вариантах, а также могут выполнять контроль за переполнением (.ovf)
- Кроме того, есть логические команды `and`, `or`, `xor` (только знаковые, без контроля переполнения) и операции сравнения

Переходы и вызовы в MSIL

- Переходы – стандартные (br, bne, beq и т.п.)
- Вызовы методов:
 - вызов статического метода (call)
 - вызов виртуального метода (callvirt)
- Если вызывается экземплярный (instance) метод, то объект, которому он принадлежит, должен быть первым параметром. Для callvirt этот параметр обязателен
- Возврат осуществляется командой ret.

Иные команды MSIL

- `box/unbox` – реализуют функциональность упаковки и распаковки значений
- `newobj` – команда для создания нового объекта
- `throw, rethrow, endfinally, endfilter, leave` – команды обработки исключений

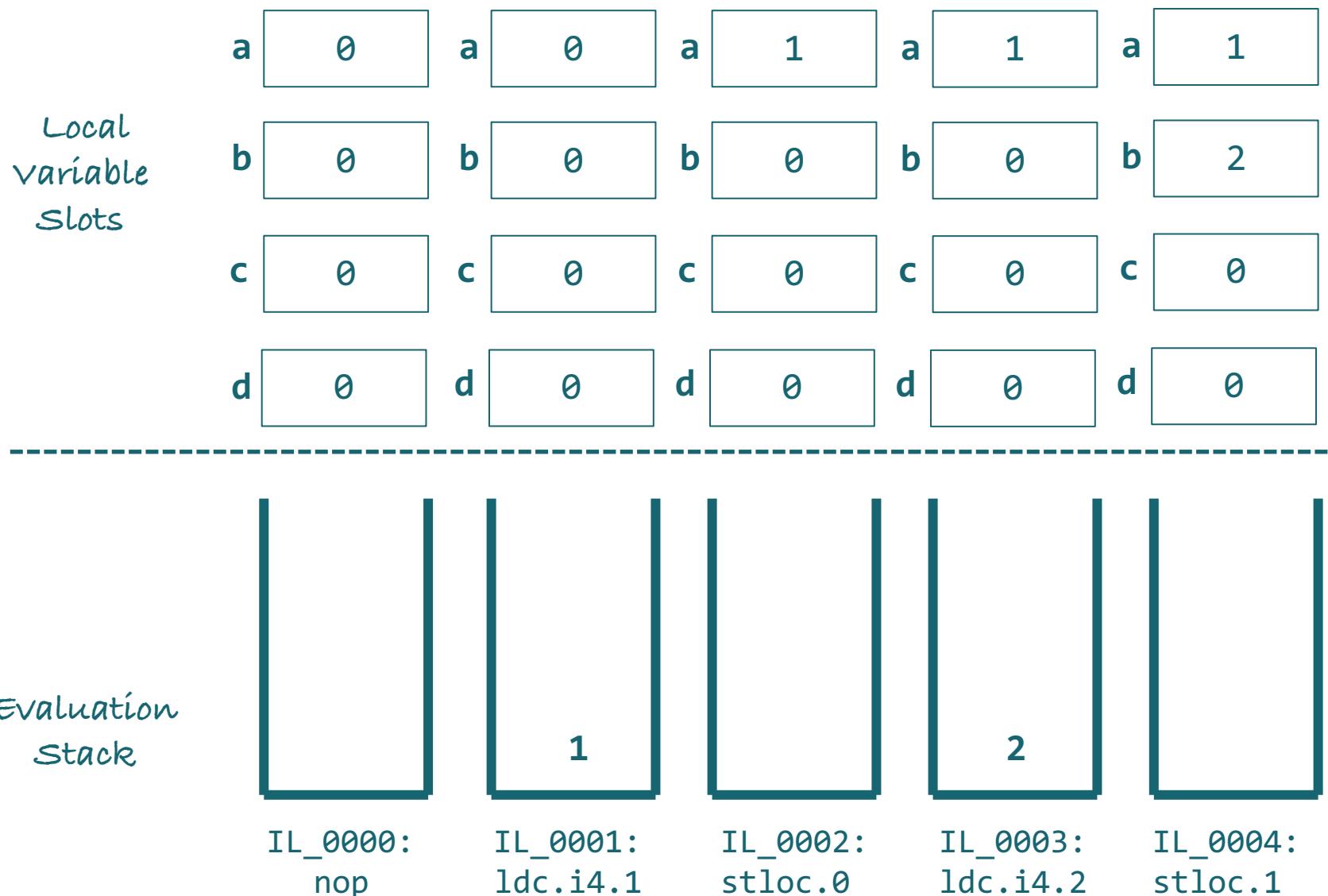
Стек вычислений

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

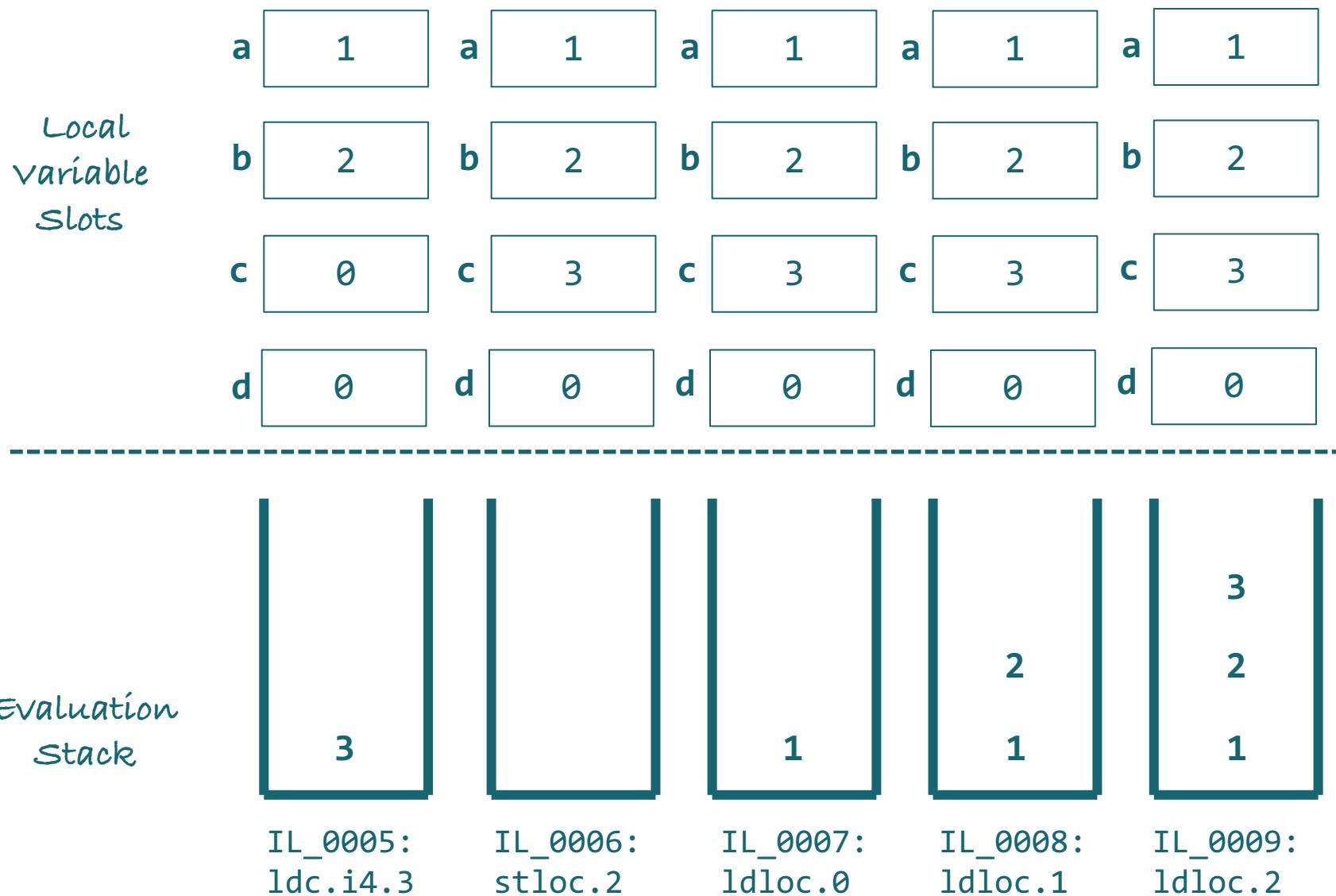
C# compiler

```
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
```

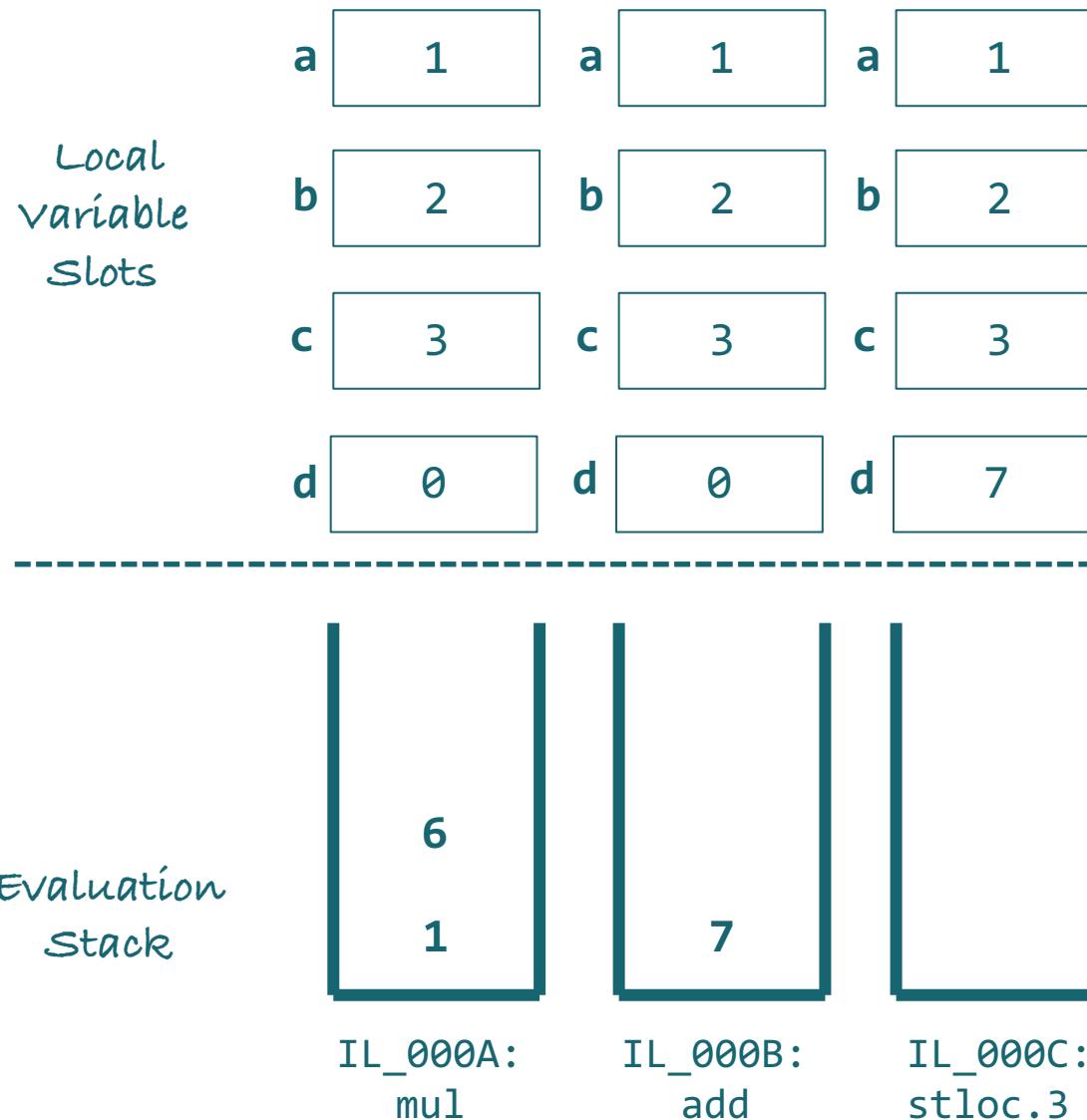
Стек вычислений



Стек вычислений



Стек вычислений



Стек вычислений

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



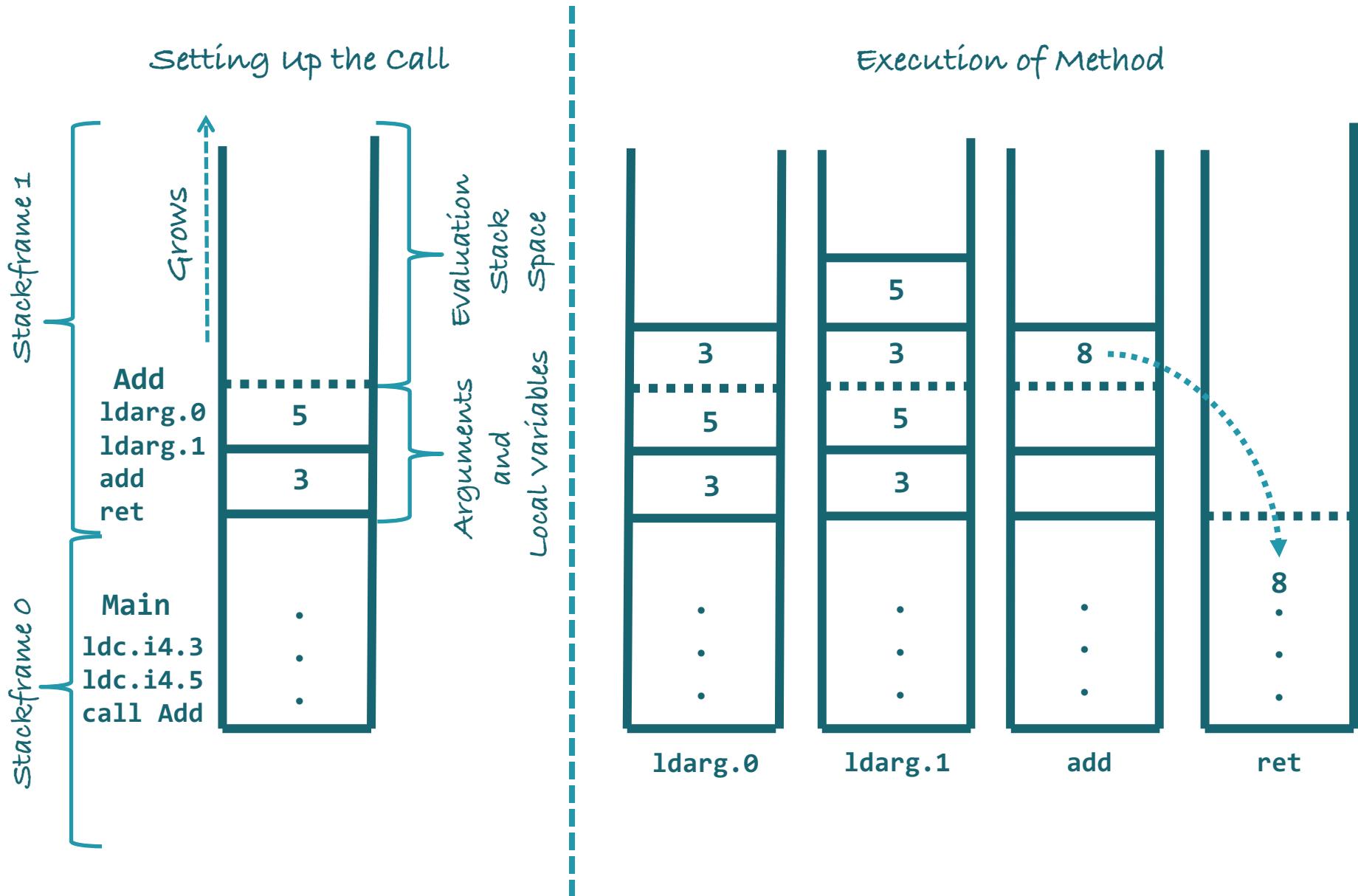
```
IL_0000:  nop  
IL_0001:  ldc.i4.1  
IL_0002:  stloc.0  
IL_0003:  ldloc.0  
IL_0004:  dup  
IL_0005:  ldc.i4.1  
IL_0006:  add  
IL_0007:  stloc.0  
IL_0008:  stloc.1  
IL_0009:  ret
```

```
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



Передача параметров по значению (pass by value)

```
void SomeMethod(int first, double second)  
{  
    ...  
}  
...  
SomeMethod(12, 3.14);
```

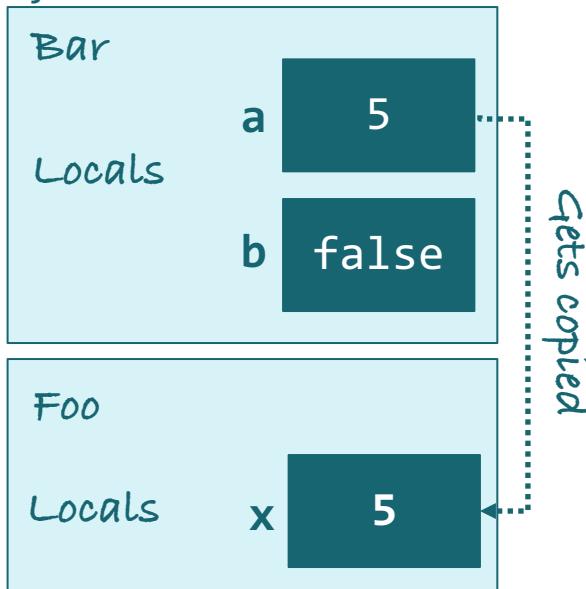
Тип параметра - параметр-значение - не имеет специальных модификаторов. Получает свой аргумент в месте вызова по значению.

Параметр-значение не имеет ничего общего с типами значениями или ссылочными типами, а является просто аспектом вызова метода!

Передача параметров по значению

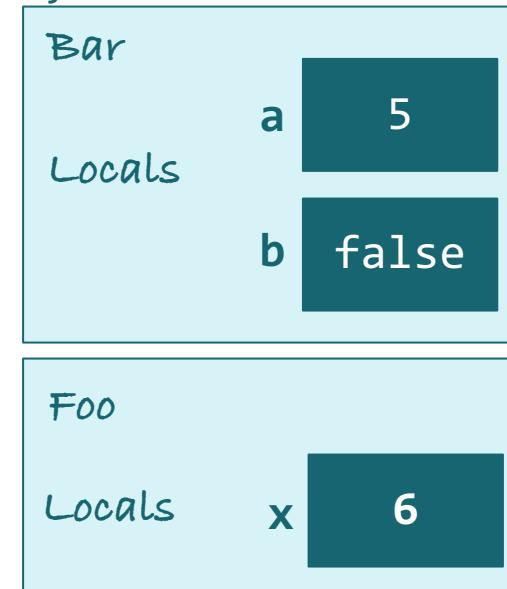
```
void Bar()    Тип значение  
{  
    int a = 5;  
    Foo(a);  
    bool b = a == 5;  
}
```

```
void Foo(int x)  
{  
    x++;  
}
```



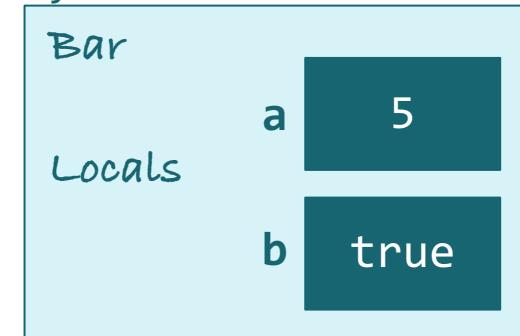
```
void Bar()  
{  
    int a = 5;  
    Foo(a);  
    bool b = a == 5;  
}
```

```
void Foo(int x)  
{  
    x++;  
}
```



```
void Bar()  
{  
    int a = 5;  
    Foo(a);  
    bool b = a == 5;  
}
```

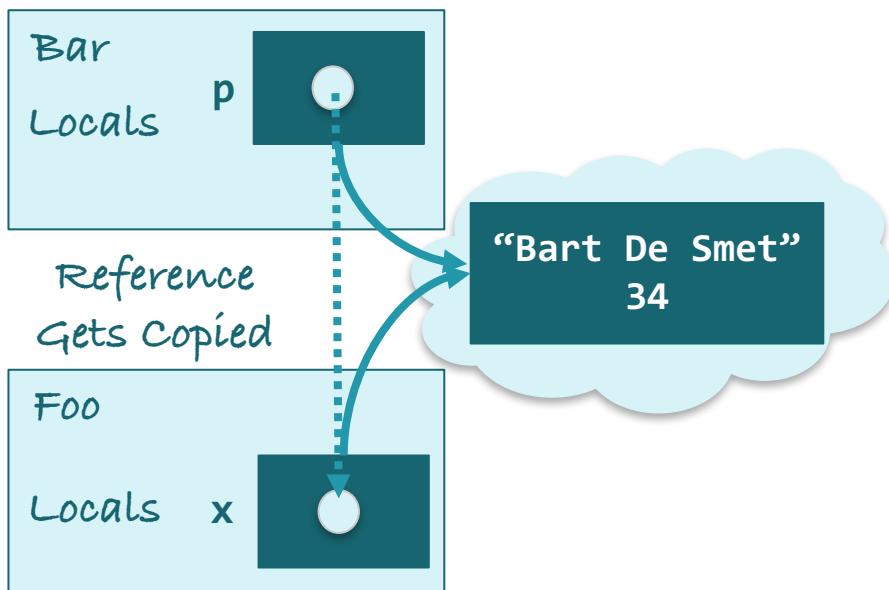
```
void Foo(int x)  
{  
    x++;  
}
```



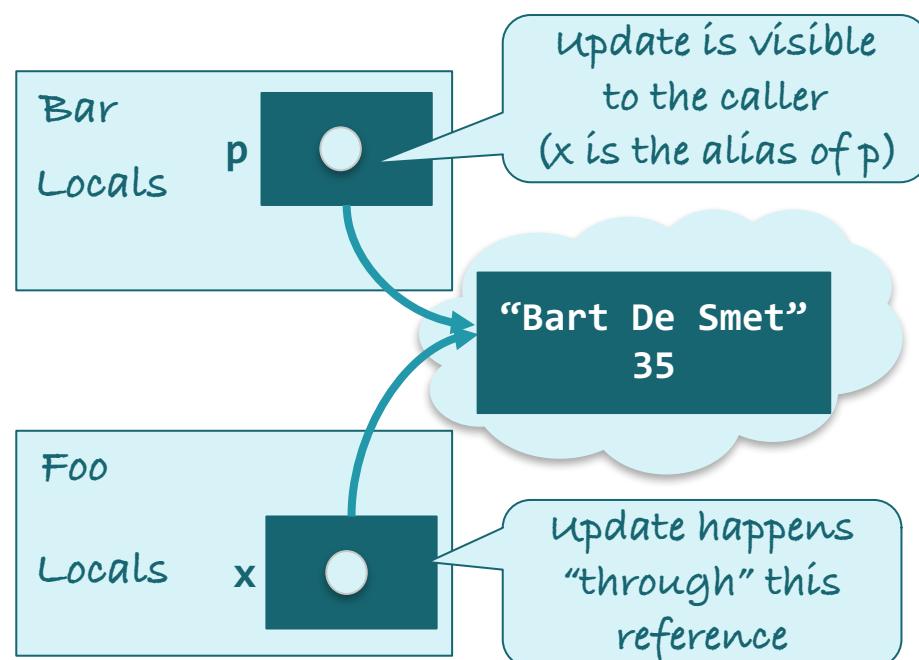
Передача параметров по значению

```
void Bar()
{
    Person p = ...;
    Foo(p);
    ...
}
void Foo(Person x)
{
    x.Age++;
    x = new Person();
}
```

Ссылочный тип (класс)



```
void Bar()
{
    Person p = ...;
    Foo(p);
    ...
}
void Foo(Person x)
{
    x.Age++;
    x = new Person();
}
```



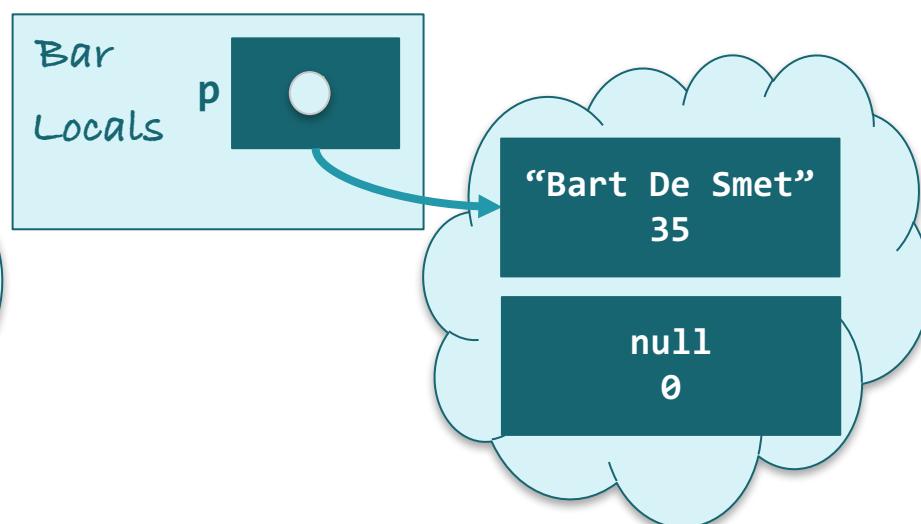
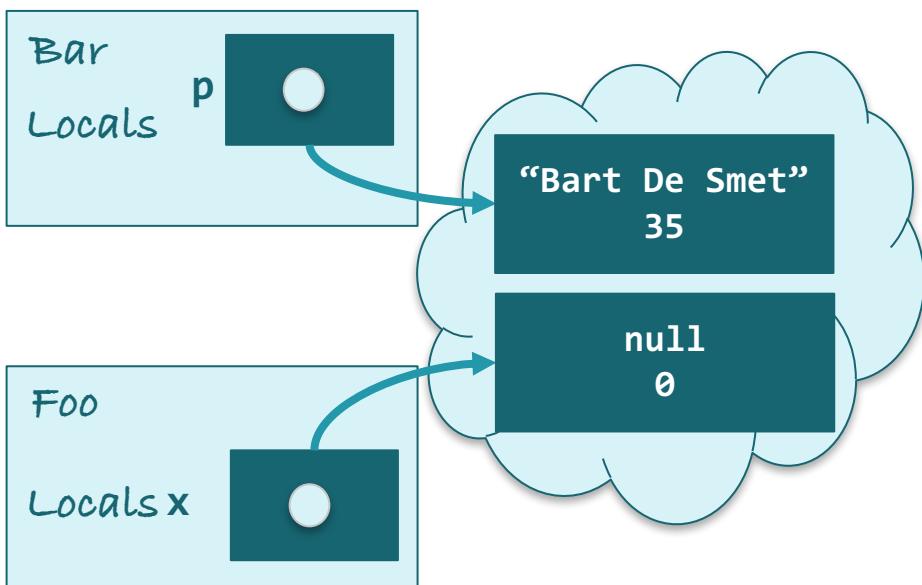
Передача параметров по значению

```
void Bar()
{
    Person p = ...;
    Foo(p);
    ...
}

void Foo(Person x)
{
    x.Age++;
    x = new Person();
}
```

```
void Bar()
{
    Person p = ...;
    Foo(p);
    ...
}

void Foo(Person x)
{
    x.Age++;
    x = new Person();
}
```



Передача параметров по значению

Поведение переменных значимого и ссылочного типов отличается при передаче параметров по значению в контексте возможности изменения связанных с этими типами экземпляров.

При использовании в качестве параметра-значения переменной ссылочного типа вызываемый код создает локальную «копию» этой переменной, которая, однако, указывает на тот же объект, что и ссылочная переменная вызывающего кода.

При использовании в качестве параметра-значения переменной значимого типа вызываемый код создает локальную «копию» этой переменной.

Любое изменение локальной переменной значимого невидимо для вызывающего кода. Однако изменения, сделанные с помощью локальной копии ссылочной переменной, будут видны вызывающему коду, поскольку будут приводить непосредственно к изменению объекта, на который указывает данная локальная ссылка.

Передача параметров по ссылке (pass by reference)

Изменения значений параметров, переданных по значению, не видимо для вызывающего кода, однако это ограничение могут обойти параметры, переданные по ссылке. Вместо того, чтобы создавать копию передаваемой переменной (независимо от ее типа - значимого или ссылочного типа), создается косвенная ссылка на локальную переменную, удерживаемую вызывающим кодом.

```
void SomeMethod(int first, double second, ref int refData)
{
    ...
    refData = 100;
    ...
}
    Присваивание начального
    значения обязательно
int value = 1;
SomeMethod(10, 101.1, ref value);
```

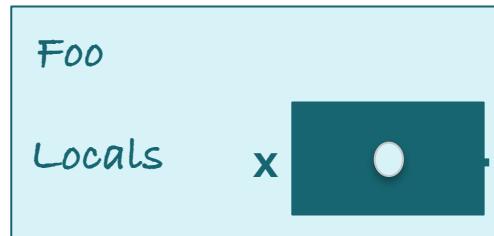
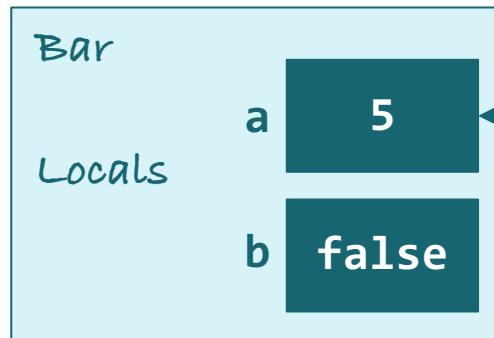
Использование ключевого слова `ref` является обязательным как при объявлении параметра, так и при передаче аргумента. Последнее требование введено, чтобы было ясно, что происходит при чтении кода вызова метода.

Всякий раз, когда аргумент передается по ссылке, можно ожидать, что переменная, которая была передана, будет изменена вызываемым кодом

Передача параметров по ссылке

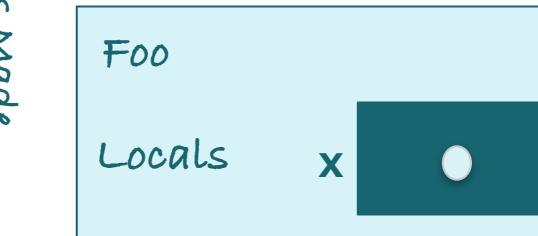
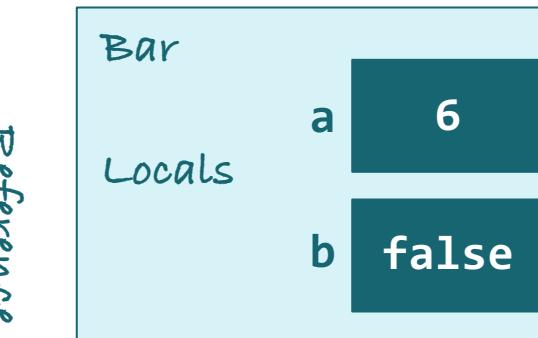
```
void Bar()    Тип значение  
{  
    int a = 5;  
    Foo(ref a);  
    bool b = a == 5;  
}
```

```
void Foo(ref int x)  
{  
    x++;  
}
```



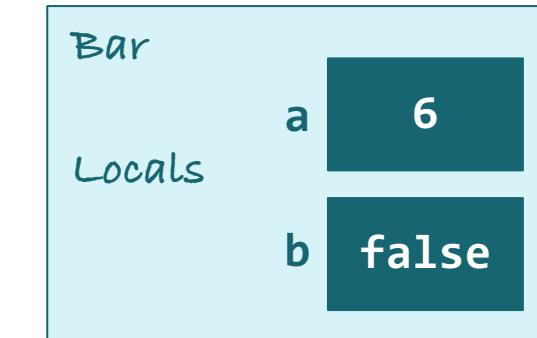
```
void Bar()  
{  
    int a = 5;  
    Foo(ref a);  
    bool b = a == 5;  
}
```

```
void Foo(ref int x)  
{  
    x++;  
}
```



```
void Bar()  
{  
    int a = 5;  
    Foo(ref a);  
    bool b = a == 5;  
}
```

```
void Foo(ref int x)  
{  
    x++;  
}
```



Передача параметров по ссылке

```
static void ChangeIt(ref int a)
{
    a = 42;
}
```

C# compiler

ChangeIt:

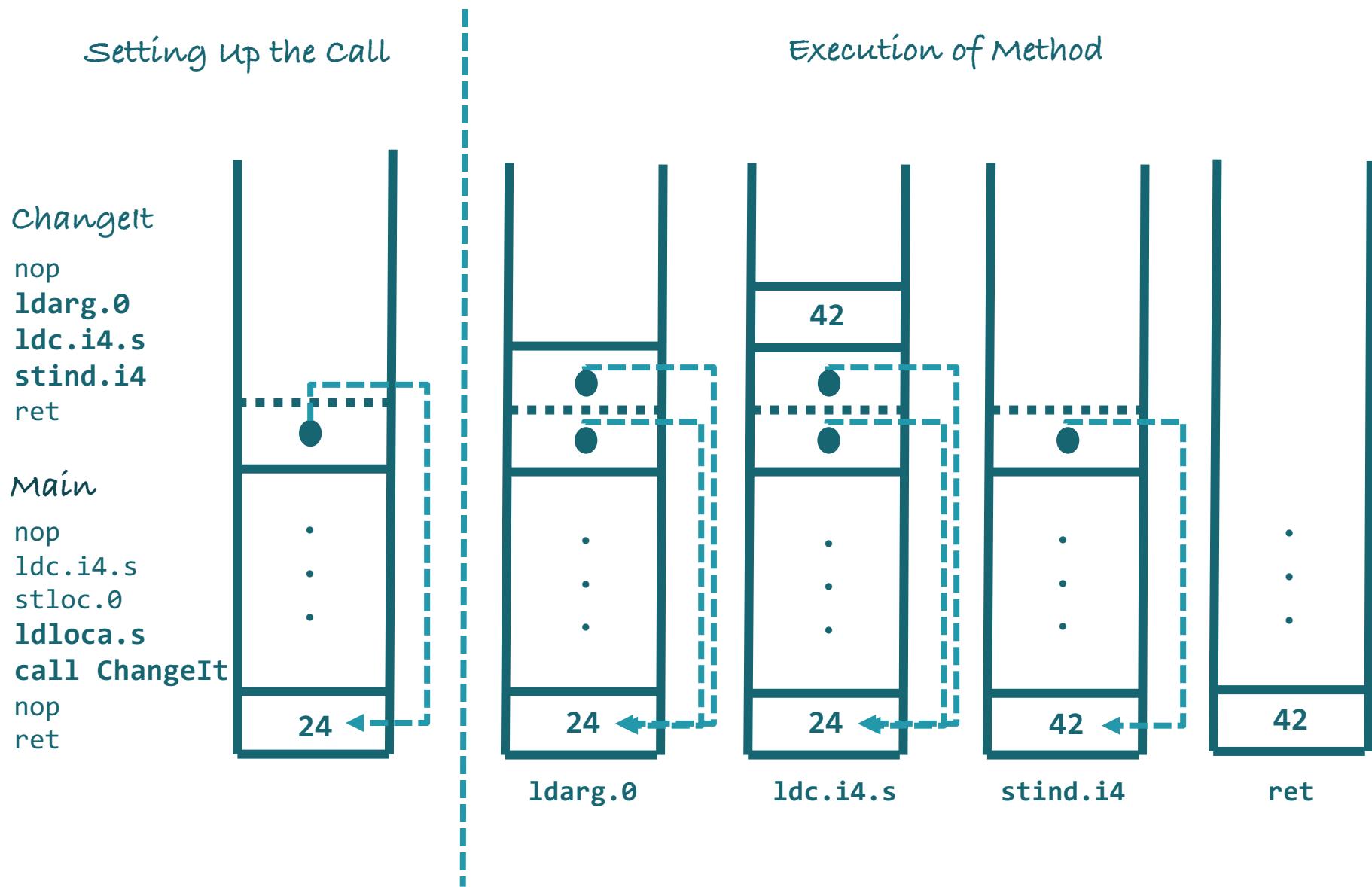
IL_0000:	nop	
IL_0001:	ldarg.0	
IL_0002:	ldc.i4.s	2A
IL_0004:	stind.i4	
IL_0005:	ret	

```
static void Main()
{
    int x = 24;
    ChangeIt(ref x);
}
```

C# compiler

IL_0000:	nop	
IL_0001:	ldc.i4.s	18
IL_0003:	stloc.0	// x
IL_0004:	ldloca.s	00 // x
IL_0006:	ChangeIt	
IL_000B:	nopcall	
IL_000C:	ret	

Передача параметров по ссылке



Передача параметров по ссылке

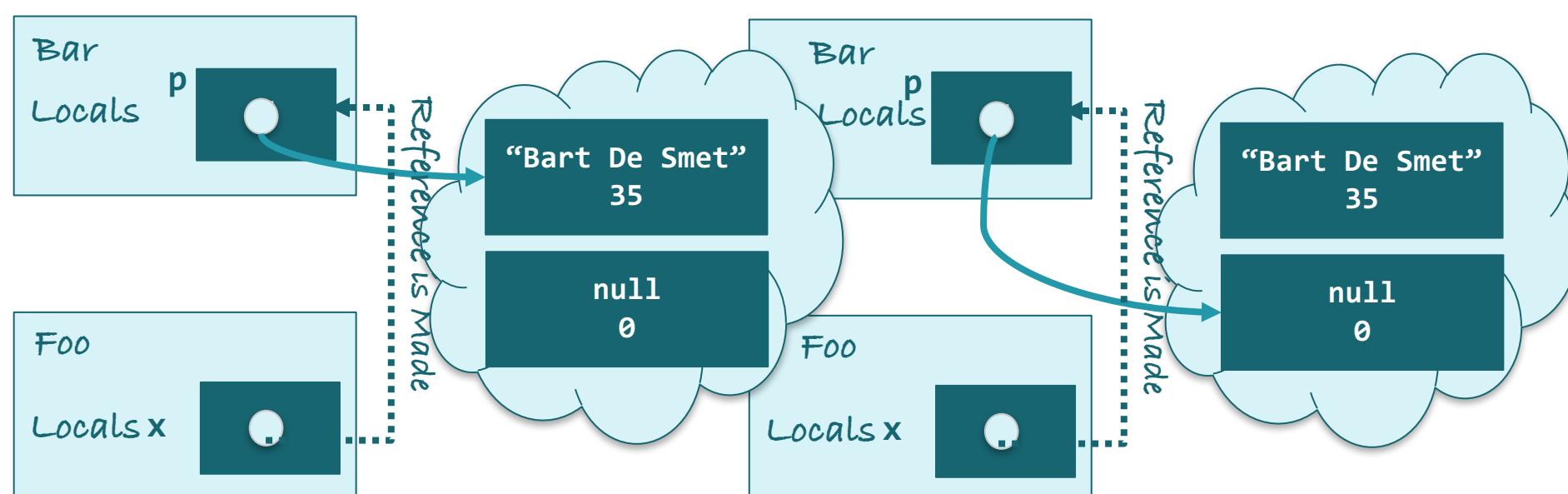
```
void Bar()
{
    Person p = ...;
    Foo(ref p);
    ...
}

void Foo(ref Person x)
{
    x = new Person();
}
```

ссылочный тип (класс)

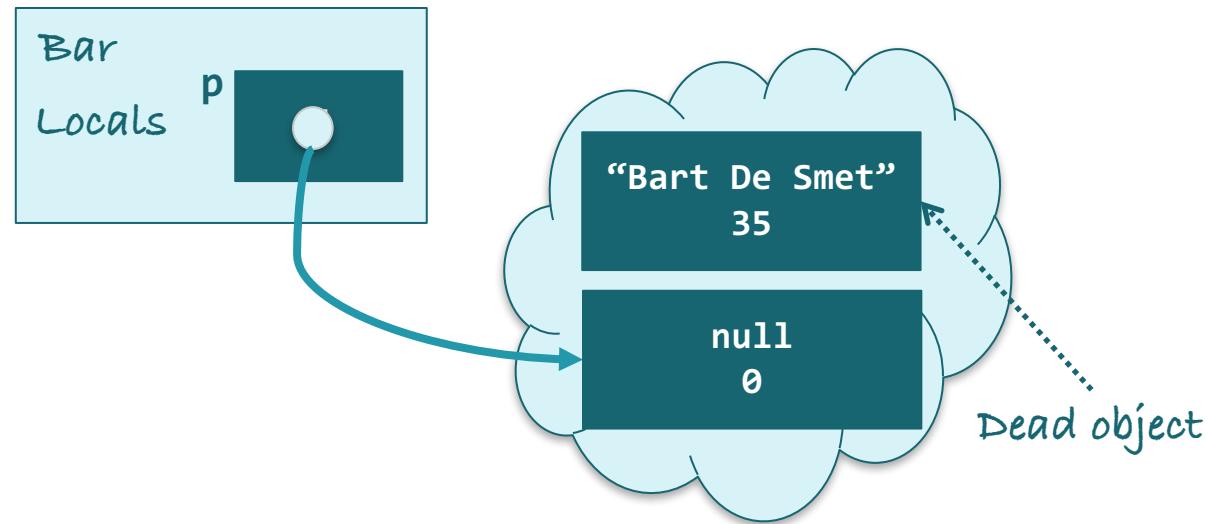
```
void Bar()
{
    Person p = ...;
    Foo(ref p);
    ...
}

void Foo(ref Person x)
{
    x = new Person();
}
```



Передача параметров по ссылке

```
void Bar()
{
    Person p = ...;
    Foo(ref p);
    ...
}
void Foo(ref Person x)
{
    x = new Person();
}
```



```
byte[] data = new byte[10];
Array.Resize(ref data, 20);
Console.WriteLine($"New array size is: {data.Length}");
```

Метод FCL `Array.Resize()` принимает параметр `data` типа `Array` по ссылке `ref`, значение которого изменяется в реализации метода таким образом, чтобы указывать на новый объект массива другой размерности

Новые возможности использования ref (C# 7.0)

```
public ref int Find (int number, int[] numbers)
{
    for (int i = 0; i < numbers.Length; i++)
    {
        if (numbers[i] == number)
        {
            return ref numbers[i]; <----- возвращает место хранения, а не значение
        }
    }
    throw new IndexOutOfRangeException ($"nameof (number) not found");
}
```

Использование ключевого слова `ref` возможно в продвинутых сценариях - с оператором `return` и с локальными переменными

```
int[] array = { 1, 15, -39, 0, 7, 14, -12 };
ref int place = ref Find (7, array);
place = 9; <----- заменяет 7 на 9 в массиве
```

Псевдоним расположения значения ≠ в массиве

Заменяет 7 на 9 в массиве

Новые возможности использования ref (C# 7.0)

```
static ref T ElementAt<T>(T[] array, int position)
{
    if (array == null)
    {
        throw new ArgumentNullException(nameof(array));
    }
    if (position < 0 || position >= array.Length)
    {
        throw new ArgumentOutOfRangeException(nameof(position));
    }
    return ref array[position];
}

int[] data = new int[10];
ref int value = ref ElementAt(data, 2);
value = 11;

int[] data = new int[10];
ElementAt(data, 2) = 11;

int[] data = new int[10];
int value = ElementAt(data, 2);
value = 11;
```

Передача параметров по ссылке. Выходные параметры

Тип возвращаемого значения используется для обозначения успеха или неудачи отработанного метода, в то время как выходной параметр будет содержать результат выполненного метода в случае его успешного завершения.

```
static bool TrySqrt(double input, out double root)
{
    root = 0.0;
    if (input < 0)
        return false;
    root = Math.Sqrt(input);
    return true;
}
```

являются дополнительной возможностью к возвращению значения из метода

```
int root; <
bool ok = TrySqrt(16, out root);
```

Присваивание начального значения не обязательно

Передача параметров по ссылке. Выходные параметры

```
Console.Write("Enter your age: ");
string input = Console.ReadLine();
int age;
if (!int.TryParse(input, out age))
    // Print error message, maybe let the user retry
else
    // We got a valid age
```

Общий шаблон для различных типов ВСЛ, например, таких как числовые типы значений, которые имеют метод TryParse

Отличие между TryParse и Parse заключается в том, что происходит при передаче недопустимого ввода метода. Parse в этом случае генерирует исключение, а TryParse - нет. Поскольку исключения являются дорогостоящими, а некорректная строка не является исключительным случаем при работе с пользовательским вводом, TryParse имеет гораздо больший смысл, поскольку сообщает об успешном завершении или неудаче с помощью логического значения, которое можно легко проверить.

Передача параметров по ссылке. Выходные параметры

НЕ ИСПОЛЬЗОВАТЬ СЛИШКОМ ЧАСТО!

Использование выходных параметров может показаться удобным для возвращения нескольких результатов из метода, но довольно скользким в месте вызова метода, потому что предполагает наличия предварительно объявленной локальная переменная и ключевого слова `out`.

В случае необходимости возвращения из метода большого количества данных, рекомендуется рассмотреть возможность их объединения в некоторый специализированный тип возвращаемого значения (новый класс или структуру). Кроме того, начиная с .NET 4, для этой цели можно использовать тип `System.Tuple`.

Передача параметров по ссылке. Выходные параметры (C# 7.0)

```
if (int.TryParse ("123", out int result),
```

Результат объявлен
встроенным

```
Console.WriteLine(result);
```

Объявленная таким образом
локальная переменная находится
в области видимости кода

```
int.TryParse ("234", out var result);
```

Out переменные могут быть
объявлены неявно (var)

Передача параметров по ссылке. Выходные параметры (C# 7.0)

```
void Foo (out int p1, out string p2, out bool p3, out char p4)
{
    p1 = 42;
    p2 = "fourty two";
    p3 = true;
    p4 = 'x';
}
```

```
string numberString = Util.ReadLine ("Enter a number");
```

Отсутствует имя
out аргумента

```
if (int.TryParse (numberString, out _))  
    Console.WriteLine("Valid number");
```

ПОДЧЕРКИВАНИЕ МОЖНО
ИСПОЛЬЗОВАТЬ НЕСКОЛЬКО
РАЗ В ОДНОМ МЕТОДЕ

```
else  
    Console.WriteLine("Invalid number");
```

```
Foo (out int interesting, out _, out _, out _);
```

Выходные параметры или кортежи (C# 7.0)

- Выходные параметры: использование не очень удобно (даже с улучшениями, C# 7.0), не работают с асинхронными методами;
- System.Tuple<...> : очевиден для использования, требует создания объекта кортежа;
- Пользовательский тип (класс или структура): накладные расходы для написания типа, целью которого является лишь временное группирование нескольких значений;

C # 7.0 добавляет типы кортежей и кортежные литералы:

```
(string, string, string) LookupName(long id) // tuple return type
{
    ...
    ... // retrieve first, middle and last from data storage
    return (first, middle, last); // tuple literal
}
```

Выходные параметры или кортежи (C# 7.0)

```
Tuple<string, int> tupleBefore = new Tuple<string, int>("three", 3);  
  
var tuple = ("three", 3); <----- Кортежи можно создавать с  
                           неявной типизацией  
  
(string, int) tuple2 = tuple; <----- С явной типизацией  
  
var namedTuple = (word:"three", number:3);  
Console.WriteLine(namedTuple.number); <----- Можно именовать  
                                         поля кортежа  
Console.WriteLine(namedTuple.word);  
  
Именованные кортежи компилируются  
аналогично неименованным с (Item1,  
Item2), однако в этом случае можно  
ссыльаться на имена в коде
```

Опциональные параметры

```
public FileStream (string path, FileMode mode, FileAccess access,  
                  FileShare share, int bufferSize, FileOptions options) {...}
```

Можно реализовать удобные перегруженные версии методов для предоставления возможности вызова методов со значениями по умолчанию.

```
public FileStream (string path, FileMode mode)
```

```
    : this (path, mode, FileAccess.ReadWrite) {...}
```

```
public FileStream (string path, FileMode mode, FileAccess access)
```

```
    : this (path, mode, access, FileShare.Read) {...}
```

```
public FileStream (string path, FileMode mode, FileAccess access, FileShare share)
```

```
    : this (path, mode, access, share, 0x1000){...}
```

```
public FileStream (string path, FileMode mode, FileAccess access,
```

```
                  FileShare share, int bufferSize)
```

```
    : this (path, mode, access, share, bufferSize, FileOptions.None) {...}
```

```
public FileStream (string path, FileMode mode, FileAccess access,
```

```
                  FileShare share, int bufferSize, FileOptions options) {...}
```

Опциональные параметры

Вместо реализации большого количества перегрузок, можно использовать опциональные параметры для указания значений по умолчанию, которые устанавливаются при вызове метода, когда параметр опущен

```
public FileStream (
    string path,
    FileMode mode,           Обязательные параметры
    FileAccess access = FileAccess.ReadWrite,
    FileShare share = FileShare.Read,
    int bufferSize = 0x1000,
    FileOptions options = FileOptions.None
)
{
    // Actual code here
    // ...
}
```

Опциональные параметры должны идти после всех обязательных параметров

Значение, присваиваемое необязательному параметру, должно быть известно во время компиляции и не может вычисляться во время выполнения.

Допустимыми значениями для опциональных параметров являются числовые значения, логические значения, строки, перечисления и ссылка null.

Опциональные параметры. Бинарная совместимость

SomeLibrary.dll
v 1.0.0.0

```
public class Foo()
{
    public void Do(int x){}
}
```



ClientApplication.exe

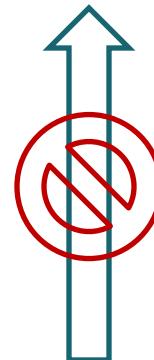
```
new Foo().Do(100);
```

Опциональные параметры. Бинарная совместимость

SomeLibrary.dll

v 1.0.0.0

```
public class Foo()
{
    public void Do(int x, int y = 0){}
}
```



ClientApplication.exe

```
new Foo().Do(100);
```

Опциональные параметры и именованные аргументы

```
public FileStream (  
    string path,  
    FileMode mode,  
    FileAccess access = FileAccess.ReadWrite,  
    FileShare share = FileShare.Read,  
    int bufferSize = 0x1000,  
    FileOptions options = FileOptions.None  
)  
{  
    // Actual code here  
    // ...  
}
```

Именованный аргумент

```
new FileStream ("temp.txt", FileMode.Create);  
new FileStream ("temp.txt", FileMode.Create, FileAccess.Write);  
new FileStream ("temp.txt", FileMode.Create, FileAccess.Write,  
    FileShare.Inheritable, 0x5000);  
new FileStream ("temp.txt", FileMode.Create, FileAccess.Write,  
    FileShare.Inheritable, 0x5000, FileOptions.Asynchronous);  
  
new FileStream ("temp.txt", FileMode.Create, options:  
FileOptions.Asynchronous);
```

Массив параметров

```
int Sum(int one, int two) => one + two;
```

```
int Sum(int one, int two, int three) => one + two + three;
```

```
int Sum(int one, int two, int three, int four) => one + two + three + four;
```

etc.

Некоторым методам может быть полезно при вызове иметь неограниченное количество переданных им аргументов

```
int Sum(int[] data)
{
    ...
}
```

```
int[] data = new int[4];
myData[0] = 99;
myData[1] = 2;
myData[2] = 55;
myData[3] = -26;
int sum = obj.Sum(data);
```

или

```
int Sum(params int[] data)
{
    ...
}
```

```
int sum = obj.Sum(99, 2, 55, -26);
```

Массив параметров

Параметры некоторого типа массива, которые появляются в конце списка формальных параметров и имеют префиксом ключевое слово `params`.

```
int Sum(params int[] data)
{
    ...
}
```

```
int sum = obj.Sum(9, 2, 55, -26);
```

Вызывающий код просто рассматривает их как массивы, имея при этом возможность опустить код создания массива и просто передавать аргументы, разделенные запятыми, как если бы они соответствовали отдельным параметрам

```
int sum = obj.Sum(9, 2, 55, -26);
```

Вызов в основном преобразуется в вызов, где массив создается со стороны вызывающего кода

```
int sum = obj.Sum(new int[]{9, 2, 55, -26});
```

Массив параметров

Определение методов с небольшим количеством параметром в дополнение к методу, который принимает массив параметров, является обычной практикой для сокращения накладных расходов, по созданию объектов массива в куче

```
int Sum(params int[] data) { ... }
```

```
int Sum(int one, int two) => one + two;
```

```
int Sum(int one, int two, int three) => one + two + three;
```

Если существует перегрузка метода, соответствующая указанному типу и количеству параметров, она будет вызываться предпочтительнее, чем версия метода, принимающего массив параметров

```
Console.WriteLine(string format, object arg0);
```

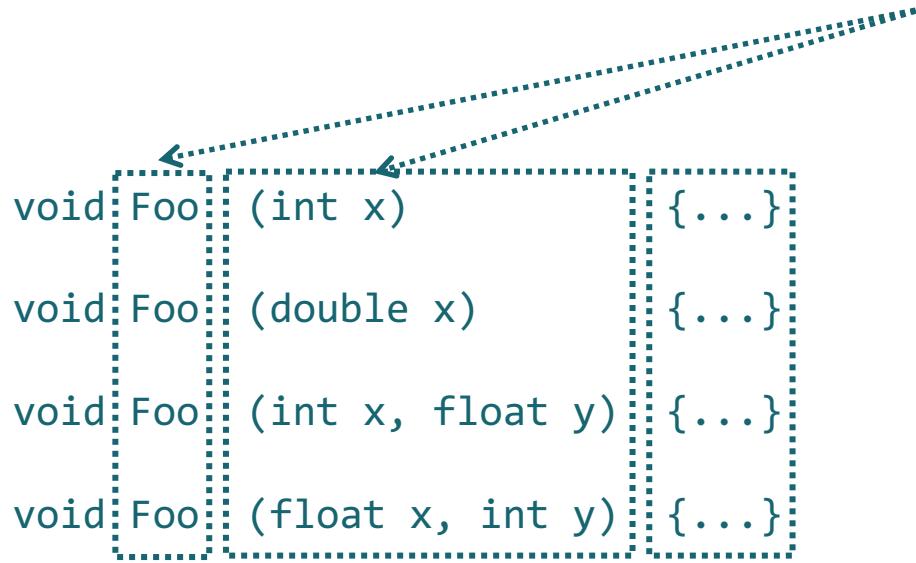
```
Console.WriteLine(string format, object arg0, object artg1);
```

```
Console.WriteLine(string format, object arg0, object artg1, object arg2);
```

```
Console.WriteLine(string format, object arg0, object artg1, object arg2,  
object arg3);
```

```
Console.WriteLine(string format, params object[] arg);
```

Перегрузка методов



Перегруженные методы

- имеют **одинаковое имя**
- имеют **уникальную сигнатуру**
- имеют **одну семантику**

При вызове метода компилятор определяет версию метода, который должен быть вызван.

Разрешение перегрузки (**overload resolution**) представляет собой механизм времени компиляции для выбора наилучшего функционального элемента из набора возможных функциональных элементов для вызова при заданном списке аргументов.

Перегрузка методов

```
void Foo (int x) { ... }  
float Foo (int x) { ... }
```

Методы не являются перегруженными, поскольку тип возвращаемого значения не является частью сигнатуры метода

```
int Foo (int[] x) { ... }  
ref int Foo (int[] x) { ... }
```

Методы не являются перегруженными, поскольку модификатор params не является частью сигнатуры метода

```
void Foo (int[] x) { ... }  
void Foo (params int[] x) { ... }
```

Методы не являются перегруженными, поскольку ключевые слова out / ref не являются частью сигнатуры метода

```
void Foo (ref int x) { ... }  
void Foo (out int x) { ... }
```

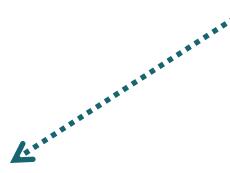
Локальные функции. C# 7.0

```
public int Fibonacci (int x)
{
    return Fib (x).current;
}

(int current, int previous) Fib (int i)
{
    if (i == 0) return (1, 0);
    var (p, pp) = Fib (i - 1);
    return (p + pp, p);
}

Console.WriteLine(Fibonacci(10));
```

Локальная функция



Статические методы

Статические методы, как правило, используются в классах для выполнения атомарных операций, не полагающихся на данные экземпляра

```
class Sales
{
    public static double GetMonthlySalesTax(double monthlyProfit)
    {...}
}
```

При объявлении метода
следует использовать
модификатор `static`

Статические методы могут использовать только данные,
хранящиеся в статических полях и данные, которые
передаются в качестве параметров в сигнатуру метода

```
class Sales
{
    private static double salesTaxPercentage = 20;

    public static double GetMonthlySalesTax(double monthlyProfit)
    {
        return (salesTaxPercentage * monthlyProfit) / 100;
    }
}
```

↓

Статические классы

При разработке служебного класса, содержащего только статические члены, можно объявить сам класс как статический (не относится к структурным типам!)

```
public static class SimpleStaticClass
{
    public static void SimpleStaticMethod(...){...}
}
```

- Создание экземпляра невозможно
- Содержит только статические члены.
- Является запечатанным.
- Не может содержать конструкторы экземпляров.
- Не наследует интерфейсов
- Не может быть членом-полем, параметром метода или локальной переменной

```
using static SimpleStaticClass;
...
SimpleStaticMethod();
```

vs SimpleStaticClass.SimpleStaticMethod();

C# 7.0

Статические конструкторы

Типы могут содержать статические конструкторы (конструкторы классов или инициализаторы типов)

```
class SimpleClass
{
    static readonly long baseline;
    static SimpleClass() => baseline = DateTime.Now.Ticks;
}
```

Статические конструкторы

- Статический конструктор не имеет модификаторов доступа и списка формальных параметров.
- Статический конструктор вызывается автоматически не более одного раза в домене приложения для инициализации типа перед созданием первого экземпляра типа или ссылке на какие-либо статические члены.
- Статический конструктор нельзя вызвать напрямую.
- Пользователь не управляет временем, в течение которого статический конструктор выполняется в программе.
- Типичным использованием статических конструкторов является случай, когда класс использует лог-файл и конструктор применяется для добавления записей в этот файл.
- Статические конструкторы также полезны при создании классов-оболочек для неуправляемого кода, когда конструктор может вызвать метод LoadLibrary.
- Если статический конструктор инициирует исключение, среда выполнения не вызывает его во второй раз, и тип остается неинициализированным на время существования домена приложения, в котором выполняется программа.

Шаблон Singleton

Класс, реализующий шаблон Singleton :

- гарантирует, что можно создать только один его экземпляр
- и предоставляет точку доступа для получения этого экземпляра
- Введение отладочной информации
- Реализация сессий
- Кэш приложения
- Менеджер печати
- Доступ к аппаратному обеспечению

Нередко используется вместе с другими шаблонами (Абстрактной фабрикой, Строителем и Прототипом) для обеспечения уникальности их экземпляра

Негативные последствия использования Singleton являются проявлением его «глобализации»

- Многие части приложения становятся зависимы от него и, косвенно, друг от друга. Это усложняет внесение изменений в дальнейшем. Облегчить ситуацию можно используя Dependency Injection
- Приложение становится сложнее тестировать, т.к. данные, полученные от Singleton, могут быть созданы в другом модуле

Шаблон Singleton

Реализация шаблона в общем виде

- объявляем только закрытый конструктор, чтобы запретить создание экземпляров извне;
- в закрытом поле размещаем единственный экземпляр класса;
- предоставляем доступ к нему через свойство, открытое только для чтения;
- клиентский код использует это свойство для получения общего экземпляра класса.

Шаблон Singleton

```
public class DataCache
{
    private static DataCache instance; ← единственный
                                            экземпляр класса

    public static DataCache Instance { get; } ← доступ к единственному
                                                экземпляру через свойство,
                                                открытое только для чтения

    private DataCache() { } ← приватный конструктор, чтобы
                            запретить создание экземпляров
                            извне

    DataCache newCache = new DataCache(); ←
    DataCache sameCache = DataCache.Instance; ←
```

• Реализация должна быть потокобезопасной
• Реализация должна быть «отложенной» (lazy)

клиентский код
использует свойство
только для чтения
для получения общего
экземпляра класса.

Шаблон Singleton. Не потокобезопасный вариант

```
public sealed class Singleton
{
    private static Singleton instance;

    private Singleton() { }

    public static Singleton Instance
    {
        get
        {
            if (instance == null)
            {
                instance = new Singleton();
            }
            return instance;
        }
    }
}
```

Шаблон Singleton. Потокобезопасный вариант

```
public sealed class Singleton
{
    private static Singleton instance;
    private static readonly object padlock = new object();

    private Singleton() { }

    public static Singleton Instance
    {
        get
        {
            lock (padlock)
            {
                if (instance == null)
                {
                    instance = new Singleton();
                }

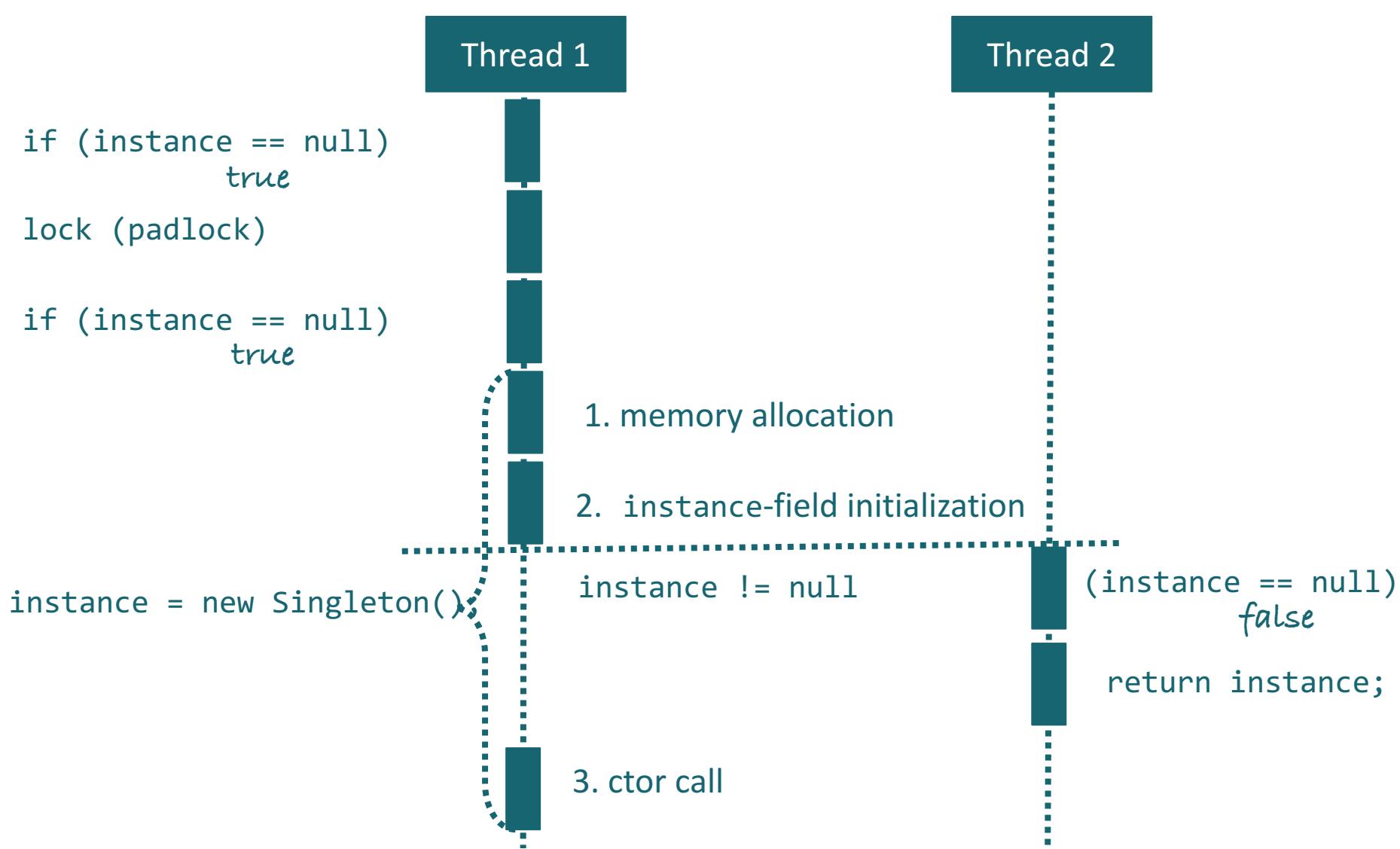
                return instance;
            }
        }
    }
}
```

Шаблон Singleton. Вариант double-check locking

```
public sealed class Singleton
{
    private static volatile Singleton instance = null;
    private static readonly object padlock = new object();
    private Singleton() { }

    public static Singleton Instance
    {
        get
        {
            if (instance == null)
            {
                lock (padlock)
                {
                    if (instance == null)
                    {
                        instance = new Singleton();
                    }
                }
            }
            return instance;
        }
    }
}
```

Шаблон Singleton



Шаблон Singleton на основе статического конструктора

```
public sealed class Singleton
{
    private static readonly Singleton instance;

    private static Singleton() => instance = new Singleton();

    private Singleton() { }

    public static Singleton Instance
    {
        get => instance;
    }
}
```

Добавление явного
статического конструктора
приказывает
компилятору не помечать
тип атрибутом `beforefieldinit`,
обеспечивая «почти
ленивую» инициализацию

Шаблон Singleton на основе статического конструктора

```
public sealed class Singleton
{
    private Singleton() { }

    public static Singleton Instance
    {
        get => Nested.instance;
    }

    private class Nested
    {
        internal static readonly Singleton instance = new Singleton();
    }
}
```

вложенный класс делает
реализацию полностью
«ленивой»

Шаблон Singleton на основе статического конструктора

```
public sealed class Singleton
{
    private Singleton() { }

    public static Singleton Instance
    {
        get => Nested.instance;
    }

    private class Nested
    {
        internal static readonly Singleton instance = new Singleton();
    }
}
```

вложенный класс делает
реализацию полностью
«ленивой»

Достоинства – простота реализации.

Недостатки – особенности генерации исключений; возможные проблемы с «ленивостью» (без вложенного класса); проблемы с временем инициализации при отсутствии статического конструктора

Шаблон Singleton на класса Lazy .NET 4.0

```
public sealed class Singleton
{
    private static readonly Lazy<Singleton> instance =
        new Lazy<Singleton>(() => new Singleton());

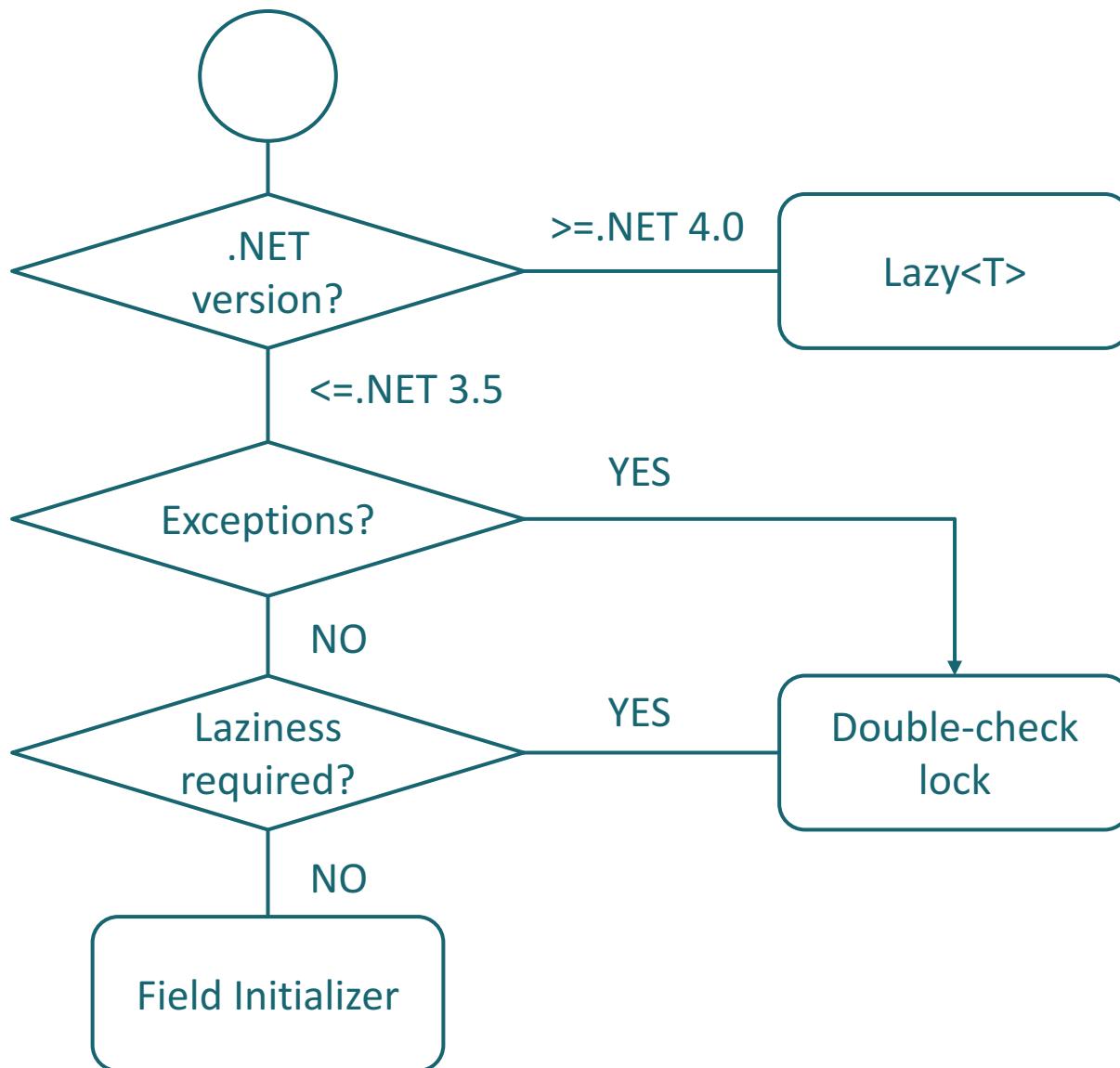
    public static Singleton Instance { get => instance.Value; }

    private Singleton(){}
}
```

Достоинства – простота реализации; потокобезопасность; «ленивость».

Недостатки – доступна начиная с версии .NET 4.0.

Статические конструкторы, использование, шаблон Singleton



Статический класс и Singleton

	Singleton	Статический класс
Количество точек доступа	Одна (и только одна) точка доступа – статическое поле Instance	N (зависит от количества публичных членов класса и методов)
Наследование классов	Возможно	Невозможно
Наследование интерфейсов	Возможно, без ограничений	Невозможно
Возможность передачи в качестве параметров	Возможно	Невозможно
Контроль времени жизни объекта	Возможно, например, отложенная инициализация	Невозможно
Использование абстрактной фабрики для создания экземпляра класса	Возможно	Невозможно
Сериализация	Возможно	Невозможно

Методы расширения

Добавление методов расширения к существующему классу возможно:

- без доступа к коду существующего класса
- без изменения или перекомпиляции существующих классов
- без получения нового типа из существующего класса

```
namespace Extensions
{
    public static class IntegerExtension
    {
        public static int NextRand(this int seed, int maxValue)
        {
            Random randomNumberGenerator = new Random(seed);
            return randomNumberGenerator.Next(maxValue);
        }
    }
}

using Extensions;
...
int i = 8;
int j = i.NextRand(20); <
```

статический метод

расширяемый тип

Метод расширения вызывается как экземплярный

Перегрузка операций

Операция это специальный метод, принимающий параметры и возвращающий значение

```
class SimpleClass { ... }

...
SimpleClass instance1 = new SimpleClass();
SimpleClass instance2 = new SimpleClass();
...

? = instance1 + instance2;
```

Перегрузка операций

C# определяет три категории операций, которые могут вызвать перегрузку

Унарные операции – при перегрузке этих операций, необходимо указать один параметр, который должен быть того же типа, что и класс, который определяет оператор

Бинарные операции – при перегрузке этих операций необходимо указать два параметра, по крайней мере один из которых должен быть того же типа, что и класс, определяющий операцию

Операции преобразования – операции можно использовать для преобразования данных одного типа в другой. При перегрузке этих операций, необходимо указать один параметр, содержащий данные, которые нужно конвертировать. Эти данные могут быть любого допустимого типа

Перегрузка операций

Операции	Возможность перегрузки
+, -, !, ~, ++, --, true, false	Унарные операции, которые могут быть перегружены
+, -, *, /, %, &, , ^, <<, >>	Бинарные операции, которые могут быть перегружены
==, !=, <, >, <=, >=	Операции сравнения, которые могут быть перегружены
&&,	Условные логические операции не могут быть перегружены, но они оцениваются с помощью & и , которые могут быть перегружены
[]	Операция индексирования массива не может быть перегружена, но можно определить индексаторы, которые могут быть перегружены
()	Операция приведения не может быть перегружена, но можно определить новые операции преобразования
+=, -=, *=, /=, %=, &=, =, ^=, <<=, >>=	Операции присваивания не могут быть перегружены, но +=, например, оценивается с помощью операции «+», которая может быть перегружена
=, ., ?:, ->, new, is, sizeof, typeid	Эти операции не могут быть перегружены

Перегрузка операций

Рекомендуемый
модификатор
доступа

public static Hour operator +(Hour lhs, Hour rhs) =>
new Hour(lhs.value + rhs.value);

Тип
возвращаемого
значения

```
struct Hour
{
    private int value;

    public Hour(int hr) => this.value = hr % 24;
}
```

Метод должен
быть static

Именем метода является ключевое
слово operator вместе с символом
перегружаемой операции

Операнды
перегружаемой
бинарной операции

Перегрузка операций

```
public static Hour operator +(Hour lhs, Hour rhs) =>  
    new Hour(lhs.value + rhs.value);
```

```
Hour Example(Hour first, Hour second) => first + second;
```



```
Hour Example(Hour a, Hour b)  
{  
    return Hour.operator +(a,b); // pseudocode  
}
```

Перегрузка операций

При перегрузке операции по крайней мере один из параметров должен быть всегда содержащего операцию типа

```
public static Hour operator +(Hour lhs, Hour rhs) =>  
    new Hour(lhs.value + rhs.value);
```

```
public static Hour operator +(Hour lhs, int rhs) =>  
    new Hour(lhs.value + rhs);
```

```
public static Hour operator +(int lhs, Hour rhs) => rhs + lhs;
```

Перегрузить операцию в классе можно столько раз, сколько необходимо до тех пор, пока компилятор C# сможет различить каждую перегрузку

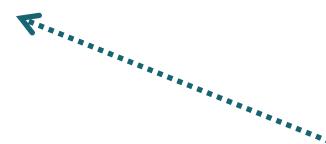
Перегрузка операций

```
public class Point
{
    public double X { get; set; }

    public double Y { get; set; }

    public static bool operator true(Point a) => (a.X > 0) && (a.Y > 0);

    public static bool operator false(Point a) => (a.X <= 0) || (a.Y <= 0);
}
```



Любой класс или структура могут перегрузить операции `true` и `false`, которые перегружаются парой

```
var p1 = new Point { X = 10, Y = 20 };
var p2 = new Point();
if (p2)
    Console.WriteLine("Point is positive");
else
    Console.WriteLine("Point is non-positive");
```

Перегрузка операций. Операции преобразования

```
struct Hour
{
    public static implicit operator int (Hour from) => from.value;
    private int value;
}

class Example
{
    public static void OtherMethod(int parameter) { }

    public static void Main()
    {
        Hour lunch = new Hour(12);
        Example.OtherMethod(lunch);
    }
}
```

неявное
преобразование:
расширяющее
происходит без
потери точности

неявное
преобразование

Перегрузка операций. Операции преобразования

```
struct Hour
{
    public Hour(int hr) => this.value = hr % 24;
```

```
    public static explicit operator Hour (int from) => new Hour(from);
```

```
    private int value;
```

```
}
```

```
class Example
{
```

```
    public static void OtherMethod(Hour parameter) { }
```

```
    public static void Main()
```

```
    {
        int lunch = 12;
```

```
        Example.OtherMethod((Hour)lunch);
```

Явное преобразование
сужающее преобразование:
существует риск потери
данных

Явное преобразование

```
}
```

Перегрузка операций. Операции преобразования

```
struct Hour
{
    private int value;

    public Hour(int hr) => this.value = hr % 24;

    public static Hour operator +(Hour lhs, Hour rhs) =>
        new Hour(lhs.value + rhs.value);

    public static explicit operator Hour(int from) => new Hour(from);
}
```



Операции преобразования предоставляют альтернативный способ решения проблемы обеспечения симметрических операций

Ограничения перегрузки операций

- Нельзя изменить приоритет или ассоциативность операции
- Нельзя изменить множественность операции
- Нельзя придумать новые символы операции
- Нельзя изменить смысл операций по отношению ко встроенным типам
- Операции сравнения необходимо реализовать в парах
- Нельзя перегрузить все операции
- Если в классе определяются операции «==» и «!=», необходимо переопределить методы GetHashCode и Equals

Рекомендации при перегрузка операций

C# Operator Symbol	Spesial Method Name	Suggested CLR-Complant Method Name
+	op_Addition	Add
-	op_Substraction	Substruct
*	op_Multiply	Multiply
/	op_Division	Devide
==	op_Equality	Equals
!=	op_Inequality	Compare
...

Рекомендации при перегрузка операций

```
class Salary
{
    private decimal amount;

    public decimal Amount
    {
        get => this.amount;
    }

    public Salary(decimal amount) => this.amount = amount;

    public static Salary operator +(Salary salary, decimal number)
    {
        salary.amount += number; // Не следует изменять операнды
return salary;
        return new Salary(salary.Amount + number);
    }
}

Salary salary = new Salary (99);
Salary newSalary = salary + 10;
Console.WriteLine($"{salary.Amount} {newSalary.Amount}");
```

Рекомендации при перегрузка операций

salary + 99 = 99 + salary

Определять
симметричные
операции

```
public static Salary operator +(Salary lhs, decimal rhs)  
    => new Salary(lhs.Amount + rhs);  
  
public static Salary operator +(decimal rhs, Salary lhs) => rhs + lhs.Amount;
```

вызов первого метода
позволит избежать
дублирования кода

Рекомендации при перегрузка операций

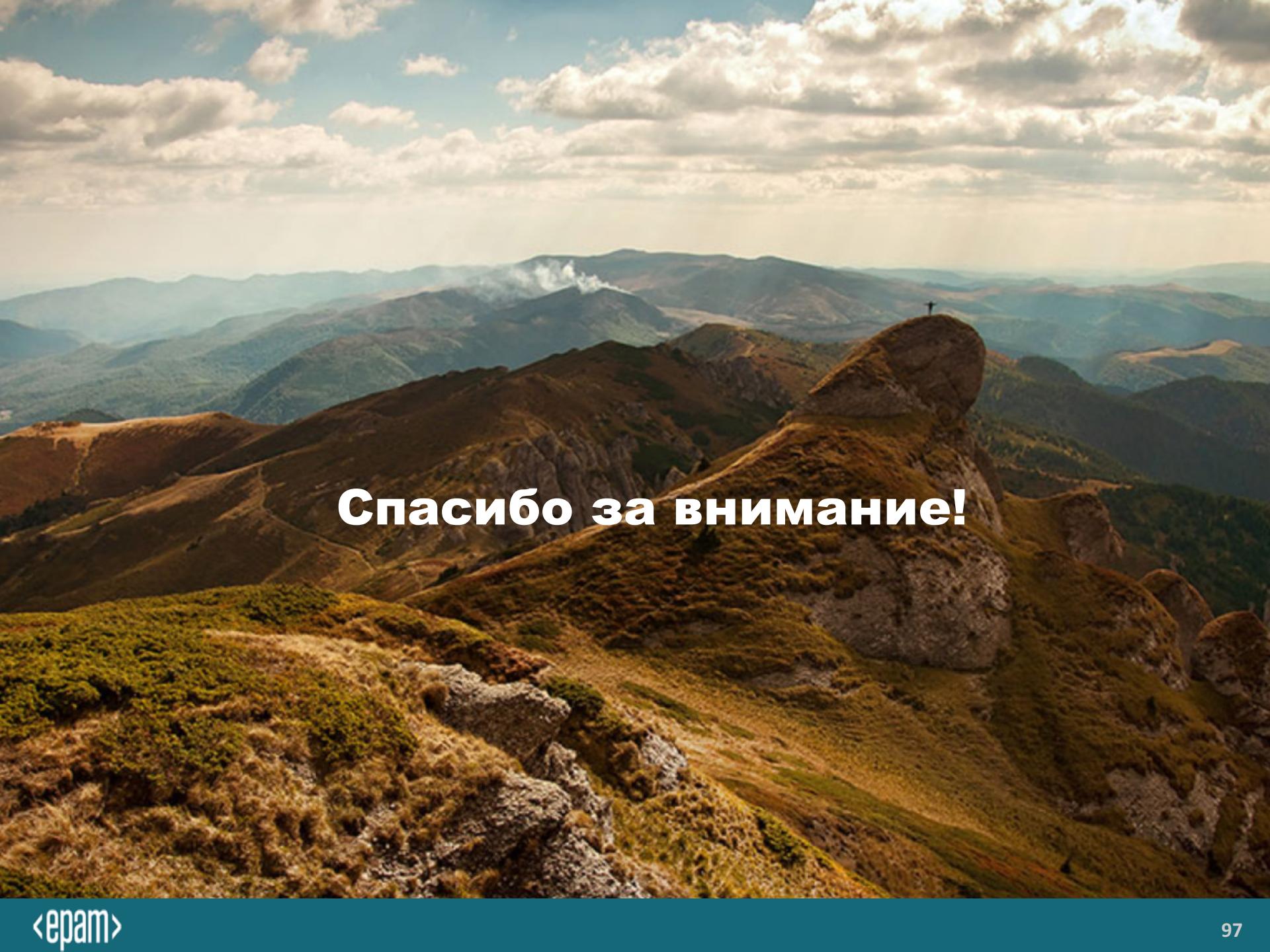
```
class Account
{
    public static Account operator +(Account account, decimal amount) { }
    public static Account operator -(Account account, decimal amount) { }

    Withdraw(decimal amount) { }

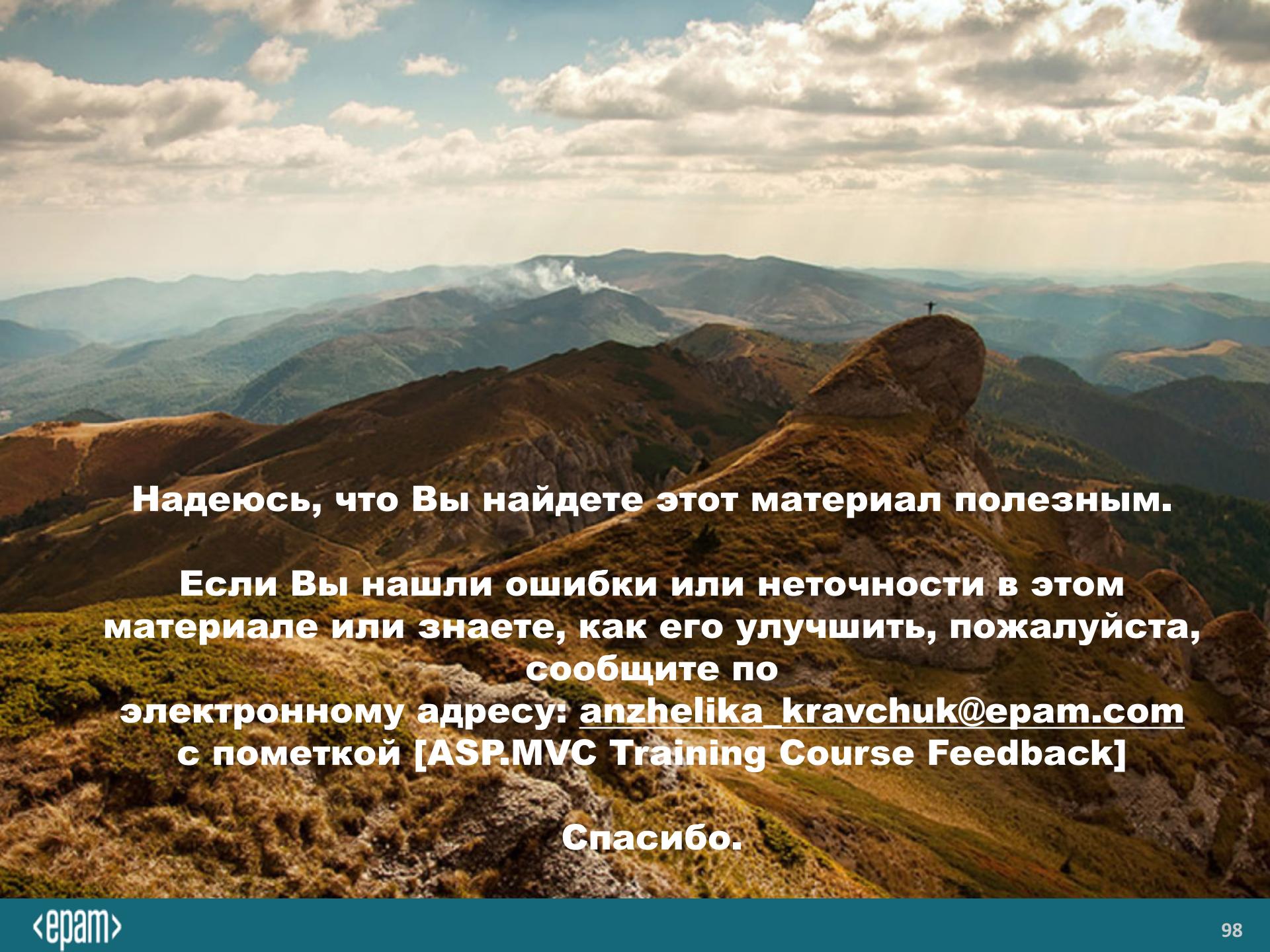
    Deposit(decimal amount) { }

}
```

Определять
только имеющие
смысла операции

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 the middle layer. The sky is filled with large, fluffy clouds.

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.