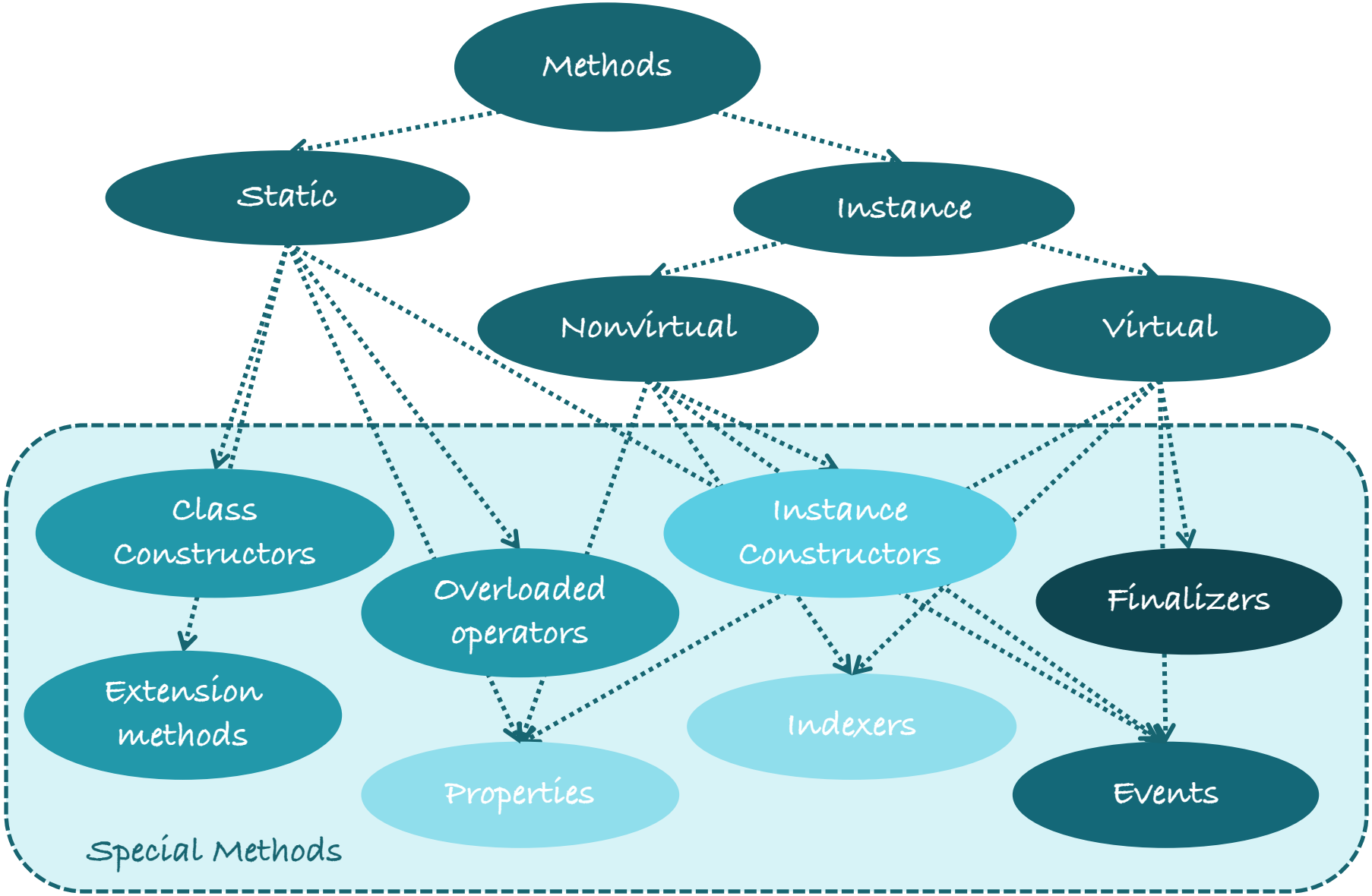

epam
METHODS IN DETAILS
.NET & JS LAB, RD BELARUS
Anzhelika Kravchuk

Methods types



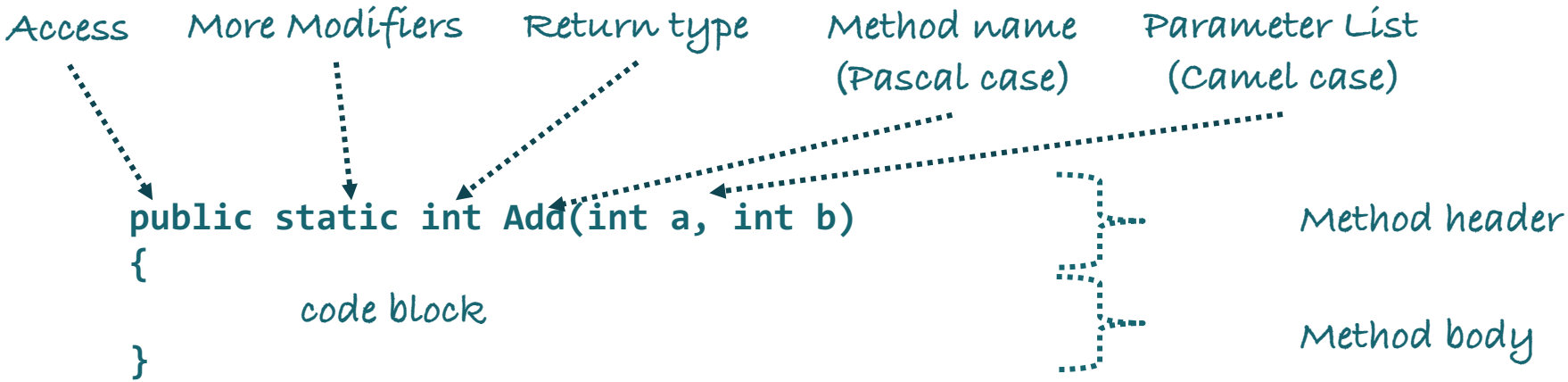
Defining Methods

A method is a class member that contains a code block that represents an action:

- all executable code belongs to a method
- a C# application must have at least one method

The declaration of a method consists of two parts:

- a method header
- an optional implementation code block (referred to as the method body).



Methods modifiers

Static modifier	`static`
Access modifier	`public internal private protected`
Inheritance modifier	`new virtual abstract override sealed`
Unmanaged code modifier	`unsafe extern`
Partial method modifier	`partial`
Asynchronous code modifier	`async`

Parameters

Terms that are often confused and incorrectly used interchangeably are parameter and argument. The distinction is quite simple, though. Parameters live on a declaration site (for example, in the parameter list of a method), whereas arguments are specified in a call site (for example, when invoking a method). Depending on the angle you approach them, they're very similar: One makes up the receiving end; the other makes up the sending end.

Parameters

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

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

Arguments

```
SomeMethod(10);

SomeMethod(100, 54.321F);
```

Here are a couple of valid method headers with a different numbers of parameters and some modifiers

```
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)
```

Return Type

The return type of a method can be either a type or void. This indicates what a caller of the method will get (under normal circumstances) in return for calling the method.

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

     return n / d;
}
```

When a type is specified, all execution paths through the method's body should reach a point where the return keyword is used to hand back a result to the caller.

It's also possible for a method to throw an exception.

Expression-bodies methods (C# 6)

A method that comprises a single expression, such as the following can be written more tersely as an expression-bodied method. A fat arrow replaces the braces and return keyword:

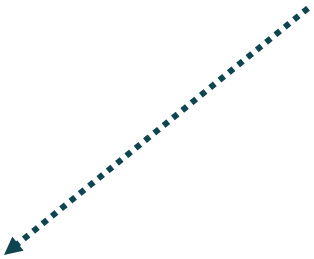
```
class Customer
{
     public string First { get; }
     public string Last { get; }

     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);

}
```

Expression-bodied syntax can use in properties and indexers

Expression-bodied functions can also have a void return type

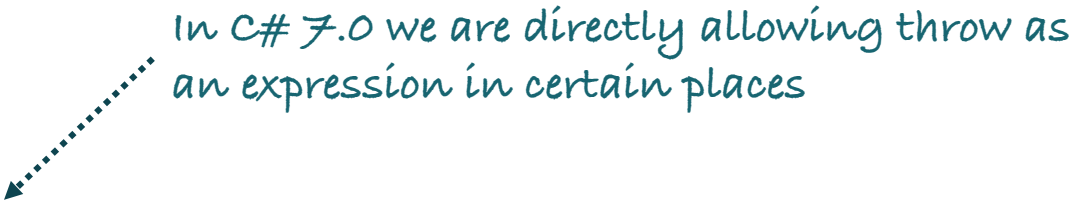
Expression-bodied methods (C# 7)

More expression bodied members –
C# 7.0 adds accessors, constructors and finalizers to the list of things that can have expression bodies



```
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;
     }
}
```

Throw expressions (C# 7)




```
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();
}
```

Calling a Method

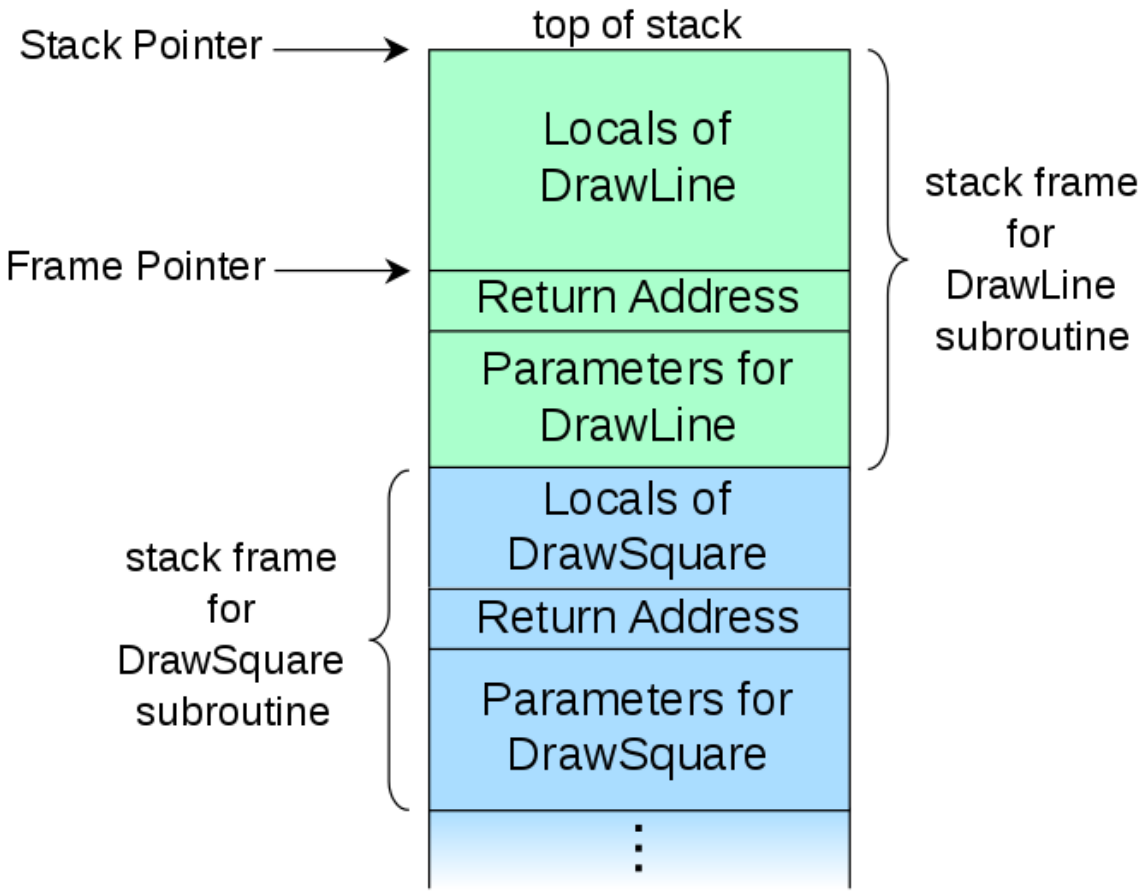
To call a method:

- Specify the method name
- Provide an argument for each parameter
- Handle the return value

```
int i = 1;
int j = 2;
int result = Sum(i++, i+j);

int Sum(int first, int second)
{
    return first + second;
}
```

Method arguments are evaluated in left-to-right order. This can have an effect if the process of evaluating one argument affects the value of another argument

Calling a Method



Intermediate language (IL)

Intermediate language (IL) is an object-oriented programming language designed to be used by compilers for the .NET Framework before static or dynamic compilation to machine code. The IL is used by the .NET Framework to generate machine-independent code as the output of compilation of the source code written in any .NET programming language.

IL is a stack-based assembly language that gets converted to byte code during execution of a virtual machine. It is defined by the common language infrastructure (CLI) specification. As IL is used for automatic generation of compiled code, there is no need to learn its syntax.

This term is also known as Microsoft intermediate language (MSIL) or 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)

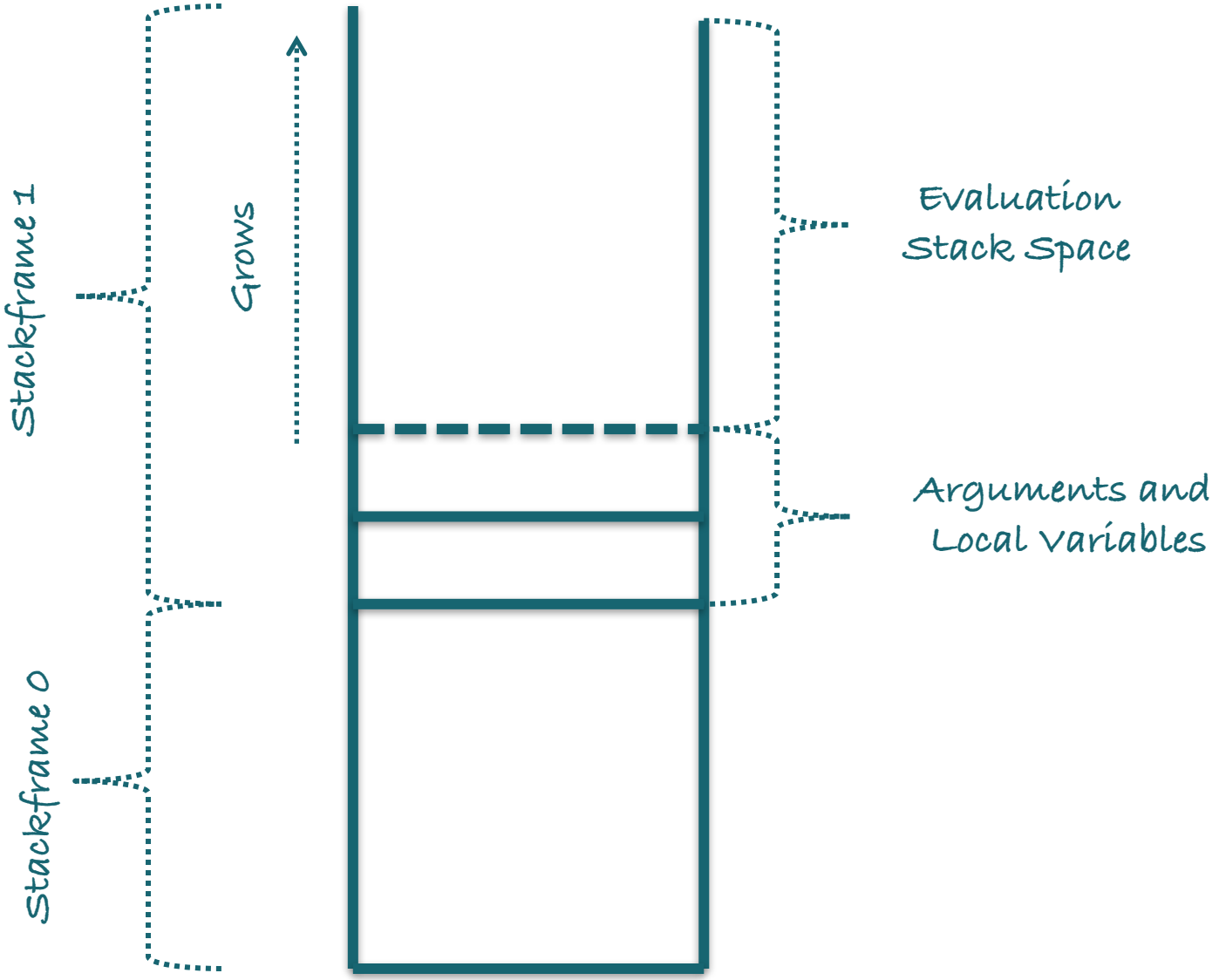
The Evaluation Stack

The evaluation stack is the pivotal structure of MSIL applications. It is the bridge between your application and memory locations. It is similar to the conventional stack frame, but there are salient differences.

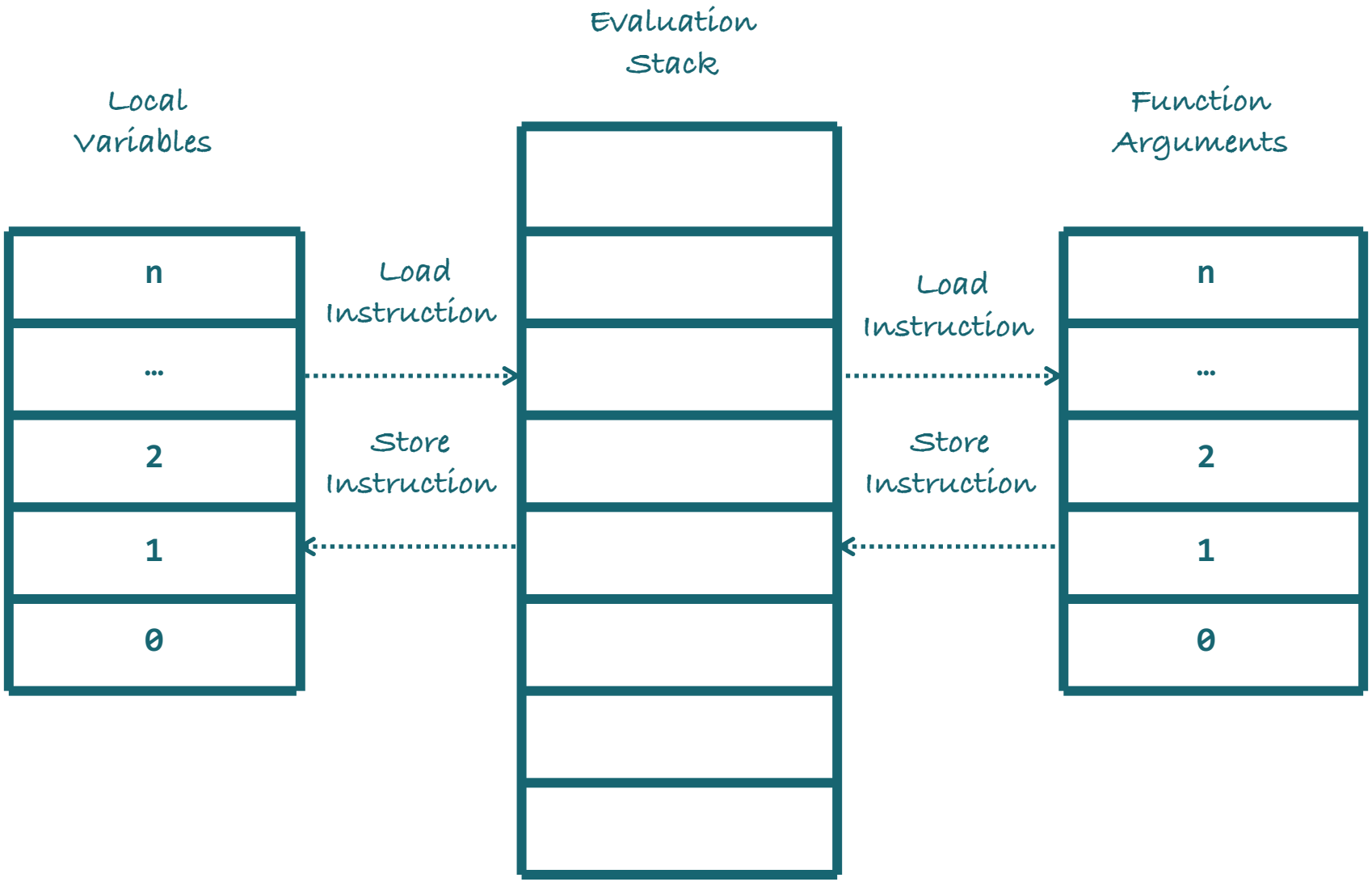
The evaluation stack is the viewer of the application, and you can use it to view function parameters, local variables, temporary objects, and much more. Traditionally, function parameters and local variables are placed on the stack. In .NET, this information is stored in separate repositories, in which memory is reserved for function parameters and local variables.

You cannot access these repositories directly. Accessing parameters or local variables requires moving the data from memory to slots on the evaluation stack using a *load* command. Conversely, you update a local variable or parameter with content on the evaluation stack using a *store* command. Slots on the evaluation stack are either 4 or 8 bytes.

The Evaluation Stack



The Evaluation Stack



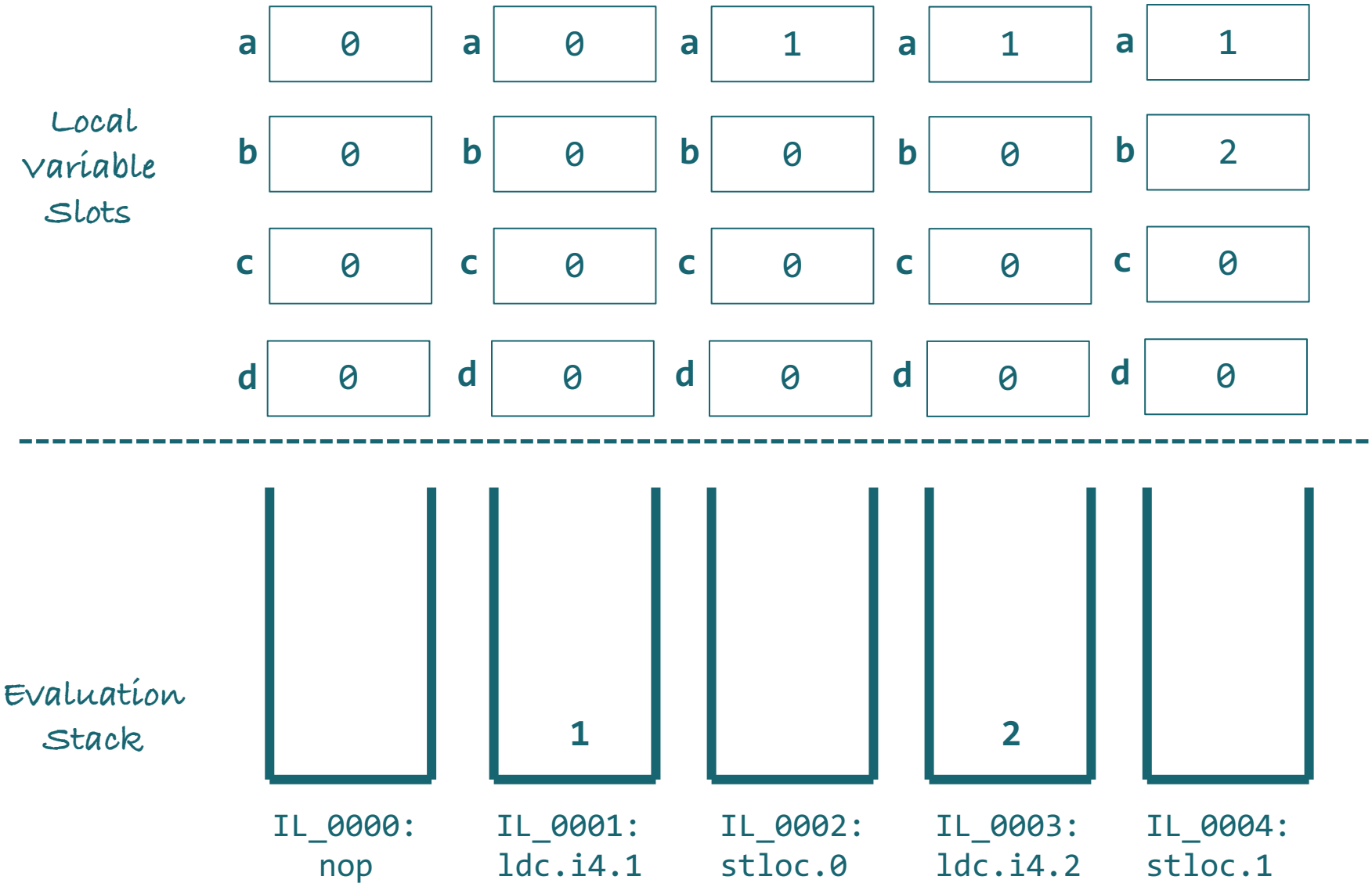
The Evaluation Stack

```
int a = 1;
int b = 2;
int c = 3;

int d = a + b * c;
```

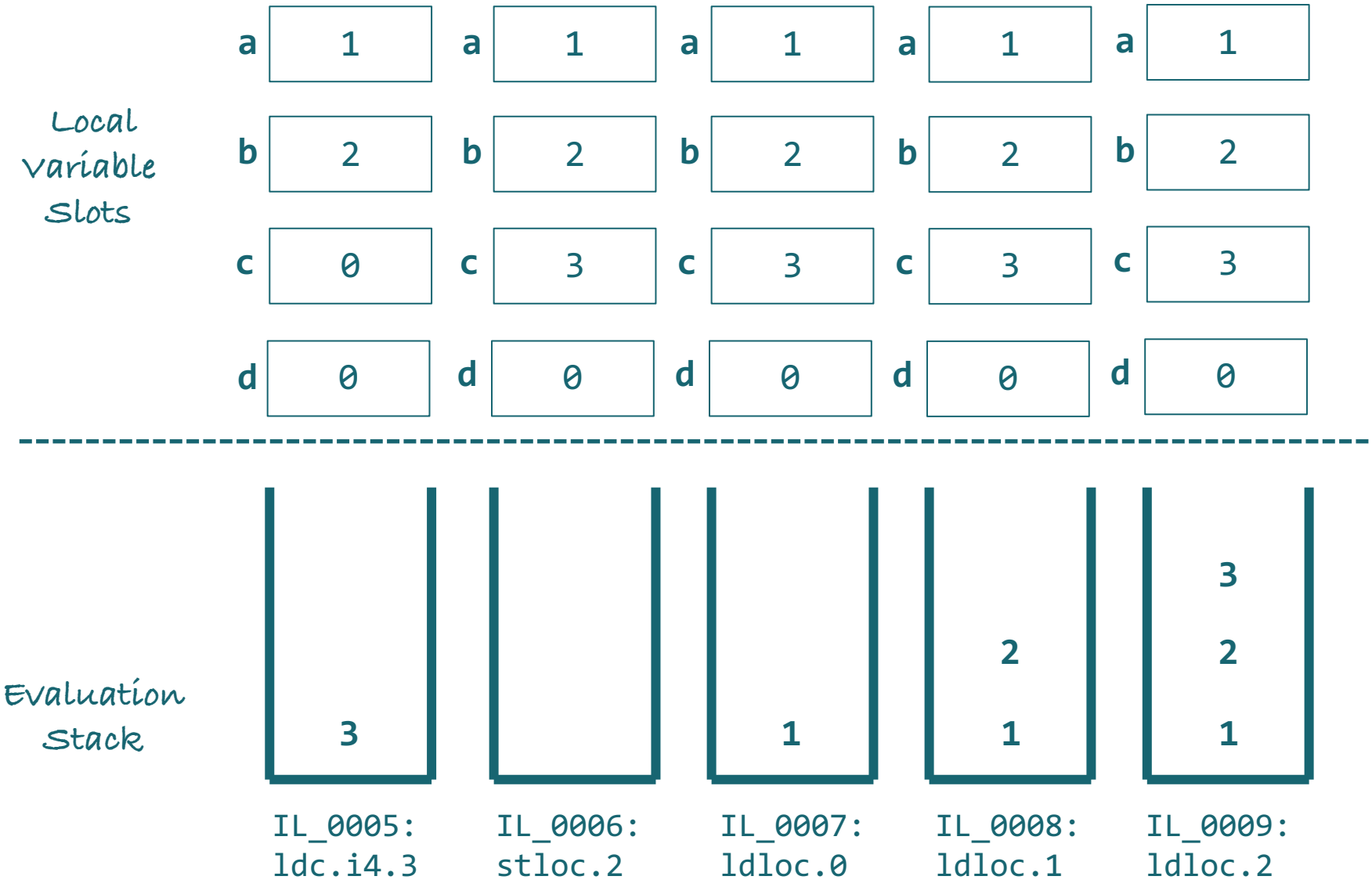
C# compiler

```
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
```

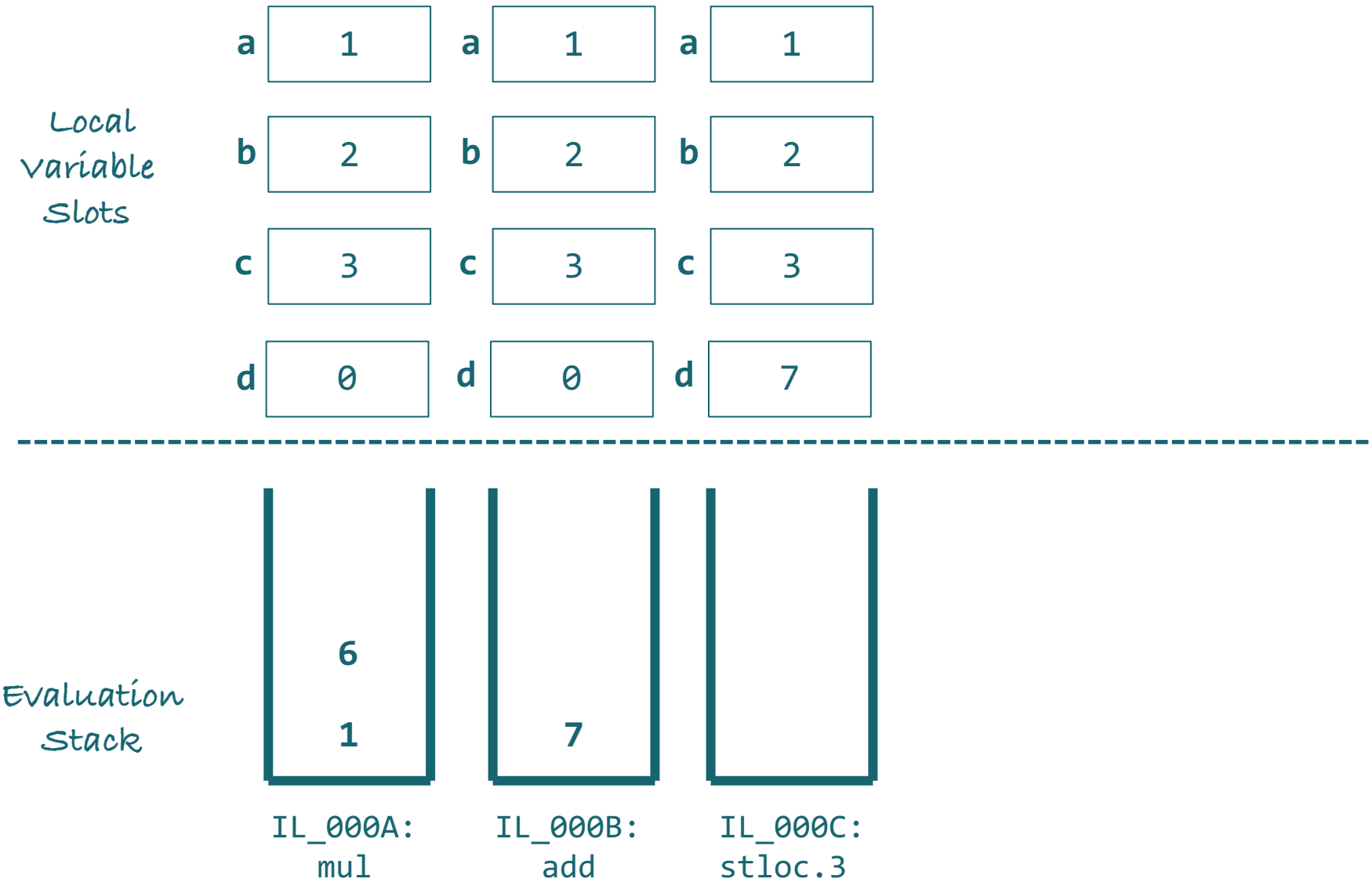
The Evaluation Stack



The Evaluation Stack



The Evaluation Stack



The Evaluation Stack

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

C# compiler

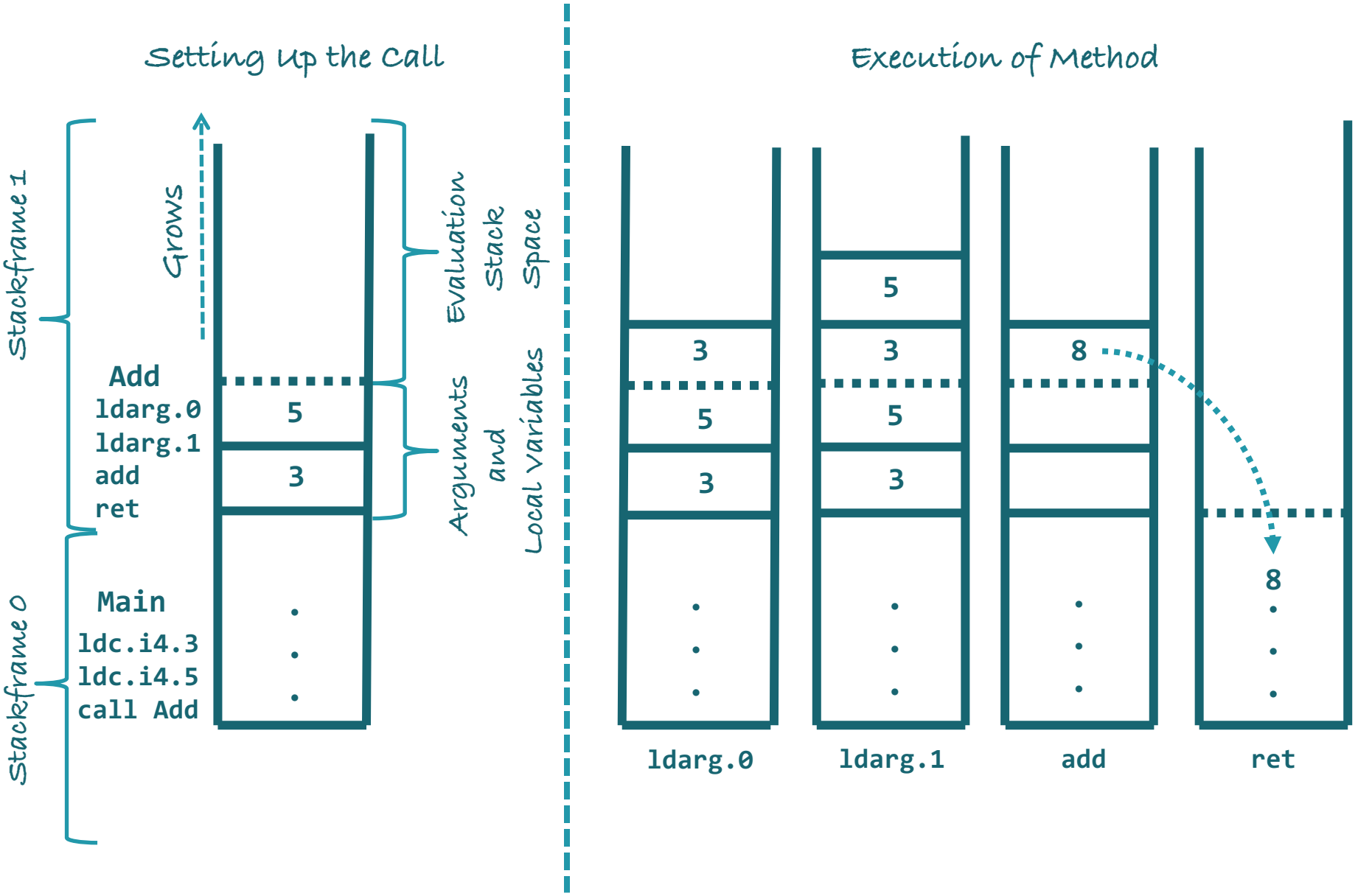
```
IL_0000:  nop
IL_0001:  ldc.i4.1
IL_0002:  stloc.0
IL_0003:  ldloc.0
IL_0004:  dup
IL_0005:  ldc.i4.1
IL_0006:  add
IL_0007:  stloc.0
IL_0008:  stloc.1
IL_0009:  ret
```

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

C# compiler

```
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
```

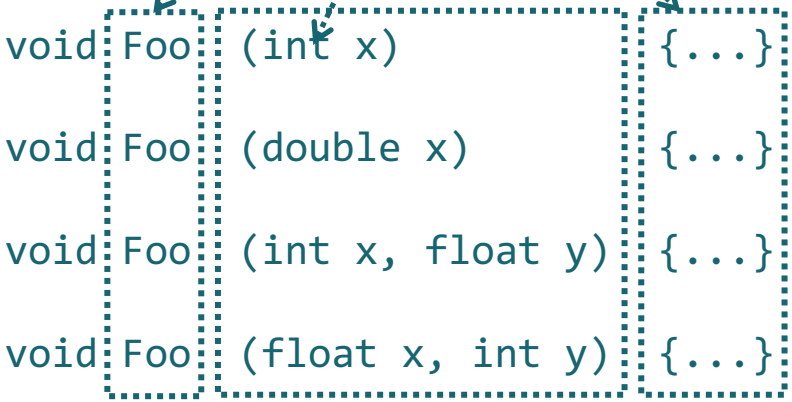
The Evaluation Stack



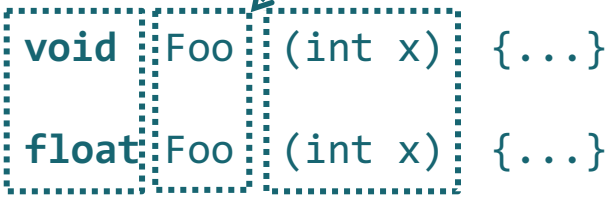
Overloaded Methods

An overloaded method:

- Has the same name as an existing method
- *Should* perform the same operation as the existing method
- Uses different parameters to perform the operation



The following pairs of methods cannot coexist in the same type, since the return type are not part of a method's signature



Value Parameters

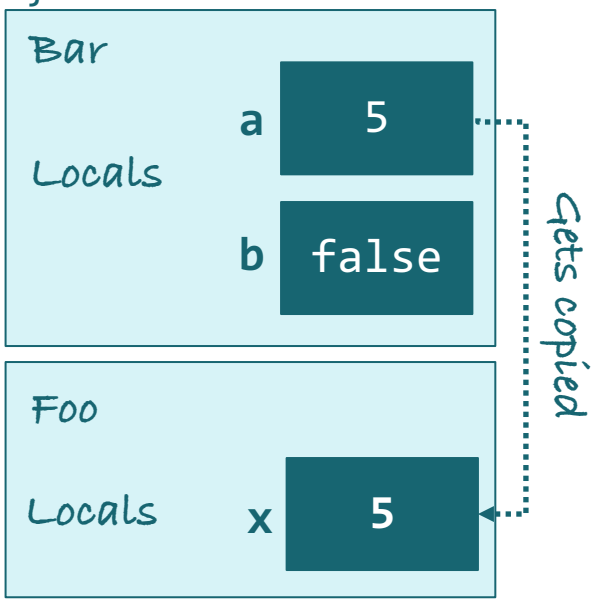
One type of parameter is known as a value parameter. Basically, it's a parameter that doesn't have special modifiers. But why is it called a *value* parameter? Basically, such a parameter receives its input (the *argument* on the call site) *by value*.

Notice that this has nothing to do with value or reference types but is simply an aspect of the invocation of a method! Passing a parameter by value means that one can simply assign to the parameter inside the method without affecting the call-site argument.

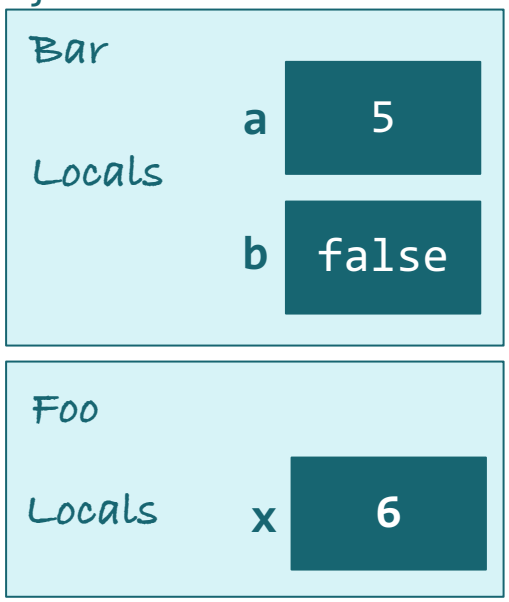
```
void SomeMethod(int first, double second)
{
    ...
}
```

Value Parameters

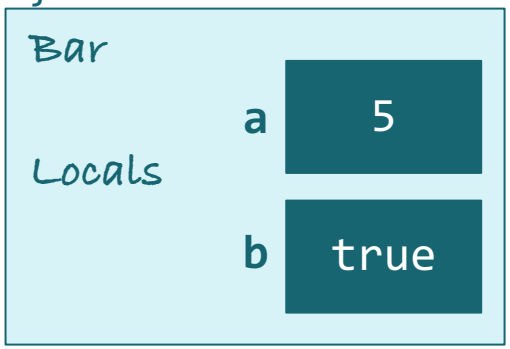
```
void Bar()
{
    int a = 5;
    Foo(a);
    bool b = a == 5;
}

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




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

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

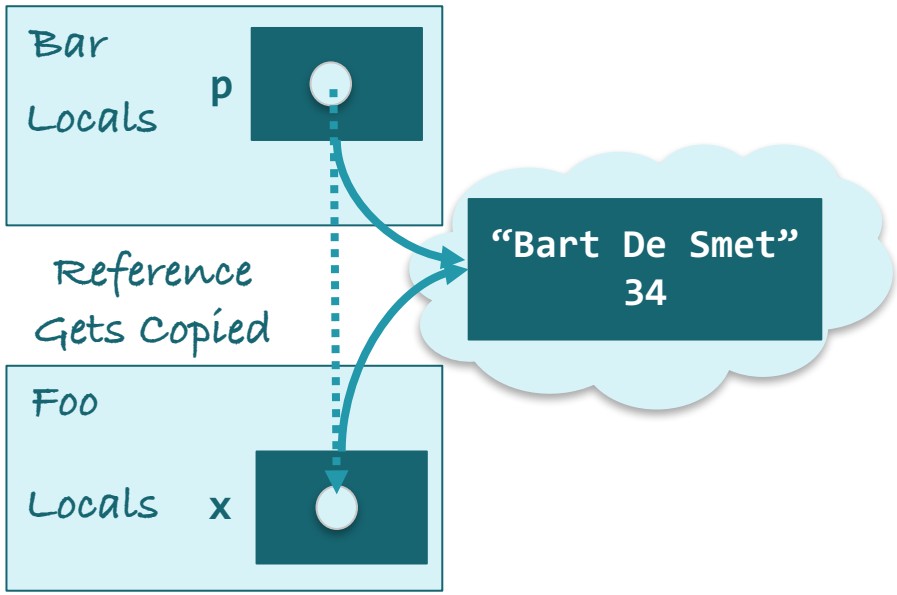



```
void Bar()
{
    int a = 5;
    Foo(a);
    bool b = a == 5;
}

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

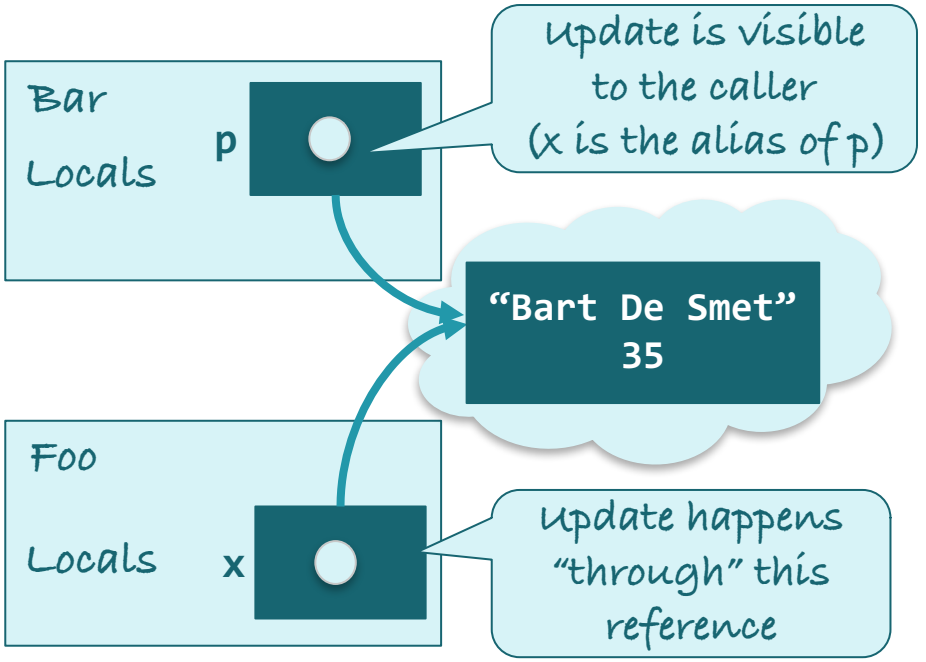


Value Parameters

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



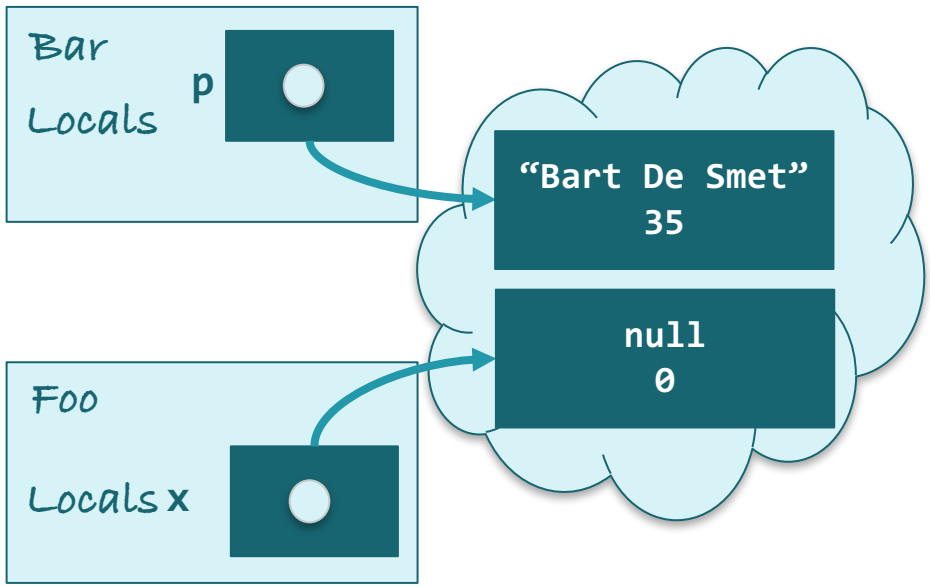

```
void Bar()
{
    Person p = ...;
    Foo(p);
    ...;
}
void Foo(Person x)
{
    x.Age++;
    x = new Person();
```

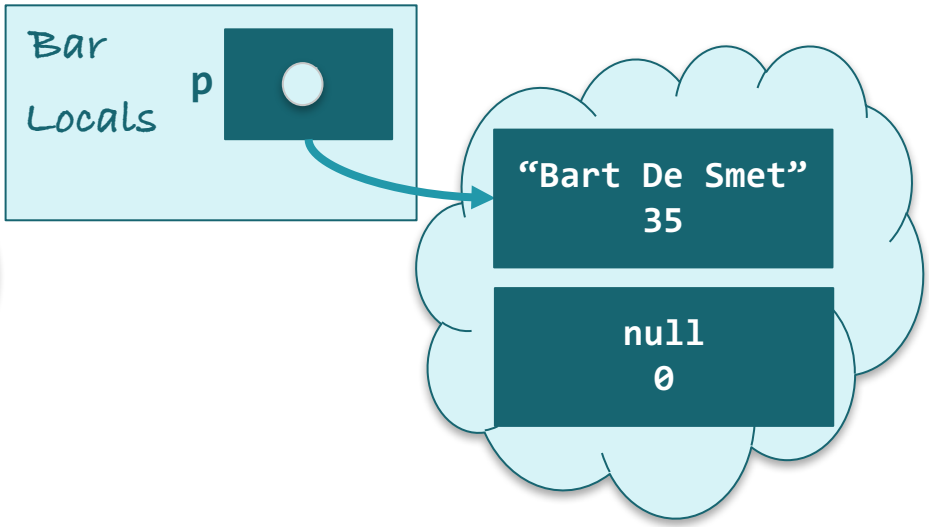


Value Parameters

```
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();
}
```





Value Parameters

A reference or value type simply indicates where instances of the type get allocated and what gets copied upon passing such an object *by value*.

Value-typed objects are copied wholesale, whereas for reference-typed objects the reference to the target instance is copied, causing aliasing (multiple references referring to the same instance).

When using value parameters, the receiving end holds such a local “copy” of the object. Any assignment to that local cell is invisible to the caller, but modifications made through a reference will be visible to the caller due to aliasing.

Reference Parameters

Although assignment to a value parameter is visible only to the current method and not to its caller, reference parameters can circumvent this. Instead of making a copy of the object (whatever that means is dependent on the value- or reference-type characteristic of the object), a reference is made to the local variable held by the caller.

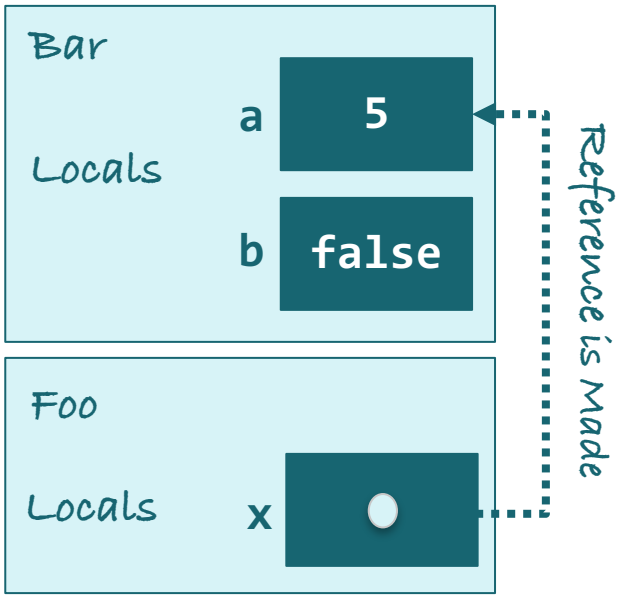
```
void SomeMethod(int first, double second, ref int refData)
{
    ...
    refData = 100;
    ...
}

int value = 1;
SomeMethod(10, 101.1, ref value);
```

The use of the ref keyword is mandatory both on the declaration of the parameter and when passing the argument. The latter requirement was introduced to make it obvious what's going on when reading the code of the call site.

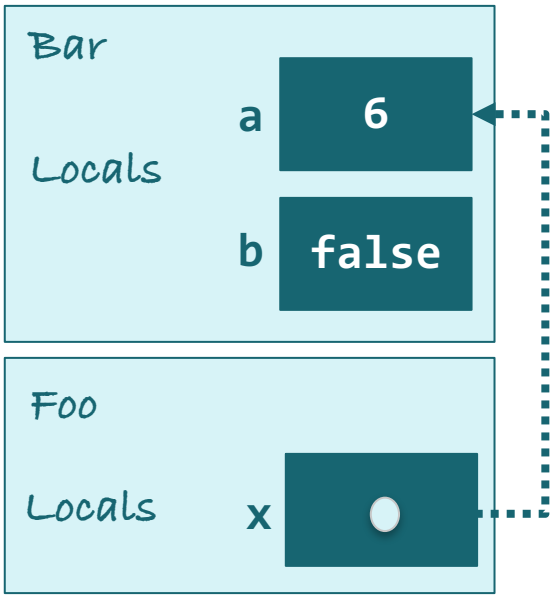
Whenever you see an argument being passed by reference, you can expect the variable that's passed to be replaced wholesale by the method being called

Reference Parameters

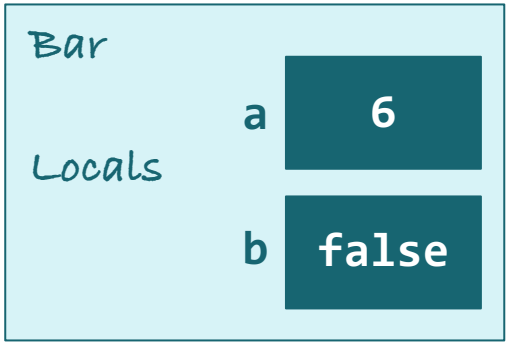
```
void Bar()
{
    int a = 5;
    Foo(ref a);
    bool b = a == 5;
}

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




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

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




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

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



Reference Parameters (C# 7)

Ref returns/locals are now possible for advanced scenarios

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

Return the storage location, not the value

Aliases 7's place in array

```
int[] array = { 1, 15, -39, 0, 7, 14, -12 };

ref int place = ref Find (7, array);

place = 9;
```

Replaces 7 with 9 in the array

Output Parameters

We use the return type to denote success or failure, while the output parameter will receive the result in case of success.

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


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

Closely related to reference parameters are output parameters. Conceptually, they serve as additions to the single "return channel" provided by a method's return type.

Output Parameters

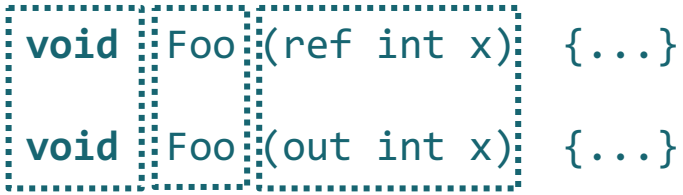
```
Console.Write("Enter your age: ")
string input = Console.ReadLine();
int age;
if (!int.TryParse(input, out age))
    // Print error message, maybe let the user retry
else
    // We got a valid age
```

This is a common pattern for various Base Class Library (BCL) types such as numeric value types that have a TryParse method

The difference between TryParse and Parse is what happens upon passing invalid input to the method. Parse throws an exception, whereas TryParse doesn't. Because exceptions are expensive and an invalid numeric string is not an exceptional case when dealing with user input, TryParse makes much more sense because it communicates success or failure through a Boolean value that can be checked easily.

Output Parameters

The following pairs of methods cannot coexist in the same type, since the out/ref key words are not part of a method's signature



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

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

Internally, ref and out are implemented both as ref. In fact, the common language runtime does not have a notion of output parameters.

So what's the key difference between both parameter types from a C# programmer's point of view? The answer lies in "definite assignment rules" both at the side of the caller and of the callee. For a variable to be passed to a reference parameter, it must be assigned first. For output parameters, the responsibility for assignment is with the method being called, which should guarantee that every code path ensures the output parameter to get assigned to.

Output Parameters

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

C# compiler

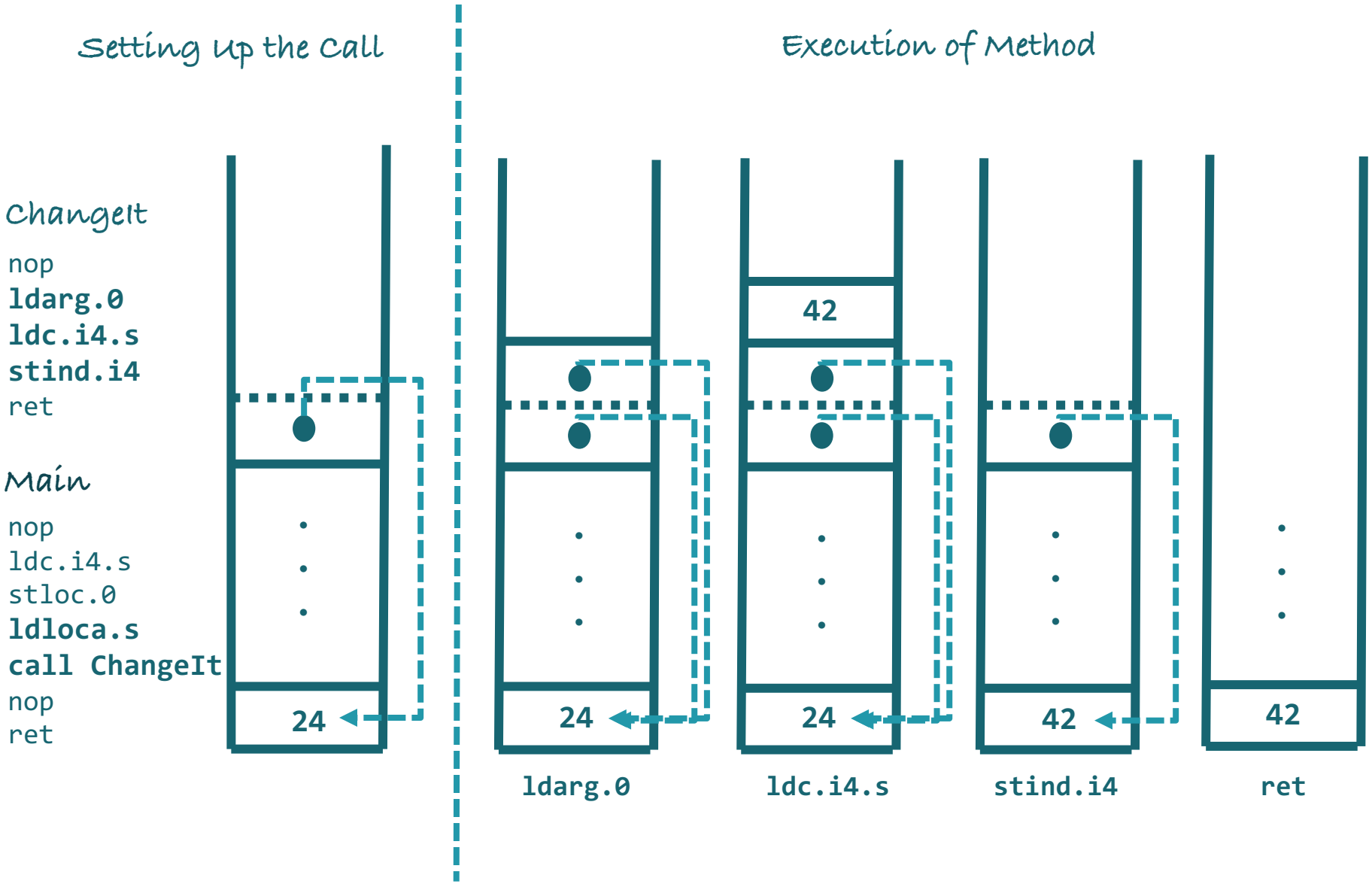
```
ChangeIt:
IL_0000:  nop
IL_0001:  ldarg.0
IL_0002:  ldc.i4.s    2A
IL_0004:  stind.i4
IL_0005:  ret
```

```
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
```

Output Parameters



Output Parameters

DON'T USE TOO MUCH!

Use of output parameters can seem tempting to return multiple results from a method but is rather sticky in the call site because a separate local variable must be declared upfront and the out keyword is required. When you start returning a lot of data, consider wrapping it in some specialized return type. Also starting from .NET 4, you can use the System.Tuple generic types to package objects.

Output Parameters (C# 7)

```
if (int.TryParse ("123", out int result))
Console.WriteLine(result);
int.TryParse ("234", out var result);
```

Result has been declared inline

The newly introduced variable is still in scope

Out parameters can be implicitly typed (var)

Output Parameters (C# 7)

```
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");

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

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

Discard out argument

You can use the underscore multiple times in a single method

Output Parameters vs Tuple (C# 7)

```
Tuple<string, int> tupleBefore = new Tuple<string, int>("three", 3);

var tuple = ("three", 3);

(string, int) tuple2 = tuple;

var namedTuple = (word:"three", number:3);

Console.WriteLine(namedTuple.number);

Console.WriteLine(namedTuple.word);
```

Can create tuples easily

With explicit typing

We can even name the fields

Named tuples are compiled the same underneath (Item1, Item2)

But with compiler trickery, we can refer to the names in source code

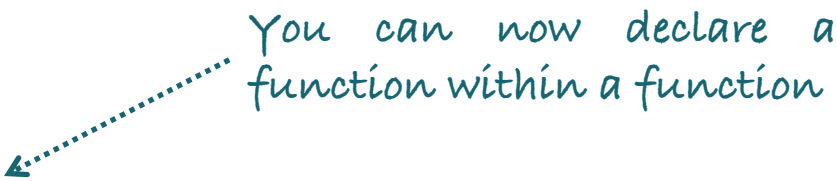
Output Parameters vs Tuple (C# 7)

- Out parameters: Use is clunky (even with the improvements described above), and they don't work with async methods.
- System.Tuple <...> return types: Verbose to use and require an allocation of a tuple object.
- Custom - built transport type for every method: A lot of code overhead for a type whose purpose is just to temporarily group a few values.
- Anonymous types returned through a dynamic return type: High performance overhead and no static type checking.

```
(string, DateTime)            Foo()       => ("Now", DateTime.Now);

(string name, DateTime time)  NamedFoo()  => ("Now", DateTime.Now);


Console.WriteLine(Foo().Item1);

Console.WriteLine(Foo().Item2);

Console.WriteLine(NamedFoo().name);

Console.WriteLine(NamedFoo().time);
```

Local Function

```
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));
```



Optional and Named Parameters

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

Methods can have overloads. Basically, that means that different method headers can exist that all share the same name but differ in the number of parameters or the type those parameters have. Often, one wants to provide “convenience overloads” for methods that take a bunch of parameters, supplying default values for more advanced parameters.

```
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) {...}
```

Optional and Named Parameters

With optional parameters, you can take an alternative approach for defining those methods. Instead of having more overloads, parameters can specify a default value that's put in place when the parameter is omitted in a method call

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

After specifying the parameter type and name, a default value is supplied using what looks like an assignment. Optional parameters have to go after all required parameters. Valid default values include constants such as numerals, Boolean values, strings, enums, and the null reference.

Optional and Named Parameters

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

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);
```

Parameter Arrays

```
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.
```

Some methods can benefit from having an unbounded number of arguments passed to them

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

versus

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

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

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

Parameter Arrays

They're simply parameters of some array type that appear at the end of the parameter list and are prefixed with the params keyword.

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

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

The receiving end simply sees them as arrays, but the calling side can omit the array creation code and simply pass arguments comma-separated as if they were corresponding to separate parameters

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

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

The last call basically gets turned into a form like the one here, where an array gets allocated on the user's behalf

Parameter Arrays

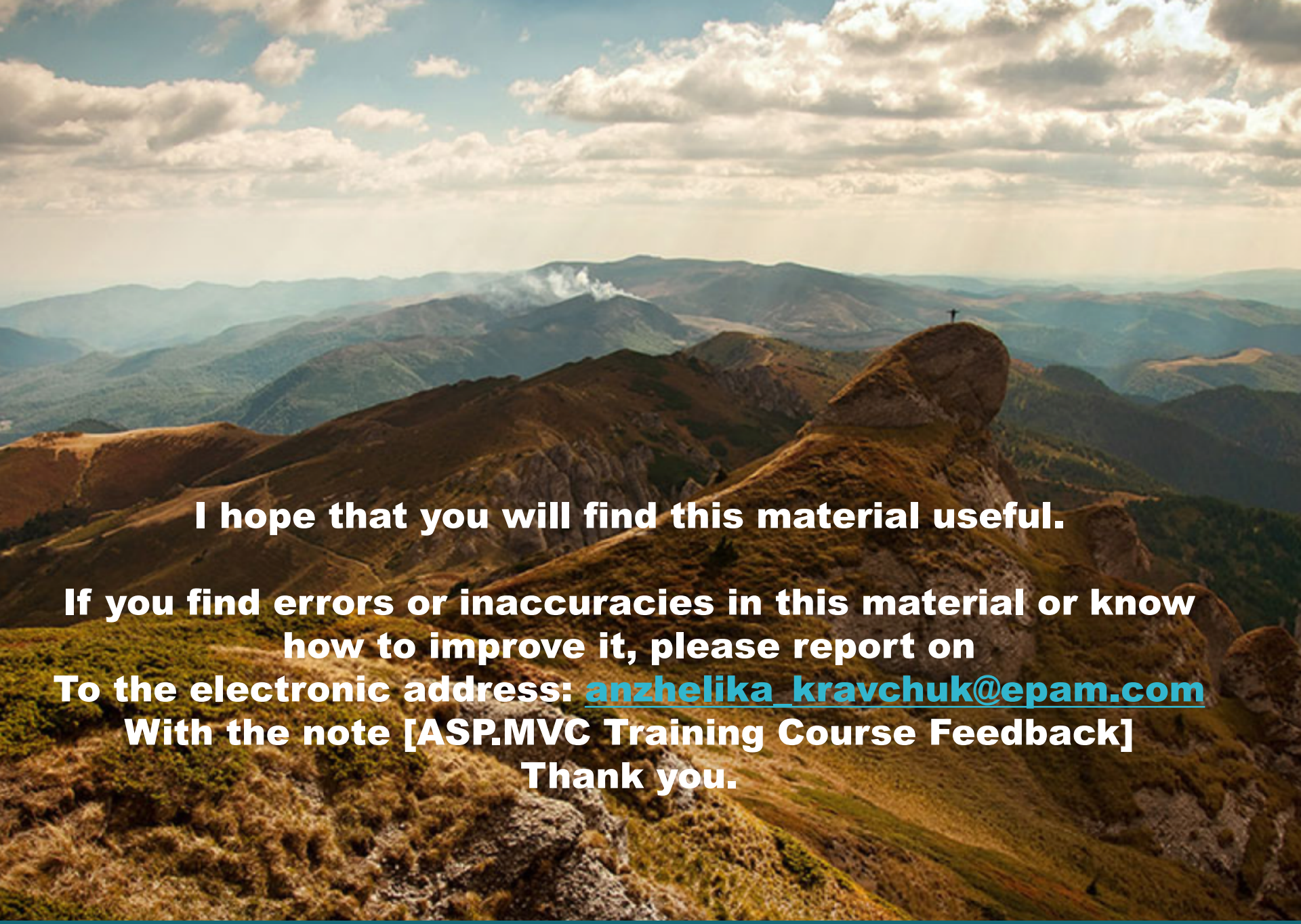
The definition of additional low-parameter-count regular methods in addition to one that takes a parameter array is a common practice to reduce excessive heap allocations for arrays passed to every single method call. When calling the Sum method with two or three arguments, the regular methods will take precedence over the parameter array one.

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

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

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

```
void Goo(int[] x)          {...}

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

The following pairs of methods cannot coexist in the same type, since the params modifier are not part of a method's signature


Thank you for attention!


I hope that you will find this material useful.
If you find errors or inaccuracies in this material or know how to improve it, please report on
To the electronic address: anzhelika_kravchuk@epam.com
With the note [ASP.MVC Training Course Feedback]
Thank you.