



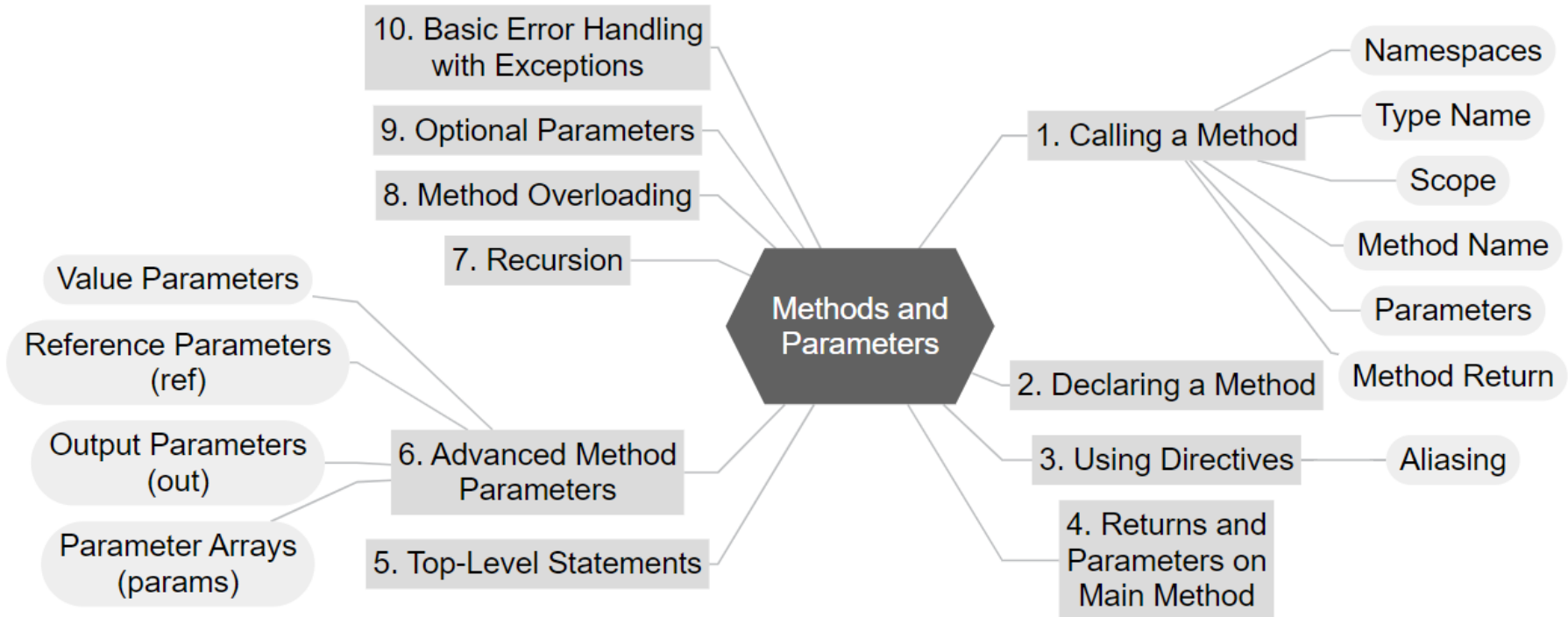
Fundamentals of Programming

Methods and Parameters

By: Võ Văn Hải

Email: vovanhai@ueh.edu.vn

Objectives



Introduction

What Is a Function?

- ▶ Functions are virtually identical to methods, used for grouping together a sequence of statements to perform a particular action or compute a particular result. In fact, frequently, method and function are used interchangeably.
- ▶ Guidelines:
 - DO give methods names that are verbs or verb phrases.

```
static void Main(string[] args) {  
    int lineCount = CountLines(text);  
    DisplayLineCount(lineCount);  
}  
  
static void DisplayLineCount(int lineCount) ...  
  
static int CountLines(String text) ...  
  
static void DisplayHelpText() ...
```

Instead of placing all of the statements into Main(), the we should break them into groups called methods/functions.

Functions/Methods

Types of function

► Types:

- Built-in Functions
- User-Defined Functions

- The function which is already defined in the framework and available to be used by the developer or programmer is called a built-in function, whereas if the function is defined by the developer or programmer explicitly, then it is called a user-defined function.

```
static void Main(string[] args)
{
    int number = 25;
    double squareRoot = Math.Sqrt(number);
    Console.WriteLine($"Square Root of {number} is {squareRoot}");
    Console.ReadKey();
}
```

Built-in Functions

User-Defined Functions

```
static int max(int a, int b)
{
    return a > b ? a : b;
}
```

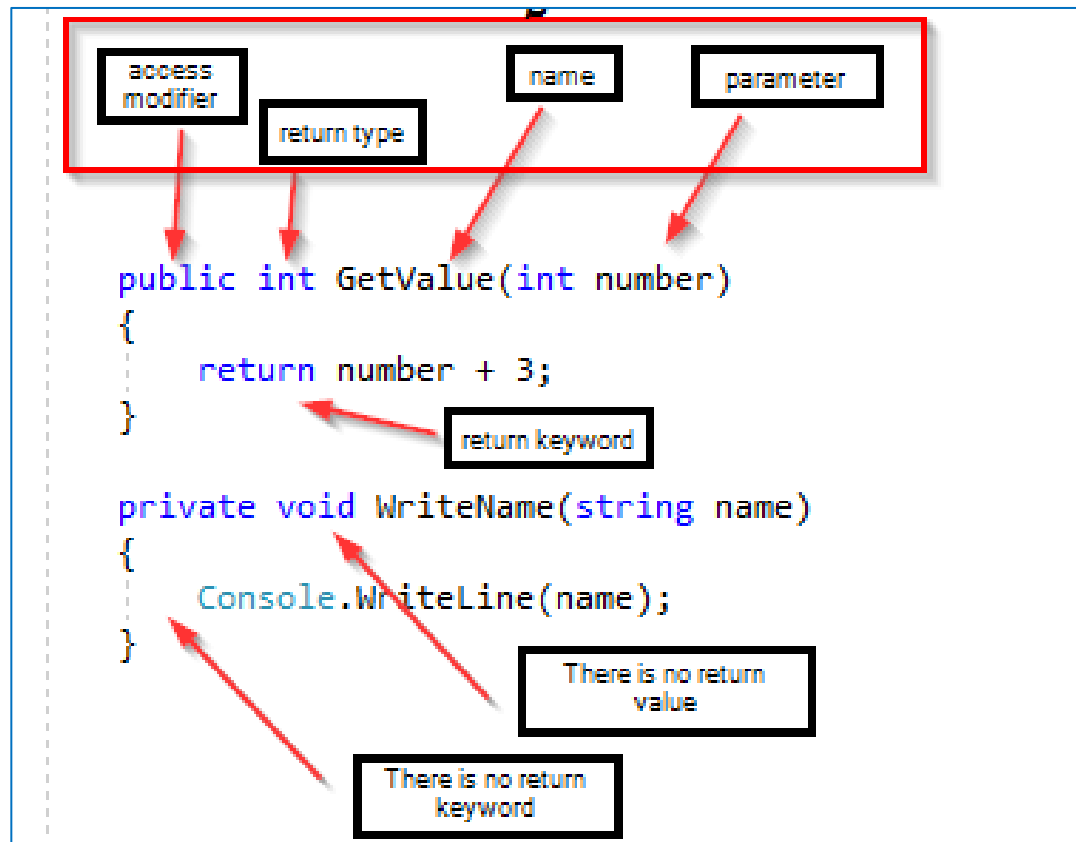
Functions/Methods

Declaration

► Syntax

```
<Access_Modifier> <return_type> <method_name>([<param_list>])
```

► Example



Functions/Methods

Different Parts of a Function

```
class Program
```

```
{
```

```
    static int Add(int a, int b)
```

← Function Signature or Prototype

```
{
```

```
    int sum = a + b;  
    return sum;  
}
```

← Function Definition or Body

```
    static void Main(string[] args)
```

```
{
```

```
        int x, y;  
        x = 10;  
        y = 15;
```

```
        int sum = Add(x, y);
```

← Function Call

```
        Console.WriteLine($"Sum is {sum}");  
        Console.ReadKey();
```

```
    }
```

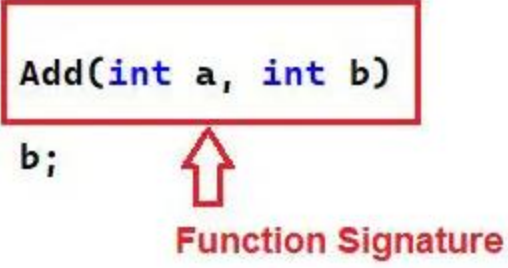
```
}
```

Functions/Methods

Function Signature

- ▶ In C#, a Function/Method Signature is consisting of two things:
 - the Method Name and
 - the Parameter List.
- ▶ The return type is not considered to be a part of the method signature.
- ▶ Example

```
public static int Add(int a, int b)
{
    int sum = a + b;
    return sum;
}
```



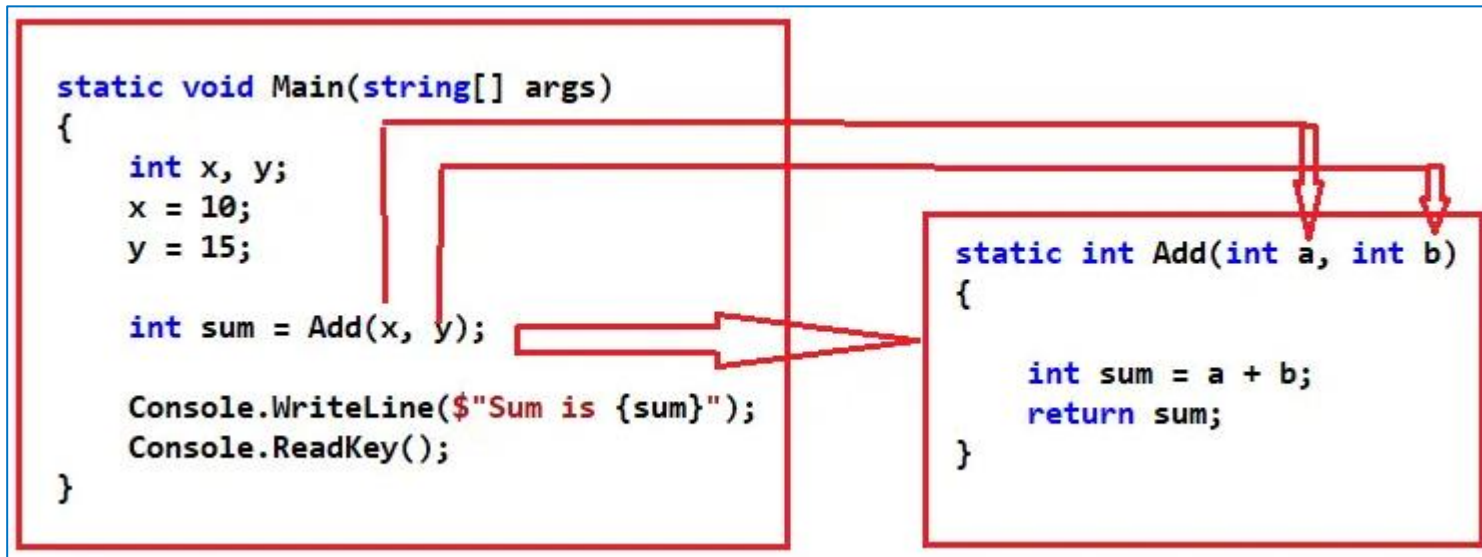
Function Signature

- ▶ The return statement:
 - terminates the execution of a function immediately and returns the control to the calling function.
 - causes your function to exit and return a value to its caller.

Functions/Methods

Invocation

- ▶ When we invoke a method, the control gets transferred to the called method. Then the called method returns the control to the caller method (from where we call the method) in the following three conditions.
 - When the return statement is executed.
 - When it reaches the method ending closing curly brace.
 - When it throws an exception that is not handled in the called method.



Functions/Methods

Parameters and Arguments

- ▶ A method can take any number of parameters, and each parameter is of a specific data type.
- ▶ The values that the caller supplies for parameters are called the arguments; every argument must correspond to a particular parameter.

```
class Program
{
    static int Add(int a, int b)
    {
        int sum = a + b;
        return sum;
    }

    static void Main(string[] args)
    {
        int x, y;
        x = 10;
        y = 15;

        int sum = Add(x, y);

        Console.WriteLine($"Sum is {sum}");
        Console.ReadKey();
    }
}
```

Formal Parameters

Actual Parameters

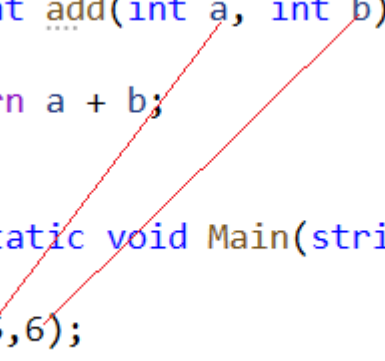
Functions/Methods

Named parameters

- ▶ Named parameters allow programmers to pass values to the parameters of a method by referring to the names of the parameters.
- ▶ You may or may not pass the parameters in the order in which they are defined in the method signature. Here is an example of how to use named parameters in C#:

```
static int add(int a, int b)
{
    return a + b;
}

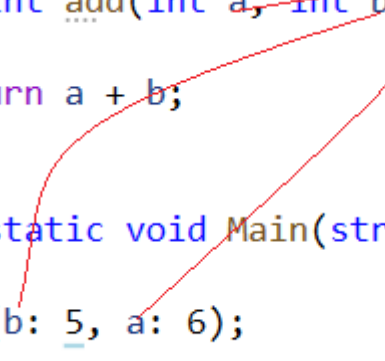
public static void Main(string[] args)
{
    add(5,6);
}
```



Without Named parameters

```
static int add(int a, int b)
{
    return a + b;
}

public static void Main(string[] args)
{
    add(b: 5, a: 6);
}
```



With Named parameters

Functions/Methods

Parameter Types

- ▶ Value Parameters
- ▶ Reference parameters
 - Reference Parameters (ref)
 - Output Parameters (out)
 - Read-Only Pass by Reference (in)
 - Return by Reference
- ▶ You apply one of the following modifiers to a parameter declaration to pass arguments by reference instead of by value:
 - **ref**: The argument must be initialized before calling the method. The method can assign a new value to the parameter but isn't required to do so.
 - **out**: The calling method isn't required to initialize the argument before calling the method. The method must assign a value to the parameter.
 - **readonly ref**: The argument must be initialized before calling the method. The method can't assign a new value to the parameter.
 - **in**: The argument must be initialized before calling the method. The method can't assign a new value to the parameter. The compiler might create a temporary variable to hold a copy of the argument to in parameters.

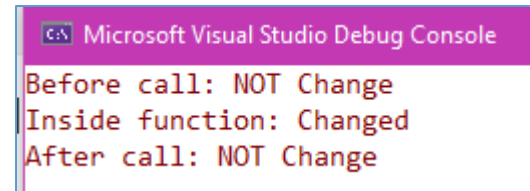
Functions/Methods

Parameter Types - Value Parameters

- Arguments to method calls are usually passed by value, which means the value of the argument expression is copied into the target parameter.

```
static void taker(string value)
{
    value = "Changed";
    Console.WriteLine("Inside function: " + value);
}

public static void Main(string[] args)
{
    string value = "NOT Change";
    Console.WriteLine("Before call: " + value);
    taker(value);
    Console.WriteLine("After call: " + value);
}
```



Microsoft Visual Studio Debug Console

```
Before call: NOT Change
Inside function: Changed
After call: NOT Change
```

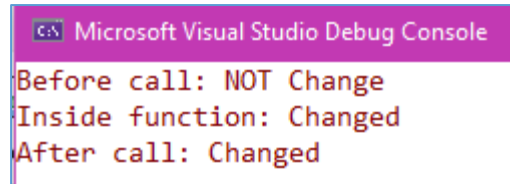
Functions/Methods

Parameter Types - Reference Parameters (ref)

- ▶ The ref modifier assigns a parameter to refer to an existing variable on the stack rather than creating a new variable and copying the argument value into the parameters.

```
static void taker(ref string value)
{
    value = "Changed";
    Console.WriteLine("Inside function: " + value);
}

public static void Main(string[] args)
{
    string value = "NOT Change";
    Console.WriteLine("Before call: " + value);
    taker(ref value);
    Console.WriteLine("After call: " + value);
}
```



Microsoft Visual Studio Debug Console

```
Before call: NOT Change
Inside function: Changed
After call: Changed
```

Functions/Methods

Parameter Types - Output Parameters (out)

```
static void taker(out string value)
{
    value = "Changed";
    Console.WriteLine("Inside function: " + value);
}

public static void Main(string[] args)
{
    string value;
    //Console.WriteLine("Before call: " + value); //error
    taker(out value);
    Console.WriteLine("After call: " + value);
}
```

Microsoft Visual Studio Debug Console

Inside function: Changed
After call: Changed

Functions/Methods

Parameter Types - Read-Only Pass by Reference (in)

- ▶ In C# 7.2, support was added for passing a value type by reference that was read only.
- ▶ Rather than passing the value type to a function so that it could be changed, read-only pass by reference was added: It allows the value type to be passed by reference so that not only copy of the value type occurs but, in addition, the invoked method cannot change the value.
- ▶ In other words, the purpose of the feature is to reduce the memory copied when passing a value while still identifying it as read only, thus improving the performance.
- ▶ This syntax is to add an in modifier to the parameter.

```
//static void taker(in string value)
static void taker(ref readonly string value)
{
    value = "Changed"; //Error
    Console.WriteLine("Inside function: " + value);
}
```

Functions/Methods

Parameter Types - Return by Reference

```
static void Main(string[] args)
{
    // Create an array of author names
    string[] authors = { "Mahesh Chand", "Mike Gold", "Dave McCarter", "Allen O'Neill", "Raj Kumar" };

    // Call a method that returns by ref
    ref string author4 = ref FindAuthor(3, authors);

    Console.WriteLine("Original author:{0}", author4);
    // Prints 4th author in array = Allen O'Neill
    Console.WriteLine();
    // Replace 4th author by new author. By Ref, it will update the array
    author4 = "Chris Sells";
    // Print 4th author in array
    Console.WriteLine("Replaced author:{0}", authors[3]);
    // Prints Chris Sells
    Console.ReadKey();
}

static ref string FindAuthor(int number, string[] names)
{
    if (names.Length > 0)
        return ref names[number]; // return the storage location, not the value
    throw new IndexOutOfRangeException($"{nameof(number)} not found.");
}
```


Functions/Methods

Method Overloading

- ▶ A class can comprise many methods with the same name with a unique signature → Method overloading.
 - Method signature includes:
 - parameter data types,
 - number of parameters,
 - order of parameters
- ▶ Method overloading is a type of operational polymorphism.

```
static float solve(int a, int b) ...  
static float solve(int a, int b, float c) ...  
static float solve(int a, float b) ...  
static float solve(float a, int b) ...
```

Functions/Methods

Optional parameters

- ▶ Optional parameters allow the association of a parameter with a constant value as part of the function/method declaration; it is possible to call a function/method without passing an argument for every parameter.

```
static void Main(string[] args){  
    readFile(@"C:\cs\a");  
}  
  
static void readFile(string path, string extension = "cs"){  
    //...  
}
```

Guidelines

DO provide good defaults for all parameters where possible.

DO provide simple method overloads that have a small number of required parameters.

CONSIDER organizing overloads from the simplest to the most complex.

Functions/Methods

Variable arguments (VarArgs)

- ▶ VarArgs is a technique that allows you to pass varying numbers of arguments to a function/method.
 - Using params keyword while declare the parameter.
- ▶ NOTE: No other parameters are permitted after the params keyword in a method declaration, and only one params keyword is permitted in a method declaration.

```
int x = Sum(1, 2, 3, 4);  
int y = Sum(1);  
int z = Sum(1, 2, 3, 4, 5, 6, 7, 8);
```

```
static int Sum(params int[] pars)  
{  
    int sum = 0;  
    foreach (int i in pars)  
        sum += i;  
    return sum;  
}
```

Functions/Methods

Exercises

1. Write a Python function to find the maximum of three numbers.
2. Write a Python function to sum all the numbers in a list.
3. Write a Python program to reverse a string.
4. Write a Python function to calculate the factorial of a number (a non-negative integer). The function accepts the number as an argument.
5. Write a Python function that takes a number as a parameter and checks whether the number is prime or not.
6. Write a Python function to print
 1. all prime numbers that less than a number (enter prompt keyboard).
 2. the first N prime numbers
7. Write a Python function to check whether a number is "Perfect" or not. Then print all perfect number that less than 1000.
8. Write a Python function to check whether a string is a pangram or not.
 - ▶ *(Note : Pangrams are words or sentences containing every letter of the alphabet at least once. For example : "The quick brown fox jumps over the lazy dog"*

Basic Error Handling with Exceptions

Exception Handling

Introduction

- ▶ An exception is a problem that arises during the execution of a program. A C# exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.
- ▶ Exceptions provide a way to transfer control from one part of a program to another. C# exception handling is built upon four keywords: try, catch, finally, and throw.
 - **try** – A try block identifies a block of code for which particular exceptions is activated. It is followed by one or more catch blocks.
 - **catch** – A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
 - **finally** – The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.
 - **throw** – A program throws an exception when a problem shows up. This is done using a throw keyword.

Exception Handling

Syntax and Exception classes

► Syntax

```
try
{
    // statements causing exception
}
catch (ExceptionName e1) {
    // error handling code
}
catch (ExceptionName e2){
    // error handling code
}
catch (ExceptionName eN){
    // error handling code
}
finally{
    // statements to be executed
}
```

No.	Exception Class & Description
1	System.IO.IOException Handles I/O errors.
2	System.IndexOutOfRangeException Handles errors generated when a method refers to an array index out of range.
3	System.ArrayTypeMismatchException Handles errors generated when type is mismatched with the array type.
4	System.NullReferenceException Handles errors generated from referencing a null object.
5	System.DivideByZeroException Handles errors generated from dividing a dividend with zero.
6	System.InvalidCastException Handles errors generated during typecasting.
7	System.OutOfMemoryException Handles errors generated from insufficient free memory.
8	System.StackOverflowException Handles errors generated from stack overflow.

Exception Handling

Handling Exceptions

```
static double SafeDivision(double x, double y)
{
    if (y == 0)
        throw new DivideByZeroException();
    return x / y;
}

public static void Main()
{
    // Input for test purposes. Change the values to see
    // exception handling behavior.
    double a = 98, b = 0;
    double result;
    try
    {
        result = SafeDivision(a, b);
        Console.WriteLine("{0} divided by {1} = {2}", a, b, result);
    }
    catch (DivideByZeroException ex)
    {
        Console.WriteLine("Attempted divide by zero.");
        Console.WriteLine(ex.StackTrace);
    }
}
```




*Thank you
for Listening!*

