Methods in C#

From zero to hero

By: Zahraa Ibrahim

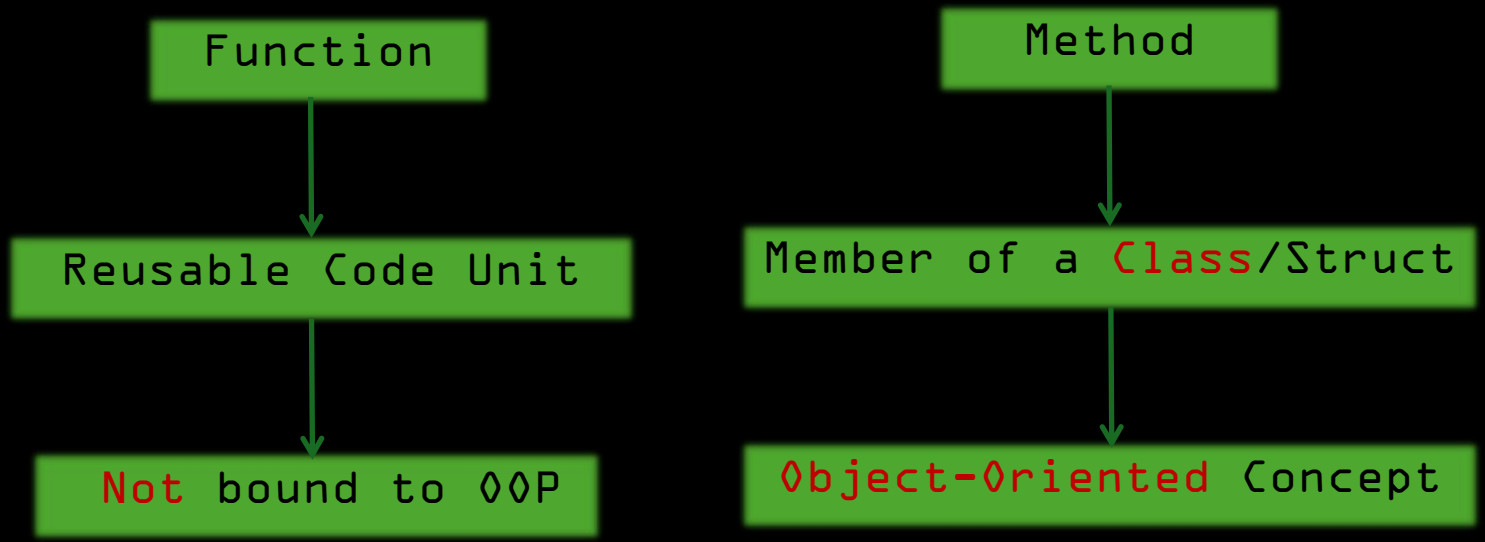
What's a Method???

- A method is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a method.
- Methods are used to perform certain actions, and they are also known as functions.

Method VS Functions

Method: A function that is associated with a class or object. In C#, all functions are methods because they exist inside classes.

Function: A reusable block of code that performs a task and can return a result. It may or may not belong to a class.

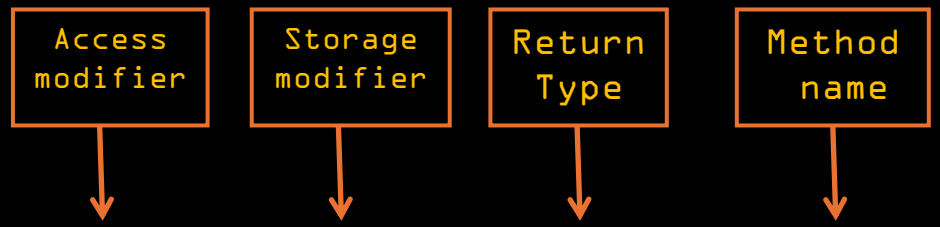


WHY METHODS???

To reuse code: define the code once, and use it many times.

Creat a method

A method is defined with the name of the method, followed by parentheses (). C# provides some pre-defined methods, which we already are familiar with, such as Main(), but we can also create our own methods like this:




```
public static void MyMethod()
{
    // code to be executed
    Console.WriteLine("Hi! I am your void Method :)\n");
}
```

Access modifier

Access modifiers define how visible or accessible a class, method, or variable is from other parts of the program.

Modifier	Description
public	Accessible from anywhere in the code (any class, any project).
private	Accessible only within the same class. This is the default for members.
protected	Accessible within the class and its subclasses (inheritance).
internal	Accessible only within the same assembly/project.
protected internal	Accessible within the same assembly or subclasses.
private protected	Accessible within the same class or subclasses in the same assembly.

Storage modifier

Storage modifiers (also called storage class specifiers) are keywords that determine how a method or variable is stored, accessed, and used in memory.

Modifier	Meaning	Applies To
static	Belongs to the class, not instances	Methods, fields, etc.
const	Compile-time constant — must be initialized immediately	Fields only
readonly	Value can be assigned only in constructor or declaration	Fields only
volatile	Field can be modified by multiple threads simultaneously	Fields only
extern	The method is implemented externally (e.g., in unmanaged code)	Methods only

Return Type

A return type in C# defines what kind of value a method sends back to whoever called it.

Return Type	Description	Example Use Case
void	Returns nothing	Printing, Logging
Primitive types	int, double, bool, char, etc.	Calculations, Conditions
Object types	Any class or interface type	Returning a Person or List
string	Textual return	Greetings, file paths
array types	Return multiple values of the same type	Returning a list of scores
Tuple	Return multiple values (different types)	Summary results
dynamic	Return anything, type checked at runtime	Flexible scenarios, scripting
Task, Task<T>	Asynchronous return values (used with async)	Web APIs, file operations
ref, out	Special keywords to return values via parameters	Advanced memory-efficient scenarios

Method name

The name of the Method.
For more Please check Rabab'ı lecture on
"Naming Conventions and operator"

Signature of a method:

- Name
- Number and Type of the Parameters

```
public void Addition(int x, int y) {...}

// inside the main():
public void Addition(1, 2) {...}
```

NOTE: When a parameter is passed to the method, it is called an argument. So, from the example above: x, y are parameters, while 1, 2 are arguments.

Default Parameter Value

Also Known as "optional parameter", You can also use a default parameter value, by using the equals sign (=).

```
static void MyMethod(string country = "Iraq") {
    Console.WriteLine(country); }

    static void Main(string[] args) {
      MyMethod("Sweden"); // Prints Sweden
      MyMethod("Japan");
      MyMethod(); // Prints Iraq
      MyMethod("USA"); }
```

Named Arguments

It is also possible to send arguments with the key: value syntax.
That way, the order of the arguments does not matter:

```
static void MyMethod(string child1, string child2, string child3) {
  Console.WriteLine("The youngest child is: " + child3); }

static void Main(string[] args)
{
  MyMethod(child3: "John", child1: "Liam", child2: "zeo");
}
// The youngest child is: John
```

Method Overloading:

Overloading is having a method with the same name but different parameters.

```
// 1. No parameters
0 references
public void Greet()
{
    Console.WriteLine("Hello!");
}

// 2. One string parameter
0 references
public void Greet(string name)
{
    Console.WriteLine($"Hello, {name}!");
}

// 3. Two parameters: string and int
0 references
public void Greet(string name, int age)
{
    Console.WriteLine($"Hello, {name}. You are {age} years old.");
}
```

A method with varying number of parameters:

```
public int Addition(int x, int y)
public int Addition(int x, int y, int z)
public int Addition(int x, int y, int z, int k)
```

instead use:

```
public int Addition(int[] numbers)
```

example:

```
var result = Methods.Addition(new int[] {1,2,3,4})
```

2D Arrays as Parameters

```
public static void Print2D(int[,] matrix) {
    for (int i = 0; i < matrix.GetLength(0); i++)      // Rows
    {
        for (int j = 0; j < matrix.GetLength(1); j++)  // Columns
            Console.Write(matrix[i, j] + " ");
        Console.WriteLine();
    }
}

// Calling the Method
int[,] grid = {
    {1, 2},
    {3, 4} };

Print2D(grid);
```

Parameter Modifiers in C#

C# gives us 3 special keywords that modify how parameters are passed to methods:

Modifier	Purpose	Can Skip?	Number of Parameters
params	Send a variable number of arguments as an array	Yes ✓	0 or more
ref	Pass by reference (must be initialized)	No ✗	Fixed
out	Pass by reference, and method must assign a value	No ✗	Fixed

Params modifier:

```
public int Addition(params int[] numbers)

var result = Methods.Addition(new int[] {1,2,3,4})

var result = Methods.Addition(1,2,3,4)
```


NOTES:

- You can only have one params parameter, and it must be the last.
- It's automatically treated like an array inside the method.

Ref modifier:

```
public static int SequreValue(ref int number)
{
    return number *= number;
}
// tha main block
int x = 10;
SequreValue(ref x);
Console.WriteLine(x); // Output: 100
```

Notes:

- The variable must be initialized before calling the method.
- Both caller and callee share the same memory reference.

Out modifier:

```
public static void SplitName(string fullName, out
string firstName, out string lastName)
{
    var parts = fullName.Split(' ');
    firstName = parts[0];
    lastName = parts.Length > 1 ? parts[1] : "";
}
// main()
string first, last;
SplitName("Ed Sheeran", out first, out last);
Console.WriteLine(first); // Ed
Console.WriteLine(last);  // Sheeran
```

Recursion

Recursion is when a method calls itself to solve a smaller version of the same problem.

Example of Recursion

```
public static int Sum(int n)
{
    if (n <= 0)
        return 0;

    else
        return n + Sum(n - 1);
}
```

Recursion Types

1. Tail Recursion: When the call happens at the last statement of the function.

2. Head Recursion: When the call happens at the First statement of the function.

3. tree Recursion: A recursive function that calls itself more than once.

4. Indirect Recursion: when two or more functions (or methods) call each other in a cycle (Sequence)

5. Nested Recursion: when a function calls itself with a recursive call as an argument.

- For examples and more check THIS out!
- for simpler examples here's my personal notes

Tasks

- Write a method named Calculate that takes two numbers and a string representing an operator ("+", "-", "*", "/") and returns the result.
  example:
  ```
  Console.WriteLine(Calculate(5, 3, "+")); // Output: 8
  ```

- Write a method to find the factorial of n numbers (use recursion)
  example:
  ```
  int n = 5;
  fact(n); // 120
  ```

- Write a method called MultiplyAllByTwo that takes an int[] array as a parameter and modifies each element by multiplying it by 2.
  example:
  ```
  int[] numbers = {1, 2, 3};
  MultiplyAllByTwo(numbers);
  // numbers is now {2, 4, 6}
  ```

You can find the compelete code in here

Thank you.