

C# Methods

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**.

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

Create a Method

A method is defined with the name of the method, followed by parentheses **()**. C# provides some pre-defined methods, which you already are familiar with, such as `Main()`, but you can also create your own methods to perform certain actions:

Example

Create a method inside the Program class:

```
class Program
{
    static void MyMethod()
    {
        // code to be executed
    }
}
```

Example Explained

- `MyMethod()` is the name of the method
- `static` means that the method belongs to the Program class and not an object of the Program class. You will learn more about objects and how to access methods through objects later in this tutorial.

- `void` means that this method does not have a return value. You will learn more about return values later in this chapter

Note: In C#, it is good practice to start with an uppercase letter when naming methods, as it makes the code easier to read.

Call a Method

To call (execute) a method, write the method's name followed by two parentheses `()` and a semicolon;

In the following example, `MyMethod()` is used to print a text (the action), when it is called:

Example

Inside `Main()`, call the `myMethod()` method:

```
static void MyMethod()
{
    Console.WriteLine("I just got executed!");
}

static void Main(string[] args)
{
    MyMethod();
}

// Outputs "I just got executed!"
```

Exercises (try it)

```
using System;

namespace MyApplication
{
    class Program
    {
```

```
static void MyMethod()  
{  
    Console.WriteLine("I just got executed!");  
}  
  
static void Main(string[] args)  
{  
    MyMethod();  
}  
}
```

A method can be called multiple times:

Example

```
static void MyMethod()  
{  
    Console.WriteLine("I just got executed!");  
}  
  
static void Main(string[] args)  
{  
    MyMethod();  
    MyMethod();  
    MyMethod();  
}  
  
// I just got executed!  
// I just got executed!  
// I just got executed!
```

Exercise (try it)

```
using System;
```

```
namespace MyApplication
{
    class Program
    {
        static void MyMethod()
        {
            Console.WriteLine("I just got executed!");
        }

        static void Main(string[] args)
        {
            MyMethod();
            MyMethod();
            MyMethod();
        }
    }
}
```

C# Method Parameters

Parameters and Arguments

Information can be passed to methods as parameter. Parameters act as variables inside the method.

They are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

The following example has a method that takes a `string` called **fname** as parameter. When the method is called, we pass along a first name, which is used inside the method to print the full name:

Example

```
static void MyMethod(string fname)
{
    Console.WriteLine(fname + " Refsnes");
}

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

```
{  
  
    MyMethod("Liam");  
  
    MyMethod("Jenny");  
  
    MyMethod("Anja");  
  
}  
  
// Liam Refsnes  
// Jenny Refsnes  
// Anja Refsnes
```

Exercise (try it)

```
using System;  
  
namespace MyApplication  
{  
    class Program  
    {  
        static void MyMethod(string fname)  
        {  
            Console.WriteLine(fname + " Refsnes");  
        }  
  
        static void Main(string[] args)  
        {  
            MyMethod("Liam");  
            MyMethod("Jenny");  
            MyMethod("Anja");  
        }  
    }  
}
```

When a **parameter** is passed to the method, it is called an **argument**. So, from the example above: `fname` is a **parameter**, while `Liam`, `Jenny` and `Anja` are **arguments**.

Default Parameter Value

You can also use a default parameter value, by using the equals sign (=). If we call the method without an argument, it uses the default value ("Norway"):

Example

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

static void Main(string[] args)
{
    MyMethod("Sweden");
    MyMethod("India");
    MyMethod();
    MyMethod("USA");
}

// Sweden
// India
// Norway
// USA
```

Exercise (try it)

```
using System;

namespace MyApplication
{
    class Program
```

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

    static void Main(string[] args)
    {
        MyMethod("Sweden");
        MyMethod("India");
        MyMethod();
        MyMethod("USA");
    }
}
```

A parameter with a default value, is often known as an **"optional parameter"**. From the example above, `country` is an optional parameter and `"Norway"` is the default value.

Multiple Parameters

You can have as many parameters as you like:

Example

```
static void MyMethod(string fname, int age)
{
    Console.WriteLine(fname + " is " + age);
}

static void Main(string[] args)
{
    MyMethod("Liam", 5);
    MyMethod("Jenny", 8);
    MyMethod("Anja", 31);
}
```

```
// Liam is 5  
  
// Jenny is 8  
  
// Anja is 31
```

Exercise (try it)

```
using System;  
  
namespace MyApplication  
{  
    class Program  
    {  
        static void MyMethod(string fname, int age)  
        {  
            Console.WriteLine(fname + " is " + age);  
        }  
  
        static void Main(string[] args)  
        {  
            MyMethod("Liam", 5);  
            MyMethod("Jenny", 8);  
            MyMethod("Anja", 31);  
        }  
    }  
}
```

Note that when you are working with multiple parameters, the method call must have the same number of arguments as there are parameters, and the arguments must be passed in the same order.

Return Values

The `void` keyword, used in the examples above, indicates that the method should not return a value. If you want the method to return a value, you can use a primitive data type (such as `int` or `double`) instead of `void`, and use the `return` keyword inside the method:

Example

```
static int MyMethod(int x)  
{
```

```
        return 5 + x;
    }

    static void Main(string[] args)
    {
        Console.WriteLine(MyMethod(3));
    }

    // Outputs 8 (5 + 3)
```

Exercise (try it)

```
using System;

namespace MyApplication
{
    class Program
    {
        static int MyMethod(int x)
        {
            return 5 + x;
        }

        static void Main(string[] args)
        {
            Console.WriteLine(MyMethod(3));
        }
    }
}
```

This example returns the sum of a method's **two parameters**:

Example

```
static int MyMethod(int x, int y)
{
    return x + y;
}
```

```
static void Main(string[] args)
{
    Console.WriteLine(MyMethod(5, 3));
}

// Outputs 8 (5 + 3)
```

Exercise (try it)

```
using System;

namespace MyApplication
{
    class Program
    {
        static int MyMethod(int x)
        {
            return 5 + x;
        }

        static void Main(string[] args)
        {
            Console.WriteLine(MyMethod(3));
        }
    }
}
```

You can also store the result in a variable (recommended, as it is easier to read and maintain):

Example

```
static int MyMethod(int x, int y)
{
    return x + y;
}

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

```
{  
  
    int z = MyMethod(5, 3);  
  
    Console.WriteLine(z);  
  
}  
  
// Outputs 8 (5 + 3)
```

Exercise (try it)

```
using System;  
  
namespace MyApplication  
{  
    class Program  
    {  
        static int MyMethod(int x, int y)  
        {  
            return x + y;  
        }  
  
        static void Main(string[] args)  
        {  
            int z = MyMethod(5, 3);  
            Console.WriteLine(z);  
        }  
    }  
}
```

Named Arguments

It is also possible to send arguments with the **key: value** syntax.

That way, the order of the arguments does not matter:

Example

```
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: "Liam");
}

// The youngest child is: John
```

Exercise (try it)

```
using System;

namespace MyApplication
{
    class Program
    {
        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: "Liam");
        }
    }
}
```

Named arguments are especially useful when you have multiple parameters with default values, and you only want to specify one of them when you call it:

Example

```
static void MyMethod(string child1 = "Liam", string child2 = "Jenny",
string child3 = "John")
{
    Console.WriteLine(child3);
}
```

```
static void Main(string[] args)
{
    MyMethod("child3");
}

// John
```

Exercise (try it)

```
using System;

namespace MyApplication
{
    class Program
    {
        static void MyMethod(string child1 = "Liam", string child2 =
"Jenny", string child3 = "John")
        {
            Console.WriteLine(child3);
        }

        static void Main(string[] args)
        {
            MyMethod("child3");
        }
    }
}
```

Method Overloading

With **method overloading**, multiple methods can have the same name with different parameters:

Example

```
int MyMethod(int x)

float MyMethod(float x)

double MyMethod(double x, double y)
```

Consider the following example, which have two methods that add numbers of different type:

```
static int PlusMethodInt(int x, int y)
{
    return x + y;
}

static double PlusMethodDouble(double x, double y)
{
    return x + y;
}

static void Main(string[] args)
{
    int myNum1 = PlusMethodInt(8, 5);
    double myNum2 = PlusMethodDouble(4.3, 6.26);
    Console.WriteLine("Int: " + myNum1);
    Console.WriteLine("Double: " + myNum2);
}
```

Exercise (try it)

```
using System;

namespace MyApplication
{
    class Program
    {
        static int PlusMethodInt(int x, int y)
        {
            return x + y;
        }

        static double PlusMethodDouble(double x, double y)
        {
```

```
        return x + y;
    }

    static void Main(string[] args)
    {
        int myNum1 = PlusMethodInt(8, 5);
        double myNum2 = PlusMethodDouble(4.3, 6.26);
        Console.WriteLine("Int: " + myNum1);
        Console.WriteLine("Double: " + myNum2);
    }
}
```

Instead of defining two methods that should do the same thing, it is better to overload one.

In the example below, we overload the `PlusMethod` method to work for both `int` and `double`:

Example

```
static int PlusMethod(int x, int y)
{
    return x + y;
}

static double PlusMethod(double x, double y)
{
    return x + y;
}

static void Main(string[] args)
{
    int myNum1 = PlusMethod(8, 5);
    double myNum2 = PlusMethod(4.3, 6.26);
}
```

```
Console.WriteLine("Int: " + myNum1);  
  
Console.WriteLine("Double: " + myNum2);  
  
}
```

Exercise (try it)

```
using System;  
  
namespace MyApplication  
{  
    class Program  
    {  
        static int PlusMethod(int x, int y)  
        {  
            return x + y;  
        }  
  
        static double PlusMethod(double x, double y)  
        {  
            return x + y;  
        }  
  
        static void Main(string[] args)  
        {  
            int myNum1 = PlusMethod(8, 5);  
            double myNum2 = PlusMethod(4.3, 6.26);  
            Console.WriteLine("Int: " + myNum1);  
            Console.WriteLine("Double: " + myNum2);  
        }  
    }  
}
```

Note: Multiple methods can have the same name as long as the number and/or type of parameters are different.

Reference

www.w3schools.com