METHODS

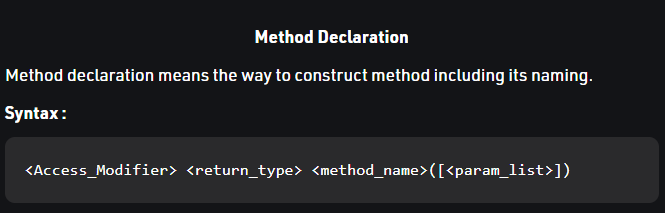
Methods are generally the block of codes or statements in a program that gives the user the ability to **reuse** the same code which ultimately saves the excessive use of memory, acts as a **time saver** and more importantly, it provides a better **readability** of code.

**Method Signature:** Method Signature is defined by mainly two parameters (number of parameters, type of the parameters and order of the parameters), One of them is **Method Name** and second one is its **Parameter list**.

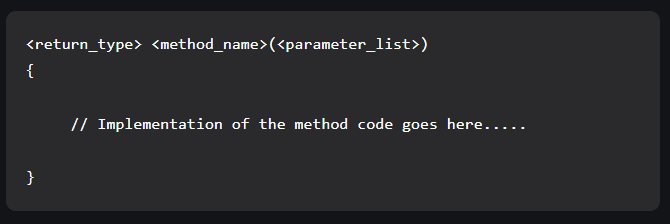
The first letter of the method name can be either a small letter or a Capital letter, however, **it is recommended to use the capital one**.

**Advantages of using the Methods :**   
There are many advantages of using methods. Some of them are listed below:

* It makes the program well structured.
* Methods enhance the readability of the code.
* It provides an effective way for the user to reuse the existing code.
* It optimizes the execution time and memory space.



**Method Body**



Method Overloading

***Method Overloading*** is the common way of implementing polymorphism.

It is the ability to redefine a function in more than one form.

A user can implement function overloading by defining two or more functions in a class sharing the same name.

C# can distinguish the methods with **different method signatures**. i.e. the methods can have the same name but with different parameters list (i.e. the number of the parameters, order of the parameters, and data types of the parameters) within the same class.

The **compiler does not consider the return type while differentiating the overloaded method**. But you cannot declare two methods with the same signature and different return type. It will throw a compile-time error. If both methods have the same parameter types, but different return type, then it is not possible.

**Different ways of doing overloading methods-**  
Method overloading can be done by changing:

1. The number of parameters in two methods.
2. The data types of the parameters of methods.
3. The Order of the parameters of methods.

Method Parameters

***C# contains the following types of Method Parameters:***

* **Named Parameters**

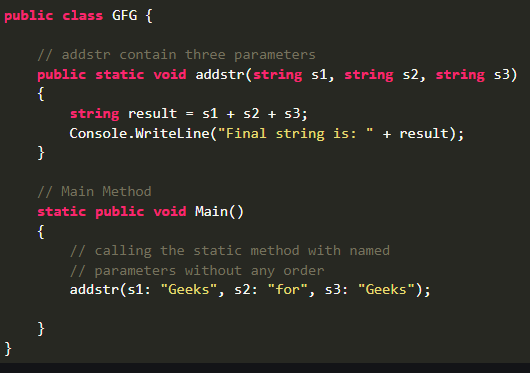
Using named parameters, you can specify the value of the parameter according to their names not their order in the method.

Or in other words, it provides us a facility to not remember parameters according to their order.

This concept is introduced in C# 4.0.

It makes your program easier to understand when you are working with a larger number of parameters in your method.

But always remember **named parameters are always appear after fixed arguments**, if you try to provide fixed argument after named parameter, then the compiler will throw an error.



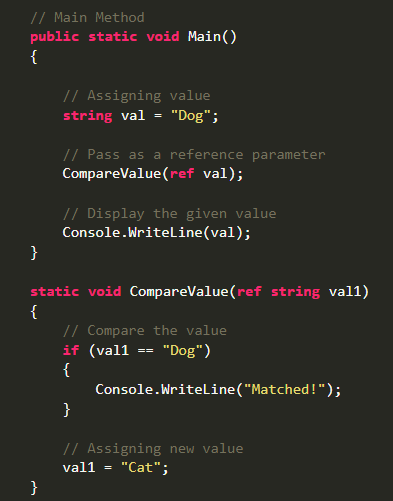
* **Ref Parameters**

The ref is a keyword in C# which is used for **passing the value types by reference**.

Or we can say that if any changes made in this argument in the method will reflect in that variable when the control return to the calling method.

The ref parameter **does not pass the property**.

In ref parameters, **it is necessary that the parameters should initialize before it pass to ref**.



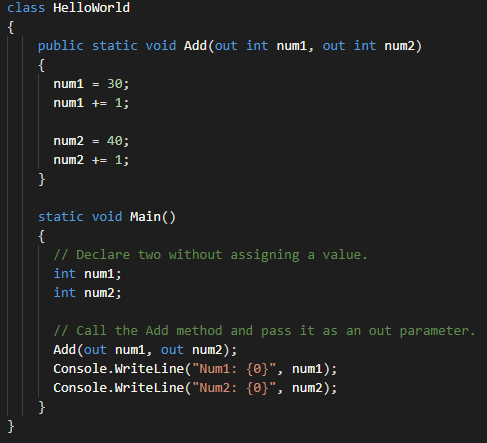
* **Out Parameters**

The **out** parameter in C# is used to pass arguments to methods [by reference](https://www.educative.io/edpresso/pass-by-value-vs-pass-by-reference).

It differs from the ref keyword in that it does not require parameter variables to be initialized before they are passed to a method.

The out keyword must be explicitly declared in the method’s definition​ as well as in the calling method.

The out parameter **does not pass the property**.



* **Default or Optional Parameters**

As the name suggests optional parameters are not compulsory parameters, they are optional.

It helps to exclude arguments for some parameters.

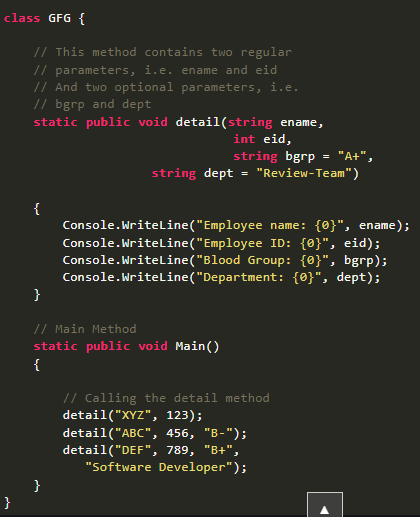
Or we can say in optional parameters, it is not necessary to pass all the parameters in the method.

This concept is introduced in C# 4.0.

Here, each and every optional parameter contains a default value which is the part of its definition.

If we do not pass any arguments to the optional parameters, then it takes its default value.

The optional parameters are **always defined at the end of the parameter list**. Or in other words, the last parameter of the method, constructor, etc. is the optional parameter.

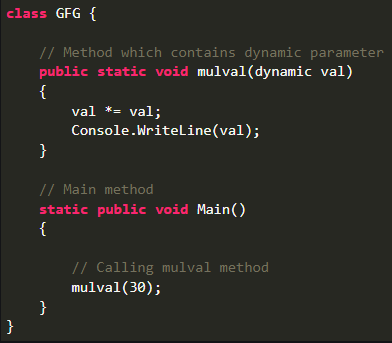


* **Dynamic Parameters**

In C# 4.0, a new type of parameters is introduced that is known as a dynamic parameter.

Here the parameters pass dynamically means the compiler does not check the type of the dynamic type variable at compile-time, instead of this, the compiler gets the type at the run time.

The dynamic type variable is created using a **dynamic** keyword.



* **Value Parameters (generally we use)**

It is a normal value parameter in a method or you can say the passing of value types by value.

So when the variables are passed as value type they contain the data or value, not any reference.

If you will make any changes in the value type parameter then it will not reflect the original value stored as an argument**.**

* **Params**

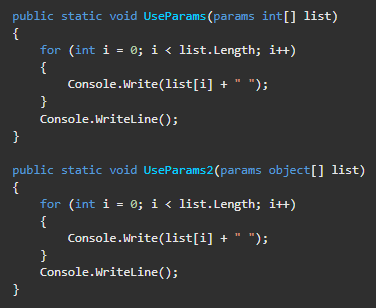
By using the params keyword, you can specify a method parameter that takes a variable number of arguments.

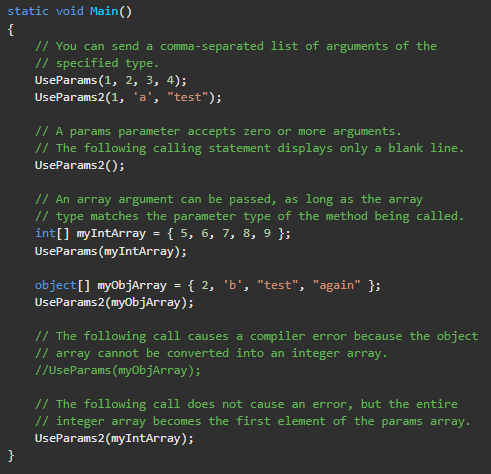
The parameter **type must be a single-dimensional array**.

**No additional parameters are permitted after the params keyword** in a method declaration, and only one params keyword is permitted in a method declaration

When you call a method with a params parameter, you can pass in:

* A comma-separated list of arguments of the type of the array elements.
* An array of arguments of the specified type.
* No arguments. If you send no arguments, the length of the params list is zero.

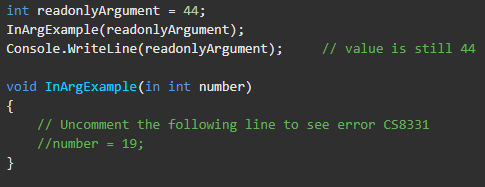




* **In Parameter**

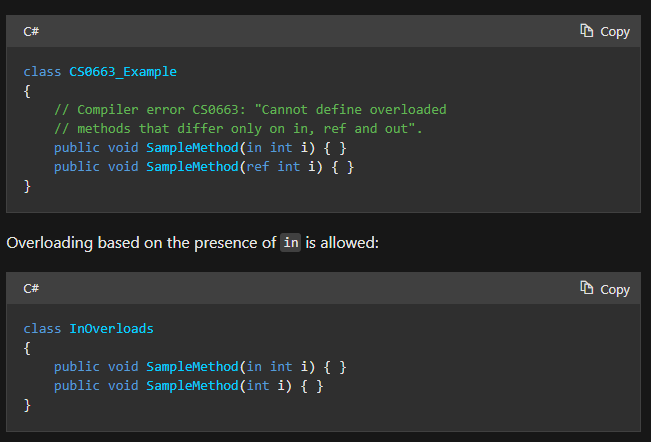
The in keyword causes arguments to be passed by reference but **ensures the** **argument is not modified**.

It is like the ref or out keywords, except that in arguments cannot be modified by the called method. Whereas ref arguments may be modified, out arguments must be modified by the called method, and those modifications are observable in the calling context.



Variables passed as in **arguments must be initialized before being passed** in a method call. However, the called method may not assign a value or modify the argument.

methods **cannot be overloaded** if the only difference is that one method takes a ref or out argument and the other takes an in argument.



**Method Overriding**

Method Overriding is a technique that allows the invoking of functions from another class (base class) in the derived class.

Creating a method in the derived class with the same signature as a method in the base class is called as method overriding.

When a method in a subclass has the same **name, same parameters or signature and same return type(or sub-type)** as a method in its super-class, then the method in the subclass is said to override the method in the super-class.

Method overriding is one of the ways by which C# achieve *Run Time Polymorphism (Dynamic Polymorphism)*.

An override method is a new implementation of a member that is inherited from a base class. The overridden base method must be **virtual, abstract, or override**.

**In C# we can use 3 types of keywords for Method Overriding:**

* **virtual keyword:** This modifier or keyword use within base class method. It is used to modify a method in *base class* for *overridden* that particular method in the derived class.
* **override:** This modifier or keyword use with derived class method. It is used to modify a *virtual* or *abstract* method into *derived class* which presents in base class.
* **base Keyword**: This is used to access members of the base class from derived class. It basically used to access constructors and methods or functions of the base class. The base keyword cannot use within a static method. Base keyword specifies which constructor of the base class should be invoked while creating the instances of the derived class.

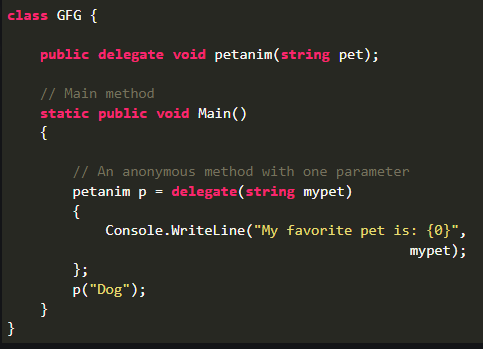
**Note:**

* Method overriding is possible only in derived classes. Because a method is overridden in the derived class from the base class.
* A non-virtual or a static method can’t be overridden.
* Both the override method and the virtual method must have the same access level modifier.

**Anonymous Method**

It is useful when the user wants to create an inline method and also wants to pass parameter in the anonymous method like other methods.

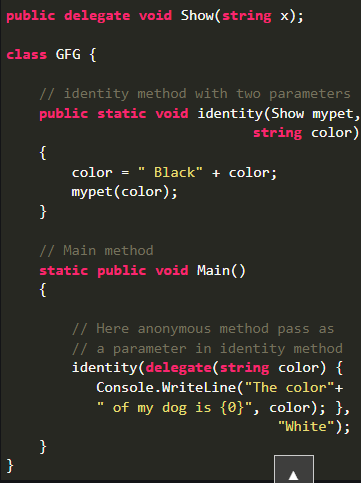
An Anonymous method is defined using the **delegate** keyword and the user can assign this method to a variable of the delegate type.



Declaring Delegate.

**Important Points:**

* This method is also known as *inline delegate*.
* Using this method you can create a delegate object without writing separate methods.
* This method can access variable present in the outer method. Such type of variables is known as *Outer variables*
* You can pass this method to another method which accepts delegate as a parameter. As shown in the below example:



Function which accept delegate

Declaring delegate

* An anonymous method does not contain jump statements like goto, break, or continue.
* An anonymous method does not access unsafe code.
* An anonymous method does not access in, ref, and out parameter of the outer scope.
* You can not use an anonymous method to the left side of the is operator.
* You can also use an anonymous method as an event handler.

**Example:**

