1.5 Methods in C#

C# Method

- 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.
- A method is a collection of statements that perform some specific task and return the result to the caller.
- A method can also perform some specific task without returning anything.

Declaring a Method

Methods are declared inside the body of a class, normally after the declaration of data field.

The general form of a method declaration is

modifiers type methodname(formal-parameter-list)

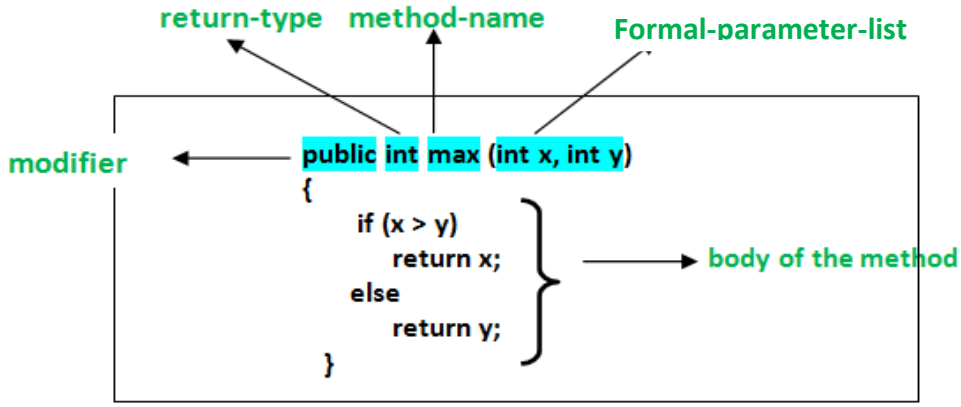
{
method-body

}

Method declaration has five parts:

- Name of the method (*methodname*)
- Type of value the method returns(*type*)
- List of parameters (*formal-parameter-list*)
- Body of the method
- Method modifiers(*modifiers*)

- ✓ The ***methodname*** is a valid identifier
- ✓ The ***type*** specifies the type of value the method returns. It can be any simple datatype such as int, float, char, string etc. If a method does not return anything, we specify a return type void.
- ✓ The ***formal-parameter-list*** is always enclosed in parentheses. This list contains variable names and types of all the values we want to give to the method as input. In case where no input data are required, the declaration must still include an empty set of parentheses() after the method name.
- ✓ The modifier specify keyword that decide the nature of accessibility and the mode of application of the method



- ✓ The formal parameter should be declared for their types individually.

int add(int x, y) //invalid int add(int x, int y) //valid

List of Access modifiers

public	The method can be accessed from anywhere, including outside the class
private	The method can only be accessed from inside the class to which it belongs
protected	The method can only be accessed from within the class to which it belongs or a type derived from the class
Static	The method does not operate on a specific instance of the class

Invoking a method

Once methods have been defined, they must be activated for operations. The process of activating a method is known as ***invoking*** or ***calling***

The invoking is done using dot operator

objectname.methodname(actual-parameter-list)

- ***objectname*** is the name of the object on which we are calling the method ***methodname***
- The ***actual-parameter-list*** is a comma separated list of 'actual values' that must match in type, order and number with formal parameter list of the methodname declared in the class.
- The values of the actual parameters are assigned to the formal parameters at the time of invocation.

Example:

```
 class Program
{
int Sum(int x, int y)  // formal-parameter-list
    {
     int return=x+y;
     return result;
     }
     static void Main(string[] args)
    {
      Program P= new Program();
        int a = 12;
        int b = 23;
        int c = P.Sum(a, b);    // actual-parameter-list
       Console.WriteLine("The Value of the sum is " + c);
    }
}
```

Output

```
The Value of the sum is 35
```

Nesting of method

A method can be called using only its name by another method of the same class. This is known as ***nesting of method.***

```
Class Nesting
{
void Largest (int m, int n)
  {
        int large = Max(m,n);
        Console.WriteLine(large);
  }
int Max(int a, int b)
  {
       int x=( a > b ) ? a : b ;
       return x;
  }
public static void Main()
{
        Nesting N = new Nesting()
        N.Largest(100,200);
  }
}
```

Output

```
200
```

Method Parameter

The invocation involves not only passing the values into the method but also getting back the results from the method. For managing the process of passing values and getting back the result. C# employs four kinds of parameter

- Value parameter
- Reference parameter
- Output parameter
- Parameter arrays

Value parameter

- Value parameter are used for passing parameters into method by value.
- When the method is invoked, the value of actual parameter are assigned to the corresponding formal parameters.
- If any changes made in the value type parameter it will not reflect the original data stored as argument.

Example

```
class Program
{
int Sum(int x, int y)
    {
     int return=x+y;
     return result;
     }
     static void Main(string[] args)
    {
        Program P= new Program();
        int c = P.Sum(12,23);
       Console.WriteLine("The Value of the sum is " + c);
    }
}
```

Output

```
The Value of the sum is 35
```

Reference parameter

- This method copies the reference to the memory location of an argument into the formal parameter. This means that changes made to the parameter affect the argument.
- The **ref** keyword in C# is used for passing or returning references of values to or from Methods.
- When a formal parameter is declared as ref , the corresponding actual parameter in the calling method must also be declared as ref.

Example

```
class Program
{
int Sum(ref int x, ref int y)
    {
     int return=x+y;
     return result;
     }
     static void Main(string[] args)
    {
      Program P= new Program();
       int a=12;
       int b=23
       int c = P.Sum(ref a, ref b);
```

```
      Console.WriteLine("The Value of the sum is " + c);
   }
}
```

Output

```
The Value of the sum is 35
```

Output parameter

- Output parameter are used to pass results back to the calling method.
- It is generally used when a method returns multiple values.
- This achieved by declaring the parameters with an **out** keyword.
- Similar to a reference parameter , an output parameter does not create a new storage location. Instead , it becomes an alias to the parameter in the calling method.
- When a formal parameter is declared as out , the corresponding actual parameter in the calling method must also be declared as out.

Example

```
class Program
{
 int display( int x, out int y)
   {

      y=x+10;
      return y;
    }
    static void Main(string[] args)
   {
       Program P= new Program();
       int a=10;
       int b;
     P.display(a, out b);
      Console.WriteLine("The Value of b"+b);
   }
}
```

Output

```
The Value of b 20
```

Parameter Array

- The "params" keyword in C# allows a method to accept a variable number of arguments. C# params works as an array of objects. By using params keyword in a method argument definition, we can pass a number of arguments.

- Note: There can't be anymore parameters after a params.

Example

```
class Program
{
 static void display(params int [] arr)
    {
     Console.WriteLine("Array elements are : ");
     foreach( int a in arr)
         {
        Console.WriteLine(a);
         }
     }
     static void Main(string[] args)
    {

        int [] x= {1,2,3,4,5};
        display(x);
        display();
        display(100,200);

    }
}
```

Output

```
Array elements are : 1 2 3 4 5

Array elements are :

Array elements are : 100, 200
```

Note that invocation **display()** and **display(100,200)** are equivalent to

display(new int[] {})

display(new int[] {100,200})

Method Overloading

Two or more methods in a class with the same name but different numbers, types, and order of parameters, it is called method overloading.

Example

```
using System;
namespace MethodOverload
{
class Program
 {
     int add(int a, int b)
      {
       int c=a+b;
         return c;
     }
   double add(int a, double b)
    {
       double c=a+b;
        return c;
   }
    string add(string a, string b)
    {
      string c=a+b;
      return c;
   }
 static void Main(string[] args)
{
 Program p1 = new Program();
      int c1=p1.add(100,200);
     double c2=  p1.add(100,2.5);
     string c3= p1.add("Hello","World");
Console.WriteLine(c1);
Console.WriteLine(c2);
Console.WriteLine(c3);

}

}

}
```

Output

```
Arguments: 100

Arguments: 100 and 200
```