Methods

A ***method*** is a member that implements a computation or action that can be performed by an object or class. ***Static methods*** are accessed through the class. ***Instance methods*** are accessed through instances of the class.

Methods may have a list of ***parameters***, which represent values or variable references passed to the method. Methods have a ***return type***, which specifies the type of the value computed and returned by the method. A method's return type is void if it doesn't return a value.

Like types, methods may also have a set of type parameters, for which type arguments must be specified when the method is called. Unlike types, the type arguments can often be inferred from the arguments of a method call and need not be explicitly given.

The ***signature*** of a method must be unique in the class in which the method is declared. The signature of a method consists of the **name of the method**, the **number of type parameters**, and the **number**, **modifiers**, and **types of its parameters**. The signature of a method **doesn't include the return type**.

Example:

```
public class Example
{
   public static void Main()
   {
      // Call with an int variable.
      int num = 4;
      int productA = Square(num);

      // Call with an integer literal.
      int productB = Square(12);

      // Call with an expression that evaluates to int.
      int productC = Square(productA * 3);
   }

   static int Square(int i)
   {
      // Store input argument in a local variable.
      int input = i;
      return input * input;
   }
}
```

Parameters

Parameters are used to pass values or variable references to methods. The parameters of a method get their actual values from the ***arguments*** that are specified when the method is invoked. There are four kinds of parameters: **value parameters**, **reference parameters**, **output parameters**, and **parameter arrays.**

A ***value parameter*** is used for passing input arguments. A value parameter corresponds to a local variable that gets its initial value from the argument that was passed for the parameter. Modifications to a value parameter don't affect the argument that was passed for the parameter.

Value parameters can be optional, by specifying a default value so that corresponding arguments can be omitted.

Example:

```
using System;

public class Options
{
   public void ExampleMethod(int required, int optionalInt = default(int),
                             string description = "Optional Description")
   {
      Console.WriteLine("{0}: {1} + {2} = {3}", description, required,
                        optionalInt, required + optionalInt);
   }
}

public class Example
{
   public static void Main()
   {
      var opt = new Options();
      opt.ExampleMethod(10);
      opt.ExampleMethod(10, 2);
      opt.ExampleMethod(12, description: "Addition with zero:");
   }
}
// The example displays the following output:
//      Optional Description: 10 + 0 = 10
//      Optional Description: 10 + 2 = 12
//      Addition with zero:: 12 + 0 = 12
```

A ***reference parameter*** is used for passing arguments by reference. The argument passed for a reference parameter must be a variable with a definite value. During execution of the method, the reference parameter represents the same storage location as the argument variable. A reference parameter is declared with the **ref** modifier. The following example shows the use of ref parameters:

Example:

```
static void Swap(ref int x, ref int y)
{
    int temp = x;
    x = y;
    y = temp;
}

public static void SwapExample()
{
```

```
    int i = 1, j = 2;
    Swap(ref i, ref j);
    Console.WriteLine($"{i} {j}");    // "2 1"
}
```

An ***output parameter*** is used for passing arguments by reference. It's similar to a reference parameter, except that it doesn't require that you explicitly assign a value to the caller-provided argument. An output parameter is declared with the out modifier. The following example shows the use of out parameters using the syntax introduced in C# 7.

Example:

```
static void Divide(int x, int y, out int result, out int remainder)
{
    result = x / y;
    remainder = x % y;
}

public static void OutUsage()
{
    Divide(10, 3, out int res, out int rem);
    Console.WriteLine($"{res} {rem}");      // "3 1"
}
```

A ***parameter array*** permits a variable number of arguments to be passed to a method. A parameter array is declared with the params modifier. Only the last parameter of a method can be a parameter array, and the type of a parameter array must be a single-dimensional array type. A caller can then invoke the method with parameter array in either of four ways: By passing an **array of the appropriate type** that contains the desired number of elements, By passing a **comma-separated list** of individual arguments of the appropriate type to the method, By passing **null**, By **not providing an argument** to the parameter array.

The Write and WriteLine methods of the System.Console class are good examples of parameter array usage. They're declared as follows:

Example:

```
public class Console
{
    public static void Write(string fmt, params object[] args) { }
    public static void WriteLine(string fmt, params object[] args) { }
    // ...
}

...

int x, y, z;
x = 3;
y = 4;
z = 5;
Console.WriteLine("x={0} y={1} z={2}", x, y, z);
```

Return values

Methods can return a value to the caller. If the return type is not void, the method can return the value by using the `return` keyword. A statement with the `return` keyword followed by a variable, constant, or expression that matches the return type will return that value to the method caller. Methods with a non-void return type are required to use the return keyword to return a value. The return keyword also stops the execution of the method.

If the return type is void, a return statement without a value is still useful to stop the execution of the method. Without the return keyword, the method will stop executing when it reaches the end of the code block.

Sometimes, you want your method to **return more than a single value**. Starting with C# 7.0, you can do this easily by using tuple types and tuple literals. The tuple type defines the data types of the tuple's elements. Tuple literals provide the actual values of the returned tuple. In the following example, `(string, string, string, int)` defines the tuple type that is returned by the `GetPersonalInfo` method. The expression `(per.FirstName, per.MiddleName, per.LastName, per.Age)` is the tuple literal; the method returns the first, middle, and last name, along with the age, of a `PersonInfo` object.

Example:

```
public (string FName, string MName, string LName, int Age) GetPersonalInfo(string
id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}

...

var person = GetPersonalInfo("111111111");
Console.WriteLine($"{person.FName} {person.LName}: age = {person.Age}");

...
```

Static and instance methods

A method declared with a `static` modifier is a *static method*. A static method doesn't operate on a specific instance and can only directly access static members.

A method declared without a `static` modifier is an *instance method*. An instance method operates on a specific instance and can access both static and instance members. The instance on which an instance method was invoked can be explicitly accessed as this. It's an error to refer to `this` in a static method.

Expression-bodied members

It is common to have method definitions that simply return immediately with the result of an expression, or that have a single statement as the body of the method. There is a syntax shortcut for defining such methods using =>:

```
public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public void Print() => Console.WriteLine(First + " " + Last);
```

Polymorphism

Polymorphism is often referred to as the third pillar of object-oriented programming, after encapsulation and inheritance. Polymorphism is a Greek word that means "many-shaped". When a method shows different behaviors when we passed different types and number values, then it is called polymorphism. So behaving in different ways depending on the input received is known as polymorphism i.e. whenever the input changes, automatically the output or the behavior also changes. There are two types of Polymorphism:

1. **Static polymorphism / Compile-time polymorphism / Early binding**: The object of class recognizes which method to be executed for a particular method call at the time of program compilation and binds the method call with method definition. This happens in case of **overloading** because in case of overloading each method will have a different signature and based on the method call we can easily recognize the method which matches the method signature. It is also called as static polymorphism or early binding. Static polymorphism is achieved by using **method overloading** and **operator overloading**
2. **Dynamic polymorphism / Run-time polymorphism / Late binding**: For a given method call, we can recognize which method has to be executed exactly at runtime but not in compilation time because in case of **overriding** we have multiple methods with the same signature. So, which method to be given preference and executed that is identified at Runtime and binds the method call with its suitable method. It is also called as dynamic polymorphism or late binding. Dynamic Polymorphism is achieved by using **method overriding**.

The polymorphism in C# can be implemented using the following three ways: Overloading, Overriding and Hiding.

Method overloading

The Method Overloading in C# allows a class to have multiple methods with the same name but with a different signature. So in C# functions or methods can be overloaded based on the number, type (int, float, etc), order and kind (Value, Ref or Out) of parameters.

It is also possible to overload the methods in the derived classes, it means, it allows us to create a method in the derived class with the same name as the method name defined in the base class. When compiling an invocation of an overloaded method, the compiler uses *overload resolution* to determine the specific method to invoke. Overload resolution finds the one method that best matches the arguments. If no single best match can be found, an error is reported.

The point that you need to keep in mind is the signature of a method does not include the return type and the params modifiers. So it is not possible to overload a method just based on the return type and params modifier.

If you want to execute the same logic but with different types of argument i.e. different types of values, then you need to overload the methods. For example, if you want to add two integers, two floats, and two strings, then you need to define three methods with the same name as shown in the below example.

Example:

```
namespace PolymorphismDemo
{
    class Program
    {
        public void add(int a, int b)
        {
            Console.WriteLine(a + b);
        }
        public void add(float x, float y)
        {
            Console.WriteLine(x + y);
        }
        public void add(string s1, string s2)
        {
            Console.WriteLine(s1 + s2);
        }
        static void Main(string[] args)
        {
            Program obj = new Program();
            obj.add(10, 20);
            obj.add(10.5f, 20.5f);
            obj.add("Dhaval", "Rangrej");
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }
}
// The example displays the following output:
//    30
//    31
//    DhavalRangrej
//    Press any key to exit.
```

Method Overriding (Virtual, override, and abstract methods)

The process of re-implementing the super class non-static method in the subclass with the same prototype (same signature defined in the super class) is called Method Overriding in C#. The implementation of the sub-class overrides (i.e. replaces) the implementation of super class methods.

The point that you need to keep in mind is the overriding method are always going to be executed from the current class object. Super class method is called the overridden method and sub-class method is called as the overriding method. If the super class method logic is not fulfilling the sub-class business requirements, then the subclass needs to override that method with required business logic. Usually, in

most of the real-time applications, the super class methods are implemented with generic logic which is common for all the next level sub-classes.

If you want to override a parent class method in its child class, first the method in the parent class must be declared as **virtual** by the using the keyword **virtual,** then only the child classes get the permission for overriding that method. Declaring the method as **virtual** is marking the method is overridable. If the child class wants to override the parent class virtual method then the child class can do it with the help of the **override** modifier. But overriding the method under child class is not mandatory for the child classes.

When an instance method declaration includes a virtual modifier, the method is said to be a *virtual method*. When no virtual modifier is present, the method is said to be a *nonvirtual method*.

An *abstract method* is a virtual method with no implementation. An abstract method is declared with the **abstract** modifier and is permitted only in an abstract class. An abstract method must be overridden in every non-abstract derived class.

Example:

```
namespace PolymorphismDemo
{
    class Class1
    {
        public virtual void show()
        {
            Console.WriteLine("Super class show method");
        }
    }
    class Class2 : Class1
    {
        public override void show()
        {
            base.show();
            Console.WriteLine("Sub class override show method");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Class2 obj = new Class2();
            obj.show();
            Console.ReadKey();
        }
    }
}
// The example displays the following output:
//    Super class show method
//    Sub class override show method
```

Once we re-implement the parent class methods under the child class, then the object of the child class calls its own methods but not its parent class method. But if you want to still consume or call the parent class's methods from the child class, then it can be done in two different ways.

By creating parent class object under the child class, we can call the parent class methods from child class or by using the **base** keyword, we can call parent class methods from child class, but **this** and **base** keyword cannot be used under the static block.

Method Hiding

When we use the **new** keyword to hide a base class member, then it is called as Method Hiding in C#. We will get a compiler warning if we miss the **new** keyword. This is also used for re-implementing a parent class method under child class.

In Method overriding parent class gives the permission for its child class to override the method by declaring it as **virtual**. Now the child class can override the method using the **override** keyword as it got permission from the parent. In Method hiding also parent class methods can be redefined under child classes even if they were not declared as Virtual by using '**new**' keyword.

In method overriding a base class reference variable pointing to a child class object will invoke the overridden method in the child class. In method hiding a base class reference variable pointing to a child class object will invoke the hidden method in the base class.

Example:

```
namespace PolymorphismDemo
{
    class Class1
    {
        public void display()
        {
            Console.WriteLine("Super class display method");
        }
    }
    class Class2 : Class1
    {
        public new void display()
        {
            Console.WriteLine("Sub class display method");
        }
    }
}
```

Extension Methods

It is a new feature that has been added in C# 3.0 which allows us to add new methods into a class without editing the source code of the class i.e. if a class consists of set of members in it and in the future if you want to add new methods into the class, you can add those methods without making any changes to the source code of the class.

Extension methods can be used as an approach of extending the functionality of a class in the future if the source code of the class is not available or we don't have any permission in making changes to the class. Before extension methods, inheritance is an approach that used for the extending the functionality of a class i.e. if we want to add any new members into an existing class without making a modification to the class, we will define a child class to that existing class and then we add new members in the child class.

Both these approaches can be used for extending the functionalities of an existing class whereas, in inheritance, we call the method defined in the old and new classes by using object of the new class whereas, in case of extension methods, we call the old and new methods by using object of the old class.

Points to Remember while working with C# Extension methods:

- Extension methods must be defined only under the **static** class.
- As **an extension method is defined** under a static class, compulsory that method should be defined **as static** whereas once the method is bound with another class, the method changes into non-static.
- The first parameter of an extension method is known as the **binding parameter** which should be the name of the **class to which the method has to be bound** and the binding parameter should be prefixed with **this** keyword.
- An extension method can have **only one binding parameter** and that should be defined in the first place of the parameter list.
- If required, an extension method can be defined with normal parameter also starting from the second place of the parameter list.

Example:

Let us see one real-time scenario where you can use the extension method. As we know `string` is a built-in class provided by .NET Framework. That means the source code of this class is not available to us and hence we can change the source code of `string` class. Now our requirement is to add a method to the `string` class i.e. `GetWordCount()` and that method will return the number of words present in a string and we should call this method as shown in the below image.

```
namespace ExtensionMethodsDemo
{
    public static class StringExtension
    {
        public static int GetWordCount(this string inputstring)
        {
            if (!string.IsNullOrEmpty(inputstring))
            {
                string[] strArray = inputstring.Split(' ');
                return strArray.Count();
            }
            else
            {
                return 0;
            }
```

```
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            string myWord = "SSM Infotech Solutions Pvt Ltd";
            int wordCount = myWord.GetWordCount();
            Console.WriteLine("string : " + myWord);
            Console.WriteLine("Count : " + wordCount);
            Console.Read();
        }
    }
}
// The example displays the following output:
//    string : SSM Infotech Solutions Pvt Ltd
//    Count : 5
```