Method Overloading and Method Overriding are both the techniques used to implement FUNCTIONAL POLYMORPHISM.
They enhance the methods functionality using their respective features.

Overloading:
1) It involves having another method with the same name in a class or its derived class.
2) Can be implemented with or without inheritance.
3) It is implemented by:
    a) changing the number of parameters in different methods with same name.
    b) Changing the parameter data types (if the number of parameters are same)
    c) changing parameter order.
4) Applicable to static as well as non static methods:
5) no keywords needed before the method names in C#.
6) Also called as COMPILE TIME Polymorphism.

example: int add(int a, int b)
{
return a+b;
}
int add(int a)
{
return a*5;
}

Overriding:

1) It involves having another method with the same name in a base class and derived class.
2) Always needs inheritance.
3)It is implemented by having two methods with same name, same signature, but different implementations (coding)in a base class and derived class
4)applicable to nonstatic, nonprivate methods only.access modifiers of the methods are not changed in the base and derived classes.
5) Virtual and override keywords are needed before the method names in base and derived classes.
For overriding in firther classes, override keyword will be used,
6)RUN TIME Polymorphism
example:
class A
{
public virtual void demo()
{

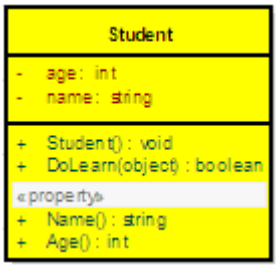
```
}
}
class B:A
{
public override void demo()
{

}
static void Main()
{
B obj=new B();
obj.demo();
}
}
```

Class:

It is a collection of objects.

Object:

It is a real time entity.
An object can be considered a "**thing**" that can perform a set of **related** activities. The set of activities that the object performs defines the object's behavior. For example, the hand can grip something or a **Student** (object) can give the name or address. In pure **OOP** terms an object is an instance of a class



The above template describe about object **Student**
Class is composed of three things name, attributes, and operations

```
public class student
{
}
student objstudent=new student ();
```

According to the above sample we can say that **Student** object, named **objstudent,** has created out of the student class.

In real world you will often find many individual objects all of the same kind. As an example, there may be thousands of other bicycles in existence, all of the same make and model. Each bicycle has built from the same blueprint. In object-oriented terms, we say that the bicycle is an instance of the class of objects known as bicycles. In the software world, though you may not have realized it, you have already used classes. For example, the **Textbox** control, you always used, is made out of the **Textbox** class, which defines its appearance and capabilities. Each time you drag a **Textbox** control, you are actually creating a new instance of the **Textbox** class.

Encapsulation:

Encapsulation is a process of binding the data members and member functions into a single unit.

Example for encapsulation is **class**. A class can contain data structures and methods. Consider the following class

```
public class Aperture
{
```

```
public Aperture ()
{
}
protected double height;
protected double width;
protected double thickness;
public double get volume()
{
Double volume=height * width * thickness;
if (volume<0)
return 0;
return volume;
}
}
```

In this example we encapsulate some data such as height, width, thickness and method Get Volume. Other methods or objects can interact with this object through methods that have public access modifier

Abstraction:

Abstraction is a process of hiding the implementation details and displaying the essential features.

Example1: A Laptop consists of many things such as processor, motherboard, RAM, keyboard, LCD screen, wireless antenna, web camera, usb ports, battery, speakers etc. To use it, you don't need to know how internally LCD screens, keyboard, web camera, battery, wireless antenna, speaker's works. You just need to know how to operate the laptop by switching it on. Think about if you would have to call to the engineer who knows all internal details of the laptop before operating it. This would have highly expensive as well as not easy to use everywhere by everyone.

So here the Laptop is an object that is designed to hide its complexity.
How to abstract: - By using **Access Specifiers**

.Net has five access Specifiers

Public -- Accessible outside the class through object reference.

Private -- Accessible inside the class only through member functions.

Protected -- Just like private but Accessible in derived classes also through member functions.

Internal -- Visible inside the assembly. Accessible through objects.

Protected Internal -- Visible inside the assembly through objects and in derived classes outside the assembly through member functions.

Let's try to understand by a practical example:-

```
public class Class1
   {
      int  i;                                  //No Access specifier means private
      public  int j;                                 // Public
      protected int k;                          //Protected data
      internal int m;                        // Internal means visible inside assembly
      protected internal int n;                  //inside assembly as well as to derived classes
outside assembly
      static int x;                             // This is also private
      public static int y;                      //Static means shared across objects
      [DllImport("MyDll.dll")]
      public static extern int MyFoo();       //extern means declared in this assembly defined
in some other assembly
      public void myFoo2()
      {
         //Within a class if you create an object of same class then you can access all data
members through object reference even private data too
         Class1 obj = new Class1();
         obj.i =10;   //Error can't access private data through object.But here it is
accessible.:)
         obj.j =10;
         obj.k=10;
         obj.m=10;
         obj.n=10;
     //     obj.s =10;  //Errror Static data can be accessed by class names only
         Class1.x = 10;
      //   obj.y = 10; //Errror Static data can be accessed by class names only
         Class1.y = 10;
      }
   }
```

Now lets try to copy the same code inside Main method and try to compile

```
[STAThread]
      static void Main()
      {
        //Access specifiers comes into picture only when you create object of class outside
the class
         Class1 obj = new Class1();
     //     obj.i =10; //Error can't access private data through object.
         obj.j =10;
    //      obj.k=10;     //Error can't access protected data through object.
         obj.m=10;
         obj.n=10;
     //     obj.s =10;  //Errror Static data can be accessed by class names only
         Class1.x = 10;  //Error can't access private data outside class
      //   obj.y = 10; //Errror Static data can be accessed by class names only
         Class1.y = 10;
      }
```

What if Main is inside another assembly

```
[STAThread]
      static void Main()
      {
```

```
        //Access specifiers comes into picture only when you create object of class outside
the class
            Class1 obj = new Class1();
        //     obj.i =10; //Error can't access private data through object.
            obj.j =10;
        //     obj.k=10;     //Error can't access protected data through object.
        //     obj.m=10; // Error can't access internal data outside assembly
        //     obj.n=10; // Error can't access internal data outside assembly

        //     obj.s =10;  //Errror Static data can be accessed by class names only
            Class1.x = 10;  //Error can't access private data outside class
          //   obj.y = 10; //Errror Static data can be accessed by class names only
            Class1.y = 10;
        }
```

In object-oriented software, complexity is managed by using **abstraction**.
Abstraction is a process that involves identifying the critical behavior of an object and eliminating irrelevant and complex details.

Inheritance:

Inheritance is a process of deriving the new class from already existing class
C# is a complete object oriented programming language. Inheritance is one of the primary concepts of object-oriented programming. It allows you to reuse existing code. Through effective use of inheritance, you can save lot of time in your programming and also reduce errors, which in turn will increase the quality of work and productivity. A simple example to understand inheritance in C#.

```
Using System;
Public class BaseClass
{
   Public BaseClass ()
   {
      Console.WriteLine ("Base Class Constructor executed");
   }

   Public void Write ()
   {
      Console.WriteLine ("Write method in Base Class executed");
   }
}

Public class ChildClass: BaseClass
{

   Public ChildClass ()
   {
      Console.WriteLine("Child Class Constructor executed");
   }

   Public static void Main ()
   {
```

```
        ChildClass CC = new ChildClass ();
        CC.Write ();
    }
}
```

In the Main () method in ChildClass we create an instance of childclass. Then we call the write () method. If you observe the ChildClass does not have a write() method in it. This write () method has been inherited from the parent BaseClass.

The output of the above program is

Output:

```
Base Class Constructor executed
Child Class Constructor executed
Write method in Base Class executed
```

this output proves that when we create an instance of a child class, the base class constructor will automatically be called before the child class constructor. So in general Base classes are automatically instantiated before derived classes.

In C# the syntax for specifying BaseClass and ChildClass relationship is shown below. The base class is specified by adding a colon, ":", after the derived class identifier and then specifying the base class name.

Syntax:

```
class ChildClassName: BaseClass
{
    //Body
}
```

C# supports single class inheritance only. What this means is, your class can inherit from only one base class at a time. In the code snippet below, class C is trying to inherit from Class A and B at the same time. This is not allowed in C#. This will lead to a compile time error: **Class 'C' cannot have multiple base classes: 'A' and 'B'**.

```
public class A
{
}
public class B
{
}
public class C : A, B
{
}
```

In C# Multi-Level inheritance is possible. Code snippet below demonstrates mlti-level inheritance. Class B is derived from Class A. Class C is derived from Class B. So class C, will have access to all members present in both Class A and Class B. As a result of multi-level inheritance Class has access to A_Method(),B_Method() and C_Method().

Note: Classes can inherit from multiple interfaces at the same time. **Interview Question:** How can you implement multiple inheritance in C#? **Ans :** Using Interfaces. We will talk about interfaces in our later article.

```
Using System;
Public class A
{
   Public void A_Method ()
   {
      Console.WriteLine ("Class A Method Called");
   }
}
Public class B: A
{
   Public void B_Method ()
   {
      Console.WriteLine ("Class A Method Called");
   }
}
Public class C: B
{
   Public void C_Method ()
   {
      Console.WriteLine ("Class A Method Called");
   }

   Public static void Main ()
   {
      C C1 = new C ();
      C1.A_Method ();
      C1.B_Method ();
      C1.C_Method ();
   }
}
```

When you derive a class from a base class, the derived class will inherit all members of the base class except constructors. In the code snippet below class B will inherit both M1 and M2 from Class A, but you cannot access M2 because of the private access modifier. Class members declared with a private access modifier can be accessed only with in the class. We will talk about access modifiers in our later article.

Common Interview Question: Are private class members inherited to the derived class?

Ans: Yes, the private members are also inherited in the derived class but we will not be able to access them. Trying to access a private base class member in the derived class will report a compile time error.

```
Using System;
Public class A
{
Public void M1 ()
{
}
Private void M2 ()
{
```

```
}
}

Public class B: A
{
Public static void Main ()
{
B B1 = new B ();
B1.M1 ();
//Error, Cannot access private member M2
//B1.M2 ();
}
}
```

Method Hiding and Inheritance We will look at an example of how to **hide** a method in C#. The Parent class has a write () method which is available to the child class. In the child class I have created a new write () method. So, now if I create an instance of child class and call the write () method, the child class write () method will be called. The child class is hiding the base class write () method. This is called method hiding.

If we want to call the parent class write () method, we would have to type cast the child object to Parent type and then call the write () method as shown in the code snippet below.

```
Using System;
Public class Parent
{
   Public void Write ()
   {
      Console.WriteLine ("Parent Class write method");
   }
}

Public class Child: Parent
{
   Public new void Write ()
   {
      Console.WriteLine ("Child Class write method");
   }

   Public static void Main ()
   {
      Child C1 = new Child ();
      C1.Write ();
      //Type caste C1 to be of type Parent and call Write () method
      ((Parent) C1).Write ();
   }
}
```

Polymorphism:

When a message can be processed in different ways is called polymorphism. Polymorphism means many forms.

Polymorphism is one of the fundamental concepts of OOP.

Polymorphism provides following features:

- It allows you to invoke methods of derived class through base class reference during runtime.
- It has the ability for classes to provide different implementations of methods that are called through the same name.

Polymorphism is of two types:

1. Compile time polymorphism/Overloading
2. Runtime polymorphism/Overriding

Compile Time Polymorphism

Compile time polymorphism is method and operators overloading. It is also called early binding.

In method overloading method performs the different task at the different input parameters.

Runtime Time Polymorphism

Runtime time polymorphism is done using inheritance and virtual functions. Method overriding is called runtime polymorphism. It is also called late binding.

When **overriding** a method, you change the behavior of the method for the derived class. **Overloading** a method simply involves having another method with the same prototype.

Caution: Don't confused method overloading with method overriding, they are different, unrelated concepts. But they sound similar.

Method overloading has nothing to do with inheritance or virtual methods.

Following are examples of methods having different overloads:

```
void area(int side);
void area(int l, int b);
void area(float radius);
```

Practical example of Method Overloading (Compile Time Polymorphism)

```
using System;

namespace method_overloading
{
   class Program
   {
```

```
        public class Print
        {

            public void display(string name)
            {
                Console.WriteLine ("Your name is : " + name);
            }

            public void display(int age, float marks)
            {
                Console.WriteLine ("Your age is : " + age);
                Console.WriteLine ("Your marks are :" + marks);
            }
        }

        static void Main(string[] args)
        {

            Print obj = new Print ();
            obj.display ("George");
            obj.display (34, 76.50f);
            Console.ReadLine ();
        }
    }
}
```

Note: In the code if you observe display method is called two times. Display method will work according to the number of parameters and type of parameters.

When and why to use method overloading

Use method overloading in situation where you want a class to be able to do something, but there is more than one possibility for what information is supplied to the method that carries out the task.

You should consider overloading a method when you for some reason need a couple of methods that take different parameters, but conceptually do the same thing.

Method overloading showing many forms.

```
using System;

namespace method_overloading_polymorphism
{
    Class Program
    {
        Public class Shape
        {
            Public void Area (float r)
            {
                float a = (float)3.14 * r;
                // here we have used function overload with 1 parameter.
                Console.WriteLine ("Area of a circle: {0}",a);
```

```
        }

        Public void Area(float l, float b)
        {
           float x = (float)l* b;
           // here we have used function overload with 2 parameters.
           Console.WriteLine ("Area of a rectangle: {0}",x);

        }

        public void Area(float a, float b, float c)
        {
           float s = (float)(a*b*c)/2;
           // here we have used function overload with 3 parameters.
           Console.WriteLine ("Area of a circle: {0}", s);
        }
     }

     Static void Main (string[] args)
     {
        Shape ob = new Shape ();
        ob.Area(2.0f);
        ob.Area(20.0f,30.0f);
        ob.Area(2.0f,3.0f,4.0f);
        Console.ReadLine ();
     }
  }
}
```

Things to keep in mind while method overloading

If you use overload for method, there are couple of restrictions that the compiler imposes.

The rule is that overloads must be different in their signature, which means the name and the number and type of parameters.

There is no limit to how many overload of a method you can have. You simply declare them in a class, just as if they were different methods that happened to have the same name.

Method Overriding:

Whereas **Overriding** means changing the functionality of a method without changing the signature. We can override a function in base class by creating a similar function in derived class. This is done by using virtual/override keywords.

Base class method has to be marked with virtual keyword and we can override it in derived class using override keyword.

Derived class method will completely overrides base class method i.e. when we refer base class object created by casting derived class object a method in derived class will be called.

Example:

```
// Base class
public class BaseClass
{
public virtual void Method1()
{
Console.Write("Base Class Method");
}
}
// Derived class
public class DerivedClass : BaseClass
{
public override void Method1()
{
Console.Write("Derived Class Method");
}
}
// Using base and derived class
public class Sample
{
public void TestMethod()
{
// calling the overriden method
DerivedClass objDC = new DerivedClass();
objDC.Method1();
 // calling the baesd class method
BaseClass objBC = (BaseClass)objDC;
objDC.Method1();
}
}
```

Output

Derived Class Method

Derived Class Method

Constructors and Destructors:

Classes have complicated internal structures, including data and functions, object initialization and cleanup for classes is much more complicated than it is for simple data structures. Constructors and destructors are special member functions of classes that are used to construct and destroy class objects. Construction may involve memory allocation and initialization for objects. Destruction may involve cleanup and deallocation of memory for objects.

- Constructors and destructors do not have return types nor can they return values.
- References and pointers cannot be used on constructors and destructors because their addresses cannot be taken.
- Constructors cannot be declared with the keyword virtual.
- Constructors and destructors cannot be declared const, or volatile.
- Unions cannot contain class objects that have constructors or destructors.

Constructors and destructors obey the same access rules as member functions. For example, if you declare a constructor with protected access, only derived classes and friends can use it to create class objects.
The compiler automatically calls constructors when defining class objects and calls destructors when class objects go out of scope. A constructor does not allocate memory for the class object it's this pointer refers to, but may allocate storage for more objects than its class object refers to. If memory allocation is required for objects, constructors can explicitly call the new operator. During cleanup, a destructor may release objects allocated by the corresponding constructor. To release objects, use the delete operator.

Example of Constructor

```
class C
{
      private int x;
      private int y;
      public C (int i, int j)
      {
              x = i;
              y = j;
      }
      public void display ()
      {
            Console.WriteLine(x + "i+" + y);
      }
}
```

Example of Destructor

```
class D
{
       public D ()
       {
          // constructor
       }
       ~D ()
       {
         // Destructor
       }
}
```