OOPS in C#

- OOP stands for Object-Oriented Programming.
- Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods.
- Object-oriented programming is a way of developing software applications using real-world terminologies to create entities that interact with one another using objects.
- Object-oriented programming makes applications flexible (easy to change or add new features), reusable, well-structured, and easy to debug and test.

Main Pillers of Object-Oriented Programming :

Abstraction: Show only the necessary things.

Polymorphism: More than one form: An object can behave differently at different levels.

Inheritance: Parent and Child Class relationships.

Encapsulation: Hides the Complexity.

Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the C# code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

Class and Object

- Everything in C# is associated with **classes and objects**, along with its attributes and methods. For example: in real life, a car is an object. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.
- A Class is like an object constructor, or a "**blueprint**" for creating objects.
- ***Class and Object*** are the basic concepts of Object-Oriented Programming which revolve around the real-life entities.
- A class is a user-defined **blueprint** or prototype from which objects are created.
- Basically, a class combines the fields and methods(member function which defines actions) into a single unit.

Class:

- A class is a user-defined data type. It consists of data members and member functions, which can be accessed and used by creating an instance of that class. It represents the set of properties or methods that are common to all objects of one type. A class is like a blueprint for an object.

  For Example: Consider the Class of Cars. There may be many cars with different names and brands but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range, etc. So here, Car is the class, and wheels, speed limits, mileage are their properties.

  Declaring classes

  Classes are declared by using the **class** keyword followed by a unique identifier, as shown in the following example:

```
//[access modifier] - [class] - [identifier]

public class Customer

{

  // Fields, properties, methods and events go here...

}
```

Object:

- It is a basic unit of Object-Oriented Programming and represents the real-life entities. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated. An object has an identity, state, and behavior. Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other’s data or code, it is sufficient to know the type of message accepted and type of response returned by the objects.

  For example: “Dog” is a real-life Object, which has some characteristics like color, Breed, Bark, Sleep, and Eats.

  Creating objects

  - Although they're sometimes used interchangeably, a class and an object are different things. A class defines a type of object, but it isn't an object itself. An object is a concrete entity based on a class, and is sometimes referred to as an instance of a class.

    Objects can be created by using the new keyword followed by the name of the class, like this:

```
Customer object1 = new Customer();
```

OOPs Features

Here are the key features of OOP:

- Object Oriented Programming (OOP) is a programming model where programs are organized around objects and data rather than action and logic.
- OOP allows decomposing a problem into many entities called objects and then building data and functions around these objects.
- A class is the core of any modern object-oriented programming language such as C#.
- In OOP languages, creating a class for representing data is mandatory.
- A class is a blueprint of an object that contains variables for storing data and functions to perform operations on the data.
- A class will not occupy any memory space; hence, it is only a logical representation of data.
- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the C# code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

Note: The "Don't Repeat Yourself" (DRY) principle is about reducing the repetition of code. You should extract out the codes that are common for the application, and place them at a single place and reuse them instead of repeating it.

Inheritance

Introduction:

- Inheritance is a fundamental concept in object-oriented programming that allows us to define a new class based on an existing class.
- The new class inherits the properties and methods of the existing class and can also add new properties and methods of its own.
- Inheritance promotes code reuse, simplifies code maintenance, and improves code organization.

In C#, it is possible to inherit fields and methods from one class to another. We group the "inheritance concept" into two categories:

- **Derived Class** (child) - the class that inherits from another class
- **Base Class** (parent) - the class being inherited from

To inherit from a class, use the : symbol.

Advantages of Inheritance:

i. **Code Reusability:** Inheritance allows us to reuse existing code by inheriting properties and methods from an existing class.

ii. **Code Maintenance:** Inheritance makes code maintenance easier by allowing us to modify the base class and have the changes automatically reflected in the derived classes.
iii. **Code Organization:** Inheritance improves code organization by grouping related classes together in a hierarchical structure.

How to use inheritance

The symbol used for inheritance is **:**. **Syntax:**

```
public class ParentClass

{ // Parent class implementation

}

public class ChildClass : ParentClass

{ //child class implementation

}
```

Here are some rules for inheritance in C#:

- A class can only inherit from a single base class. C# does not support multiple inheritance, where a class can inherit from multiple base classes.
- A class can inherit from an abstract class, which is a class that contains abstract methods that must be implemented by derived classes.
- A class can also inherit from an interface, which is a set of related methods that a class can implement. A class can inherit from multiple interfaces.
- A derived class can override methods of the base class by using the override keyword. This allows the derived class to provide its own implementation of the method, while still inheriting the other methods and properties of the base class.
- A derived class can also hide methods of the base class by using the new keyword. This allows the derived class to provide its own implementation of the method, but the base class's method is still accessible using the base keyword.
- A class can use the sealed keyword to prevent other classes from inheriting from it. This is useful when you want to prevent further modification of the class.

Inheritance concepts

Method Hiding in C#

- The method of hiding the base class's methods from the derived class is known as **Method Hiding**.
- This method is also known as **Method Shadowing**.
- The implementation of the methods of a base class can be hidden from the derived class in method hiding using the **new** keyword.

- Or in other words, the base class method can be **redefined** in the derived class by using the new keyword.

Example

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;


namespace MethodHidingInInheritance

{

   public class A

   {

      public void MethodA()

      {

         Console.WriteLine("Base class");

      }

   }


   public class B : A

   {

      public new void MethodA()

      {

         //base.MethodA();

         Console.WriteLine("Derived class");

      }

   }
}
```

How to call a hidden method?

In method hiding, you can also call the hidden method of the base class in the derived class using **three** different ways and the ways are:

- ***By using the base keyword*** **you can call the hidden method of the base class in your derived class shown in the below example:**

```
public class My_Family {

  public void member()

  {

    Console.WriteLine("Total number of family members: 3");

  }

}



// Derived Class

public class My_Member : My_Family {




  public new void member()

  {



    // Calling the hidden method of the

    // base class in a derived class

    // Using base keyword

    base.member();

    Console.WriteLine("Name: Rakesh, Age: 40 \nName: Somya,"+
                    " Age: 39 \nName: Rohan, Age: 20");
        }
    }
```

- ***By casting the derived class type to base class type*** **you can invoke the hidden method. As shown in the below example. We know that in inheritance the derived class has all the capabilities of the base class so we can easily cast the object of a derived class into base class type**

```
// Base Class
public class My_Family {

    public void member()
    {
        Console.WriteLine("Total number of family members: 2");
    }
}

// Derived Class
public class My_Member : My_Family {

    public new void member() {

        Console.WriteLine("Name: Rakesh, Age: 40 "+
                "\nName: Somya, Age: 39");
    }
}
```

```
class ITV{

    // Main method
    public static void Main()
    {

        // Creating the object of the derived class
        My_Member obj = new My_Member();

        // Invoking the hidden method
        // By type casting
        ((My_Family)obj).member();
    }
}
```

- **BY Creating the Parent class reference and child class Object.**

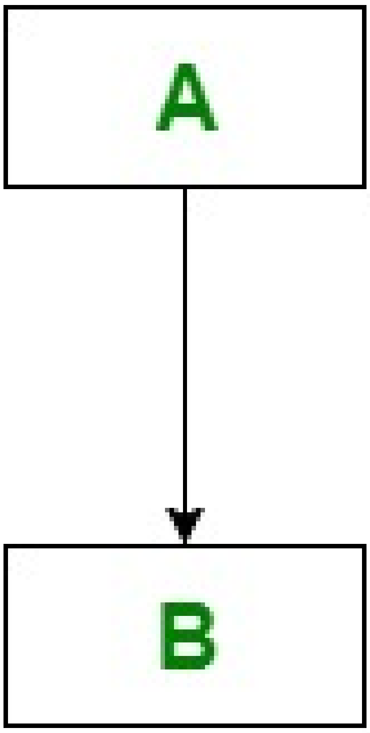
```
// Base Class
public class My_Family {

    public void member()
    {
        Console.WriteLine("Total number of family members: 2");
    }
}
```

```
// Derived Class
public class My_Member : My_Family {

    public new void member() {

        Console.WriteLine("Name: Rakesh, Age: 40 "+
                         "\nName: Somya, Age: 39");
    }
}

class ITV{

    // Main method
     public  static void Main()
    {

        // Invoking the hidden method
        My_Family obj = new My_Member();
        obj.member();
    }
}
```

Types of Inheritance in C#

Below are the different types of inheritance which is supported by C# in different combinations.

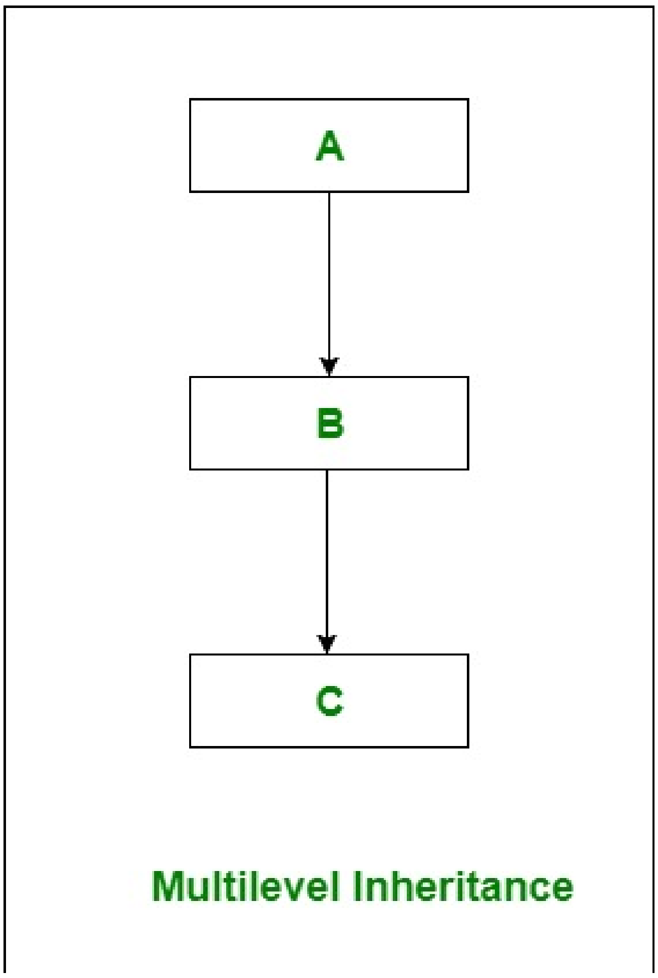
i. **Single Inheritance:** In single inheritance, subclasses inherit the features of one superclass. In image below, the class A serves as a base class for the derived class B.



Example :

```
public class A

 {

    public void MethodA()

    {

       Console.WriteLine("Base class");

    }

 }


 public class B : A

 {

    public void MethodB()

    {
            Console.WriteLine("Derived class");
          }
        }
      }
```

i. **Multilevel Inheritance:** In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.



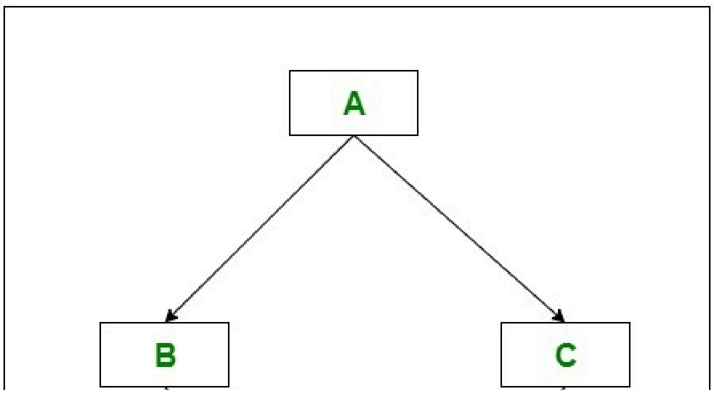
Example :

```
      public class A
```

```
{
    public void MethodA()
    {
        Console.WriteLine("Class A");
    }
}

public class B : A
{
    public void MethodB()
    {
        Console.WriteLine("Class B");
    }
}

 public class C : B
{
    public void MethodC()
    {
        Console.WriteLine("Class C");
    }
}
public class D : C
{
```

```
            public void MethodD()

            {

               Console.WriteLine("Class D");

            }

         }

      }
```

i. **Hierarchical Inheritance:** In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In below image, class A serves as a base class for the derived class B, C, and D.

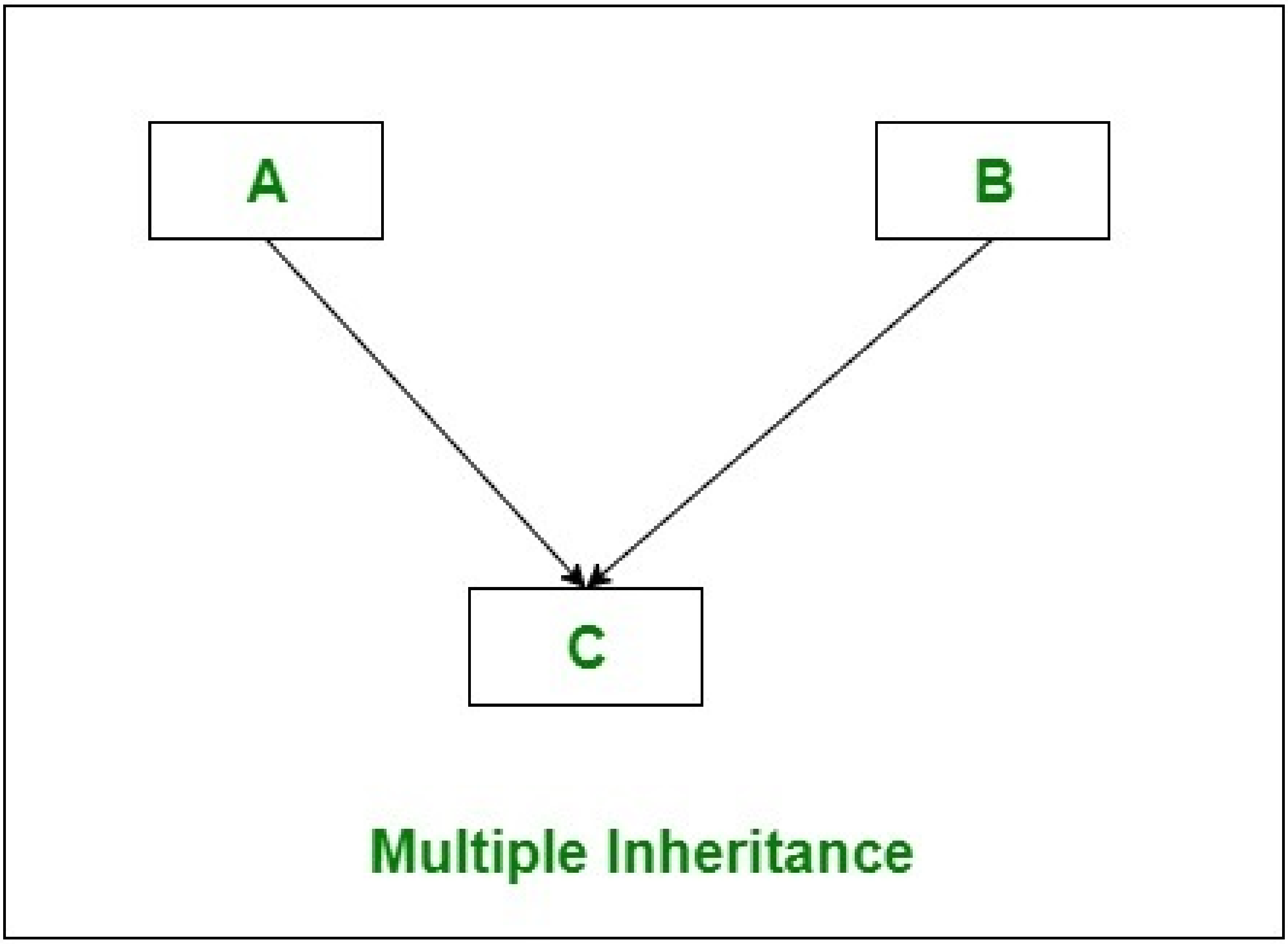


Example :

```
      public class A

         {

            public void MethodA()

            {

               Console.WriteLine("Class A");

            }

         }


         public class B : A
```

```
{
   public void MethodB()
   {
      Console.WriteLine("Class B");
   }
}


 public class C : A
{
   public void MethodC()
   {
      Console.WriteLine("Class C");
   }
}
```

i. **Multiple Inheritance(Through Interfaces):**In Multiple inheritance, one class can have more than one superclass and inherit features from all parent classes. Please note that **C# does not support multiple inheritance** with classes. In C#, we can achieve multiple inheritance only through Interfaces. In the image below, Class C is derived from interface A and B.



Example :

```
interface IA
  {
     void MethodA();
  }
  interface IB
  {
     void MethodA();
  }
```

```
public class C : IA, IB //implement

{

    public void MethodA()

    {

        Console.WriteLine("Implemented multiple inheritance using interface");

    }

}
```

Polymorphism

- Polymorphism is a Greek word meaning "**one name many forms**.
- "Poly" means many, and "morph" means forms. In other words, one object has many forms or has one name with multiple functionalities.
- Polymorphism allows a class to have **multiple implementations with the same name.**
- It is one of the core principles of Object Oriented Programming after encapsulation and inheritance.

Types of Polymorphism

There are two types of polymorphism in C#:

- **Static / Compile Time Polymorphism**
- **Dynamic / Runtime Polymorphism**

Method Overriding(Run Time Polymorphism)

- During inheritance in C#, if the same method is present in both the **superclass** and the **subclass**. Then, the method in the subclass overrides the same method in the superclass. This is called **method overriding**.
- In this case, the same method will perform one operation in the superclass and another operation in the subclass.
- We can use **virtual** and **override** keywords to achieve method overriding.

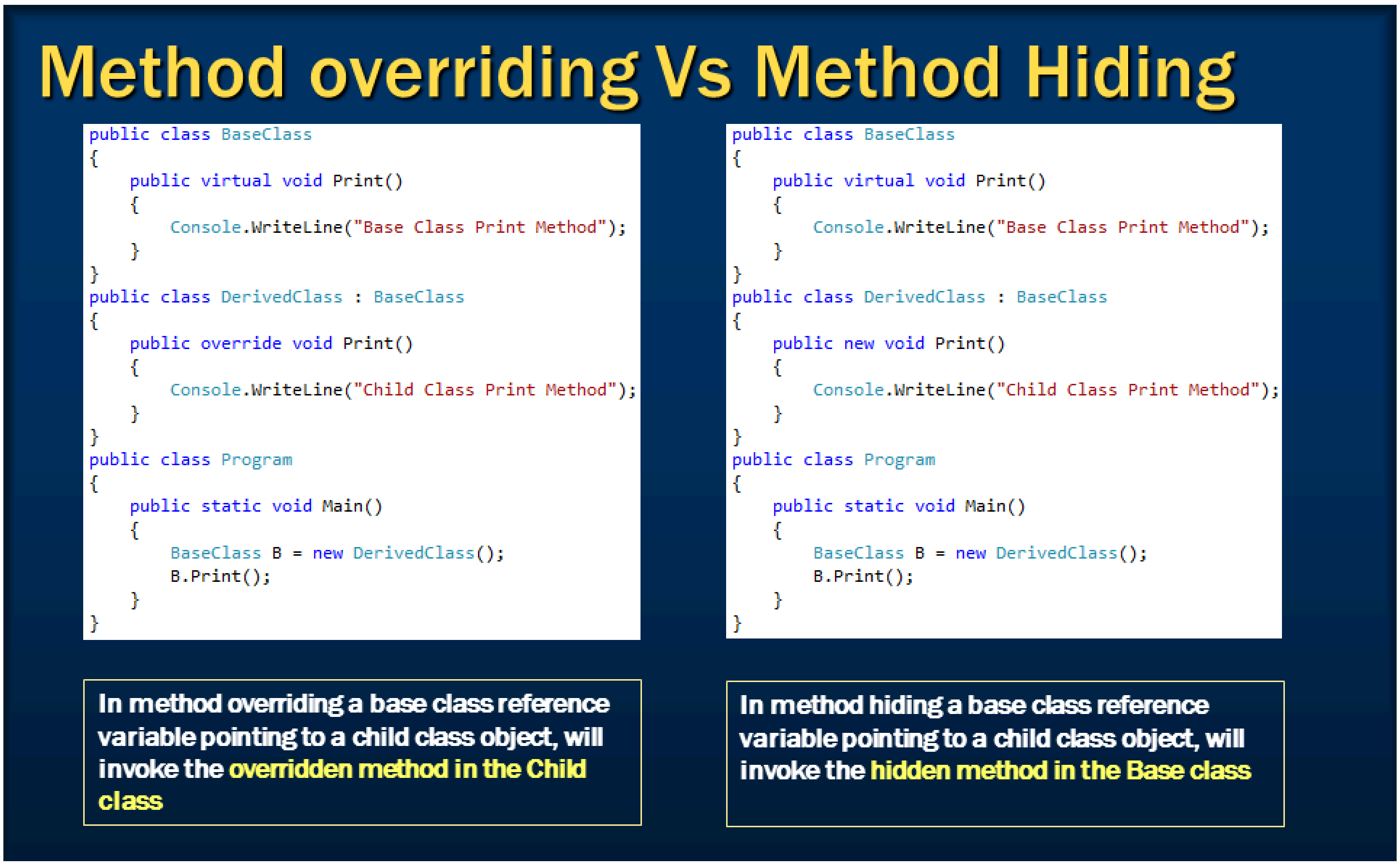
- Polymorphism allows you to invoke derived class methods through a base class reference during **runtime**.
- This is called as **late binding**.

Example

```
public class BaseClass
  {
    public virtual void MethodA()
    {
      Console.WriteLine("Base class");
    }
  }


  public class DerivedClass : BaseClass
  {
    public override void MethodA()
    {
      Console.WriteLine("Derived class");
    }
}
```

Difference between Method overriding Vs Method Hiding

Method overriding	Method hiding
In method overriding, you need to define the method of a parent class as a virtual method using virtual keyword and the method of child class as an overridden method using override keyword.	In method hiding, you just simply create a method in a parent class and in child class you need to define that method using new keyword.
It only redefines the implementation of the method.	In method hiding, you can completely redefine the method.
Here overriding is an object type.	Here hiding is a reference type.
If you do not use override keyword, then the compiler will not override the method. Instead of the overriding compiler will hide the method.	If you do not use the new keyword, then the compiler will automatically hide the method of the base class.
In method overriding, when base class reference variable pointing to the object of the derived class, then it will call the overridden method in the derived class.	In the method hiding, when base class reference variable pointing to the object of the derived class, then it will call the hidden method in the base class.



Method overriding Vs Method Hiding

```
public class BaseClass
{
    public virtual void Print()
    {
        Console.WriteLine("Base Class Print Method");
    }
}
public class DerivedClass : BaseClass
{
    public override void Print()
    {
        Console.WriteLine("Child Class Print Method");
    }
}
public class Program
{
    public static void Main()
    {
        BaseClass B = new DerivedClass();
        B.Print();
    }
}
```

In method overriding a base class reference variable pointing to a child class object, will invoke the overridden method in the Child class

```
public class BaseClass
{
    public virtual void Print()
    {
        Console.WriteLine("Base Class Print Method");
    }
}
public class DerivedClass : BaseClass
{
    public new void Print()
    {
        Console.WriteLine("Child Class Print Method");
    }
}
public class Program
{
    public static void Main()
    {
        BaseClass B = new DerivedClass();
        B.Print();
    }
}
```

In method hiding a base class reference variable pointing to a child class object, will invoke the hidden method in the Base class

Method Overloading (Compile time polymorphism)

- ***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.
- Overloaded methods are differentiated based on the number and type of the **parameters passed as arguments to the methods.**
- You can not define more than one method with the same name, Order and the type of the arguments. It would be compiler error.
- 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.
- **By Changing the return type method overloading is not possible.**
- **This is called as early binding.**

Different ways of doing overloading methods-

Method overloading can be done by changing:

1. **The number of parameters in two methods.**

   Example

```
//Number of params are different

 public class CalculatorNoOfParams

 {

   public static void Addition(int number1, int number2)

   {

     Console.WriteLine("Sum : {0}",number1 + number2);

   }

   public static void Addition(int number1, int number2, int number3)

   {

     Console.WriteLine("Sum : {0}", number1 + number2 + number3);

   }
```

```
        public static void Addition(int number1, int number2, int number3, int number4)

        {

            Console.WriteLine("Sum : {0}", number1 + number2 + number3 + number4);

        }
```

1. **The data types of the parameters of methods.**

Example

```
//Type of params are different

   public class CalculatorTypeOfParams

   {

      public static void Addition(int number1, int number2)

      {

         Console.WriteLine("Sum : {0}", number1 + number2);

      }

      public static void Addition(float number1, float number2)

      {

         Console.WriteLine("Sum : {0}", number1 + number2);

      }

      public static void Addition(int number1, float number2)

      {

         Console.WriteLine("Sum : {0}", number1 + number2);

      }

   }
```

a. **The kind of the parameters of methods.**

Example

```
//Kind of params are different
 public class CalculatorKindOfParams
 {
   public static void Addition(int number1, int number2)
   {
     Console.WriteLine("Sum : {0}", number1 + number2);
   }
   public static void Addition(int number1, int number2, out int number3)
   {
     Console.WriteLine("Sum : {0}", number1 + number2);
     number3 = number1 + number2;
     Console.WriteLine(number3);
   }
 }
```