Object Oriented Programing

Lecture 7 polymorphism

Arsalan Rahman Mirza

Computer Science Department

Faculty of Science

2022-2023

Inheritance

```
class B
{

}

class D : B
{

}
```

What is Polymorphism?

Greek Term : Many Forms

"Polymorphism is an object-oriented programming concept that refers to the ability of a variable, function or object to take on multiple forms. A language that features polymorphism allows developers to program in the general rather than program in the specific"

Object Initialization

```
B x = new B();
```

X is type of B
its Value is type of B

```
D y = new D();
```

Y is type of D
its Value is type of D

```
B z = new D();
```

Z is type of B
its Value is type of D
Only allowed if D is Derived from B

Notes

- Object pointer could be type of Base class and their value could be type of Derived Class.
- Base and Derived class both can have their objects.
- The keyword new indicate the value of the object.

Polymorphism

- In c#, **Polymorphism** means providing an ability to take more than one form and it's a one of the main pillar concept of object oriented programming, after encapsulation and inheritance.
-
- Generally, the polymorphism is a combination of two words, one is **poly** and another one is **morphs**. Here **poly** means "**multiple**" and **morphs** means "**forms**" so polymorphism means many forms.
- In c#, polymorphism provides an ability for the classes to implement a different methods that are called through the same name and it also provides an ability to invoke the methods of derived class through base class reference during runtime based on our requirements.

Polymorphism

- In c#, we have a two different kind of polymorphisms available, those are

- Compile Time Polymorphism
- Run Time Polymorphism

Overloading

```
public class Calculate
{
    public void AddNumbers(int a, int b)
    {
        Console.WriteLine("a + b = {0}", a + b);
    }
    public void AddNumbers(int a, int b, int c)
    {
        Console.WriteLine("a + b + c = {0}", a + b + c);
    }
    }
```

Overloading

- In c#, the compile time polymorphism can be achieved by using **method overloading** and it is also called as **early binding** or **static binding**.
- Same class
- Same name
- Different parameter

Overriding

```
using System;
 namespace Tutlane
{
    // Base Class
    public class BClass
    {
        public virtual void GetInfo()
        {
            Console.WriteLine("Learn C# Tutorial");
        }
    }
    // Derived Class
    public class DClass : BClass
    {
        public override void GetInfo()
        {
            Console.WriteLine("Welcome to Tutlane");
        }
    }
```

```
class Program
  {
      static void Main(string[] args)
      {
          DClass d = new DClass();
          d.GetInfo();
          BClass b = new BClass();
          b.GetInfo();
          Console.WriteLine("Press Enter Key to
Exit..");
          Console.ReadLine();
      }
  }
}
```

Overriding

- If you observe above code snippet, we created a two classes **Bclass base** and the derived class (**DClass**) is inheriting the properties from base class (**BClass**) and we are overriding the base class method **GetInfo** in derived class by creating a same function to achieve **method overriding**, this is called a **run time polymorphism** in c#.
- Here, we defined a **GetInfo** method with **virtual** keyword in base class to allow derived class to override that method using **override** keyword.
- Generally, only the methods with **virtual** keyword in base class are allowed to override in derived class using **override** keyword.
- Different class
- Same name
- Same parameter

Two Methods 1 and 2

```
class B
{
    public void Method1()
    { Console.WriteLine("Base Method 1");     }
}
class D:B
{
    public void Method2()
    { Console.WriteLine("Derived Method 2");  }
}
```

Find Errors and Callings

```
B x = new B();
D y = new D();
B z = new D();
```

Class B
{ Public Method1 ()}

Class D: B
{ Public Method2 ()}

`x.Method1();`	Calls Method 1 of Base
`x.Method2();`	Error, no Method 2 in Base
`y.Method1();`	Calls Method 1 of Base
`y.Method2();`	Calls Method 2 of Derived
`z.Method1();`	Calls Method 1 of Base
`z.Method2();`	Error, no Method 2 in Base

Notes

- Method 1 of Base has been derived into Derived Class that is why Y has access to it.
- Z, is type of Base so there is no Method 2 in this object even if its value is from Derived Class.

Two Methods 1 and 2

```
class B
{
    public void Method1()
    {   Console.WriteLine("Base Method 1");      }

    public void Method2()
    {   Console.WriteLine("Base Method 2");      }
}
class D:B
{
    public void Method2()
    {  Console.WriteLine("Derived Method 2");    }
}
```

Find Callings

```
B x = new B();
D y = new D();
B z = new D();
```

Class B
{ Public Method1 ()
  Public Method2 ()}

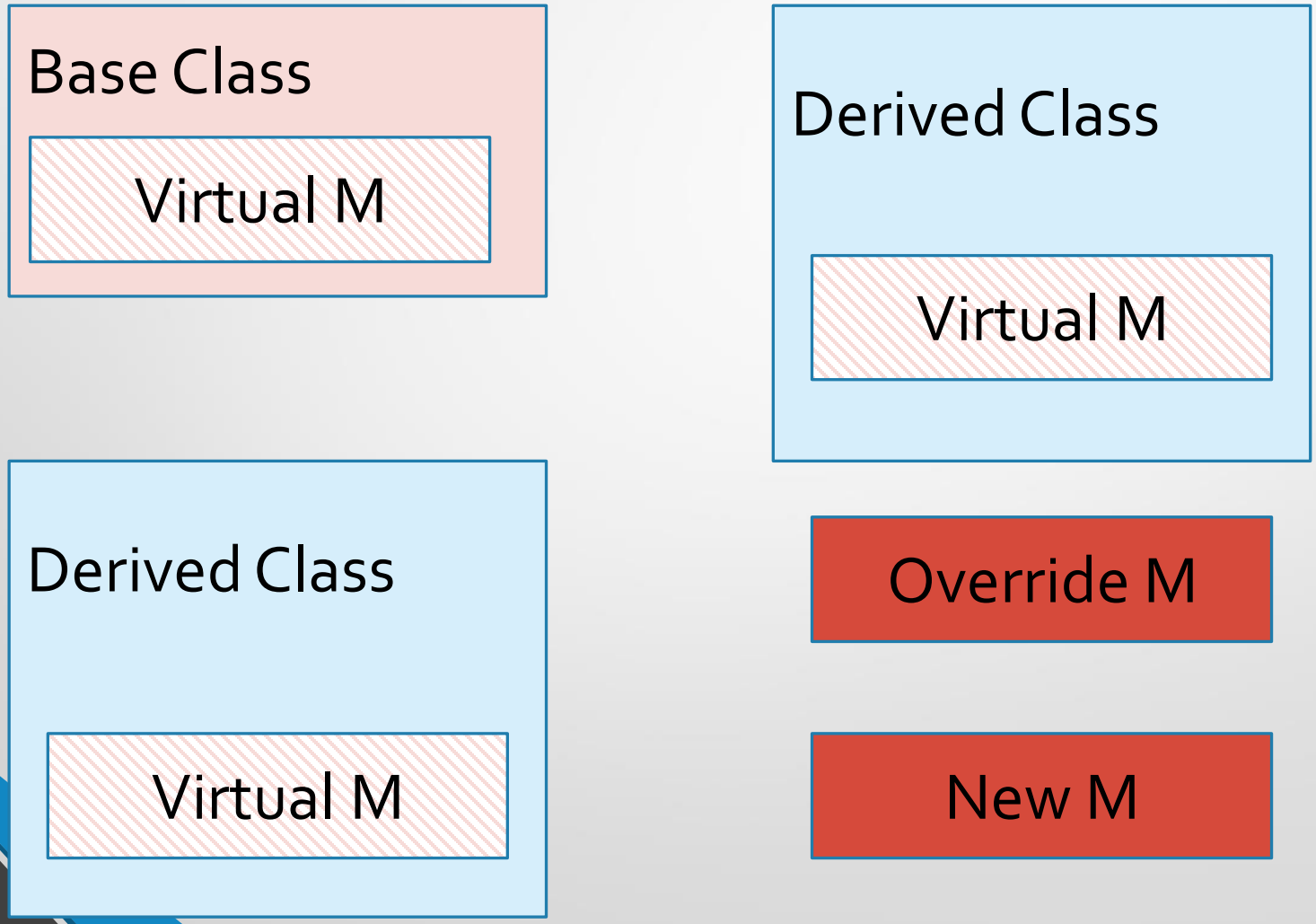
Class D: B
{ Public Method2 ()}

`x.Method1();`	Calls Method 1 of Base
`x.Method2();`	Calls Method 2 of Base
`y.Method1();`	Calls Method 1 of Base
`y.Method2();`	Calls Method 2 of Derived
`z.Method1();`	Calls Method 1 of Base
`z.Method2();`	Calls Method 2 of Base

Notes

- It is normal to have same methods in both Derived Class and Base Class.
- The object with type and value of Derived will call the Derived method.
- The object with type of Base and Value of Derived will call the Base Method.

Virtual, Override and New



Two Methods 1 and 2

```
class B
{
    public void Method1()
    {       Console.WriteLine("Base Method 1");    }

    public void Method2()
    {       Console.WriteLine("Base Method 2");    }
}
class D:B
{
    public void Method1()
    {       Console.WriteLine("Derived Method 1");    }

    public void Method2()
    {       Console.WriteLine("Derived Method 2");    }
}
```

Find Callings

```
B x = new B();
D y = new D();
B z = new D();
```

Class B
{ Public Virtual Method1 ()
Public Virtual Method2 ()}

Class D: B
{Public Override Method1 ()
Public New Method2 ()}

`x.Method1();`	Calls Method 1 of Base
`x.Method2();`	Calls Method 2 of Base
`y.Method1();`	Calls Method 1 of Derived
`y.Method2();`	Calls Method 2 of Derived
`z.Method1();`	Calls Method 1 of Derived
`z.Method2();`	Calls Method 2 of Base

Notes

- **Virtual** in Base class hides the method if there is an **Override**.
- The keyword **new**, has no effect on Virtual methods, it only shows that if this method does not exists in the base, it is new in the derived class.