OOP (Object Oriented Programming) with C#: Dynamic Binding/ Run Time Polymorphism. –by Akhil Mittal

Table of Contents:

Introduction:

In our previous parts of the learning OOP series, we were talking more of compile time polymorphism, params keyword, Inheritance, base keyword etc. This part of the article series will focus more on run time polymorphism also called late binding. We'll use same technique of learning, less theory and more hands-on. We'll take small code snippets to learn the concept more deeply .To master this concept is like learning more than 50% of OOP

Pre-requisites

Since this is the third part of the series, I expect my readers to be expert in Compile time polymorphism and inheritance. Although it doesn't matter if you are directly starting learning from this article, you can take the other articles later.

Runtime Polymorphism or Late Binding or Dynamic Binding

In simple C# language, In run time polymorphism or method overriding we can override a method in base class by creating similar method in derived class this can be achieved by using inheritance principle and using "virtual & override" keywords.

What are New and Override keywords in C#?

Create a console application named **InheritanceAndPolymorphism** in your visual studio.
Just add two classes and keep the Program.cs as it is.The two classes will be named ClassA.cs and ClassB.cs and add one method in each class as follows,

ClassA:

```
public class ClassA
  {
    public void AAA()
    {
      Console.WriteLine("ClassA AAA");
    }

    public void BBB()
    {
      Console.WriteLine("ClassA BBB");
    }

    public void CCC()
    {
      Console.WriteLine("ClassA CCC");
    }
  }
```

ClassB:

```
  public class ClassB
  {
    public void AAA()
    {
      Console.WriteLine("ClassB AAA");
    }

    public void BBB()
    {
      Console.WriteLine("ClassB BBB");
    }
```

```
        public void CCC()
        {
            Console.WriteLine("ClassB CCC");
        }
    }
```

<u>Program.cs:</u>

```
    /// <summary>
    /// Program: used to execute the method.
    /// Contains Main method.
    /// </summary>
    public class Program
    {
        private static void Main(string[] args)
        {

        }
    }
```

We see both classes ClassA and ClassB, have the same no. of methods with similar names in both the classes.Now let's inherit ClassA from ClassB, and create instances of the classes and call their methods in program.cs.
So our code for the two classes becomes,

```
/// <summary>
    /// ClassB, acting as a base class
    /// </summary>
    public class ClassB
    {
        public void AAA()
        {
            Console.WriteLine("ClassB AAA");
        }

        public void BBB()
        {
            Console.WriteLine("ClassB BBB");
        }

        public void CCC()
        {
            Console.WriteLine("ClassB CCC");
        }
    }

    /// <summary>
    /// Class A, acting as a derived class
    /// </summary>
    public class ClassA : ClassB
    {
        public void AAA()
        {
            Console.WriteLine("ClassA AAA");
        }

        public void BBB()
        {
            Console.WriteLine("ClassA BBB");
        }

        public void CCC()
        {
            Console.WriteLine("ClassA CCC");
        }
    }
```

<u>Program.cs</u>

```
    /// <summary>
    /// Program: used to execute the method.
    /// Contains Main method.
    /// </summary>
    public class Program
    {
        private static void Main(string[] args)
        {
            ClassA x = new ClassA();
            ClassB y=new ClassB();
            ClassB z=new ClassA();

            x.AAA(); x.BBB(); x.CCC();
            y.AAA(); y.BBB();y.CCC();
            z.AAA(); z.BBB(); z.CCC();
        }
    }
```

Now press F5, i.e. run the code, what do we get?

Output:

```
ClassB AAA
ClassB BBB
ClassB CCC
ClassA AAA
ClassA BBB
ClassA CCC
ClassB AAA
ClassB BBB
ClassB CCC
```

But with the output we also got three warnings,

Warnings:

```
'InheritanceAndPolymorphism.ClassA.AAA()' hides inherited member
'InheritanceAndPolymorphism.ClassB.AAA()'. Use the new keyword if hiding was intended.

'InheritanceAndPolymorphism.ClassA.BBB()' hides inherited member
'InheritanceAndPolymorphism.ClassB.BBB()'. Use the new keyword if hiding was intended.

'InheritanceAndPolymorphism.ClassA.CCC()' hides inherited member
'InheritanceAndPolymorphism.ClassB.CCC()'. Use the new keyword if hiding was intended.
```

Point to remember: In C#, a smaller object could be equated to a bigger object.

Class ClassB is the super class of class ClassA. That means ClassA the derived class and ClassB the base class. The class ClassA comprises ClassB and something more. So we can conclude that object of ClassA is bigger than object of ClassB.Since ClassA is inherited from ClassB, it contains its own methods and properties moreover it will also contain methods/properties that are inherited from ClassB too.
Let's take the case of object y. It looks like ClassB and initialized by creating an object that also looks like ClassB, well and good. Now, when we call the methods AAA and BBB and CCC through the object y we know that it will call them from ClassB.

Object x looks like that of ClassA, i.e. the derived class. It is initialized to an object that looks like ClassA. When we call AAA, BBB and CCC method through x, it calls AAA, BBB and CCC from ClassA.
Now there is somewhat tricky situation we are dealing with,
Object z again looks like ClassB, but it is now initialized to an object that looks like ClassA which does not give an error as explained earlier. But there is no change at all in the output we get and the behavior is identical to that of object y.Therefore initializing it to an object that looks like ClassB or ClassA does not seem to matter.

Experiment:

Let's experiment with the code ,and put override behind AAA and new behind BBB methods of ClassA i.e. the derived class.
Our code:

ClassB:

```
    /// <summary>
    /// ClassB, acting as a base class
    /// </summary>
    public class ClassB
    {
        public void AAA()
        {
            Console.WriteLine("ClassB AAA");
        }

        public void BBB()
        {
            Console.WriteLine("ClassB BBB");
        }

        public void CCC()
        {
            Console.WriteLine("ClassB CCC");
        }
    }
```

ClassA:

```
    /// <summary>
    /// Class A, acting as a derived class
    /// </summary>
    public class ClassA : ClassB
    {
        public override void AAA()
        {
            Console.WriteLine("ClassA AAA");
        }

        public new void BBB()
        {
            Console.WriteLine("ClassA BBB");
        }

        public void CCC()
        {
            Console.WriteLine("ClassA CCC");
        }
    }
```

Program.cs:

```
    /// <summary>
    /// Program: used to execute the method.
    /// Contains Main method.
    /// </summary>
    public class Program
    {
        private static void Main(string[] args)
        {
            ClassB y = new ClassB();
            ClassA x = new ClassA();
            ClassB z = new ClassA();

            y.AAA(); y.BBB(); y.CCC();
            x.AAA(); x.BBB(); x.CCC();
            z.AAA(); z.BBB(); z.CCC();

            Console.ReadKey();
        }
    }
```

We get **Output:**

Error: 'InheritanceAndPolymorphism.ClassA.AAA()': cannot override inherited member 'InheritanceAndPolymorphism.ClassB.AAA()' because it is not marked virtual, abstract, or override

*** InheritanceAndPolymorphism:** It's the namespace I used for my console application, so you can ignore that.

We got error after we added these two modifiers in the derived class methods, the error tells us to mark the methods virtual, abstract or override in the base class.

OK, how does it matter to me?

I marked all the methods of base class as virtual,
Now our code and output looks like,

```
/// <summary>
    /// ClassB, acting as a base class
    /// </summary>
```

```
public class ClassB
{
    public virtual void AAA()
    {
        Console.WriteLine("ClassB AAA");
    }

    public virtual void BBB()
    {
        Console.WriteLine("ClassB BBB");
    }

    public virtual void CCC()
    {
        Console.WriteLine("ClassB CCC");
    }
}

/// <summary>
/// Class A, acting as a derived class
/// </summary>
public class ClassA : ClassB
{
    public override void AAA()
    {
        Console.WriteLine("ClassA AAA");
    }

    public new void BBB()
    {
        Console.WriteLine("ClassA BBB");
    }

    public void CCC()
    {
        Console.WriteLine("ClassA CCC");
    }
}

/// <summary>
/// Program: used to execute the method.
/// Contains Main method.
/// </summary>
public class Program
{
    private static void Main(string[] args)
    {
        ClassB y = new ClassB();
        ClassA x = new ClassA();
        ClassB z = new ClassA();

        y.AAA(); y.BBB(); y.CCC();
        x.AAA(); x.BBB(); x.CCC();
        z.AAA(); z.BBB(); z.CCC();

        Console.ReadKey();
    }
}
```

Output:

```
ClassB AAA
ClassB BBB
ClassB CCC
ClassA AAA
ClassA BBB
ClassA CCC
ClassA AAA
ClassB BBB
ClassB CCC
```

Point to remember: The override modifier is needed as the derived class methods will get first priority and be called upon.

We see here that there is only a single small change in the workings of the object z only and not in x and y. This strange output occurred only after we added virtual modifier in the base class methods. The difference is in the object z. z looks like the base class ClassB but is initialized to an instance that looks like that of derived class ClassA. C# knows this fact. When we run z.AAA(), C# remembers that instance z was initialized by a ClassA object and hence it first goes to class ClassA, too obvious. Here the method has a modifier override which literally means, forget about the data type of z which is ClassB, call AAA from ClassA as it overrides the AAA of the base class. The override modifier is needed as the derived class methods will get first priority and be called upon.
We wanted to override the AAA of the base class ClassB. We are infact telling C# that this AAA is similar to the AAA of the one in base class.

New keyword acts exact opposite to override keyword. The method BBB as we see has the new modifier. z.BBB() calls BBB from ClassB and not ClassA. New means that the method BBB is a new method and it has absolutely nothing to do with the BBB in the base class. It may have the same name i.e. BBB as in the base class, but that is only a coincidence. As z looks like ClassB, the BBB of ClassB gets called even though there is a BBB in ClassA. When we do not write any modifier, then it is assumed that we wrote new. So every time we write a method, C# assumes it has nothing to do with the base class.

Point to remember: These modifiers like new and override can only be used if the method in the base class is a virtual method. Virtual means that the base class is granting us permission to invoke the method from the derived class and not the base class. But, we have to add the modifier override if our derived class method has to be called.

Run time polymorphism with three classes:

Let's get into some more action. Let's involve one more class in the play. Let's add a class named ClassC, and design our three classes and program.cs as follows,

```
/// <summary>
/// ClassB, acting as a base class
/// </summary>
public class ClassB
{
   public  void AAA()
   {
      Console.WriteLine("ClassB AAA");
   }

   public virtual void BBB()
   {
      Console.WriteLine("ClassB BBB");
   }

   public virtual void CCC()
   {
      Console.WriteLine("ClassB CCC");
   }
}

/// <summary>
/// Class A, acting as a derived class
/// </summary>
public class ClassA : ClassB
{
   public virtual void AAA()
   {
      Console.WriteLine("ClassA AAA");
   }

   public new void BBB()
   {
      Console.WriteLine("ClassA BBB");
   }
```

```
        public override void CCC()
        {
            Console.WriteLine("ClassA CCC");
        }
    }

    /// <summary>
    /// Class C, acting as a derived class
    /// </summary>
    public class ClassC : ClassA
    {
        public override void AAA()
        {
            Console.WriteLine("ClassC AAA");
        }

        public void CCC()
        {
            Console.WriteLine("ClassC CCC");
        }
    }

    /// <summary>
    /// Program: used to execute the method.
    /// Contains Main method.
    /// </summary>
    public class Program
    {
        private static void Main(string[] args)
        {
            ClassB y = new ClassA();
            ClassB x = new ClassC();
            ClassA z = new ClassC();

            y.AAA(); y.BBB(); y.CCC();
            x.AAA(); x.BBB(); x.CCC();
            z.AAA(); z.BBB(); z.CCC();

            Console.ReadKey();
        }
    }
```

Output:

```
ClassB AAA
ClassB BBB
```

```
ClassA CCC
ClassB AAA
ClassB BBB
ClassA CCC
ClassC AAA
ClassA BBB
ClassA CCC
```

Don’t be scared of the long example that we have taken.This will help you to learn the concept in detail. We have already learned that we can initialize a base object to a derived object. But vice versa will result into error. This leads to an instance of a base class being initialized to an instance of the derived class. So the question is now that which method will be called when. The method from the base class or from the derived class.

Point to remember: If the base class object declared the method virtual and the derived class used the modifier override, the derived class method will get called. Otherwise the base class method will get executed. Therefore for virtual methods, the data type created is decided at run time only.

Point to remember: All the methods not marked with virtual are non virtual, and the method to be called is decided at compile time, depending upon the static data type of the object.

If the object of a class is initialized to the same data type, none of the above rule would apply. Whenever we have a mismatch, we always need rules to resolve the mismatch. So we can land up with a scenario where an object to a base class can call a method in the derived class.

The object y that looks like of ClassB but is initialized here to the derived class i.e. ClassA.
y.AAA(), first looks into the class ClassB. Here it verifies whether the method AAA is marked virtual. The answer is an emphatic no and hence everything comes to halt and the method AAA gets called from class ClassB.
y.BBB also does the same thing, but the method now is defined virtual in class ClassB. Thus
C# looks at the class ClassB, the one it was initialized to. Here BBB is marked with the modifier “new”. That means BBB is a new method which has nothing to do with the one in the base class. They only accidentally share the same name. So as there is no method called BBB (as it is a new BBB) in the derived class, the one from base class gets called. In the scene of y.CCC(), the same above steps are followed again, but in the class ClassB, we see the modifier override, that by behaviour overrides the method in the base class. We are actually telling C# to call this method in class ClassA and not the one in the base class i.e. ClassB.

I just got this picture from the internet that depicts our current situation of classes. We are learning the concept like charm now. OOP is becoming easy now.

The object x which also looks like that of class ClassB, is now initialized with an object that looks like our newly introduced class ClassC and not ClassA like before. Since AAA is a non virtual method it gets called from ClassB. In the case of method BBB, C# now looks into class ClassC. Here it does not find a method named BBB and so ultimately propagates and now looks into class ClassA. Therefore the above rules repeat on and on and it gets called from class ClassB. In the case of x.CCC, in class ClassC, it is already marked new by default and therefore this method has nothing to do with the one declared in class ClassB. So the one from ClassC does not get called but the one from class ClassB where it is marked as override.

Now if we modify a bit our CCC method in ClassC and change it to the code as shown below,

```
public override void CCC()
{
   Console.WriteLine("ClassC CCC");
}
```

We changed default new to override, the CCC of ClassC will now be called.

The last object z looks like of ClassA but is now initialized to an object that looks like the derived class ClassC, we know we can do this. So z.AAA() when called, looks first into class ClassA where it is flagged as virtual. Do you recall that AAA is non virtual in class ClassB but marked as a virtual in ClassA. From
now on, the method AAA is virtual in ClassC also but not in class ClassB. Virtual always flows from upwards to downwards like a waterfall. Since AAA() is marked virtual, we now look into class ClassC. Here it is marked override and therefore AAA() gets called from class ClassC. In the case of BBB(),BBB() is marked virtual in class ClassB and new in ClassA, but as there is no method BBB in ClassC, none of the modifier matters at all in this case. Finally it gets invoked from class ClassA. At last for method CCC, in class ClassC it is
marked as new. Hence it has no relation with the CCC in class ClassA which results in method CCC gets invoked from ClassA but not ClassB.

One more example,

```
internal class A
{
   public virtual void X()
   {
   }
}

internal class B : A
{
   public new void X()
   {
   }
}

internal class C : B
{
    public override void X()
    {
    }
}
```

In the above example code is very much self explanatory,the out put which we’ll get is,

Error: 'InheritanceAndPolymorphism.C.X()': cannot override inherited member 'InheritanceAndPolymorphism.B.X()' because it is not marked virtual, abstract, or override

Strange ! , We got an error as the method X() in class B is marked new. That means it hides the X() of class A. If we talk about class C, B does not supply a method named X. The method X defined in class B has nothing to do with the method X in defined in class A. This means that the method X of class B does not inherit the virtual modifier from the method X() of class A. This is what the compiler complained about. As the method X in B has no virtual modifier, in C we cannot use the modifier override. We can, however, use the modifier new and remove the warning ☺.

Cut off relations:

```
internal class A
{
    public virtual void X()
    {
        Console.WriteLine("Class: A ; Method X");
    }
}

internal class B : A
{
    public new virtual void X()
    {
        Console.WriteLine("Class: B ; Method X");
    }
}

internal class C : B
{
    public override void X()
    {
        Console.WriteLine("Class: C ; Method X");
    }
```

```
  }

  /// <summary>
  /// Program: used to execute the method.
  /// Contains Main method.
  /// </summary>
  public class Program
  {
    private static void Main(string[] args)
    {
      A a = new C();
      a.X();
      B b = new C();
      b.X();

      Console.ReadKey();
    }
  }
```

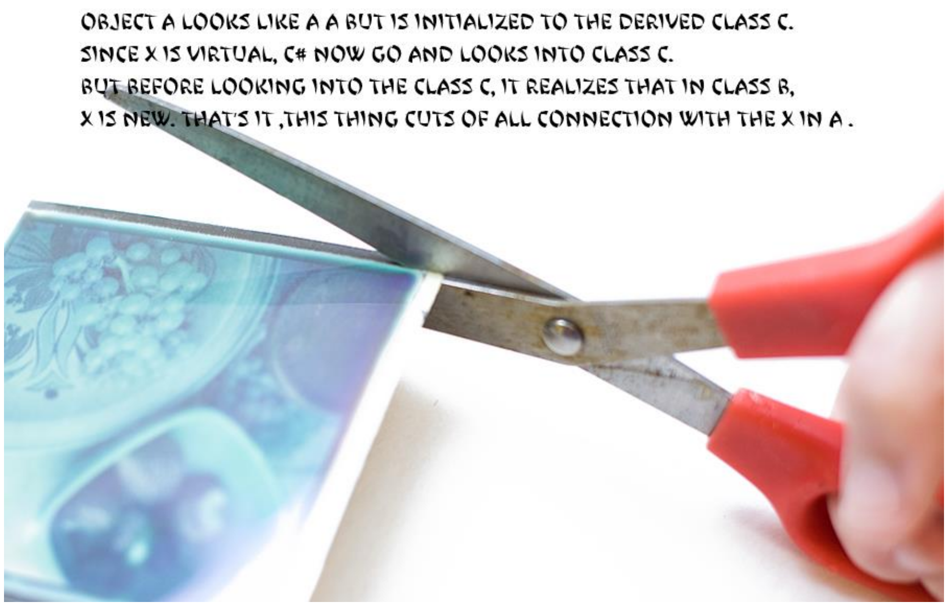
Output:

```
Class: A ; Method X
Class: C ; Method X
```

If in the above code , we remove the modifier override from X() in class C, we get,

Output:

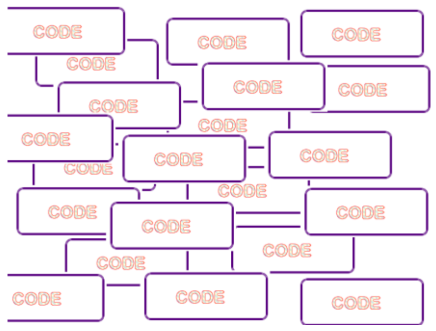
```
Class: A ; Method X
Class: B ; Method X
```


OBJECT A LOOKS LIKE A A BUT IS INITIALIZED TO THE DERIVED CLASS C.
SINCE X IS VIRTUAL, C# NOW GO AND LOOKS INTO CLASS C.
BUT BEFORE LOOKING INTO THE CLASS C, IT REALIZES THAT IN CLASS B,
X IS NEW. THAT'S IT ,THIS THING CUTS OF ALL CONNECTION WITH THE X IN A .

Yes, that's the problem with virtual methods. Sometimes they are too confusing, the result is entirely different with what we expect. Object a looks like a A but is initialized to the derived class C. Since X is virtual, C# now go and looks into class C. But before looking into the class C, it realizes that in class B, X is new. That's it ,this thing cuts of all connection with the X in A. Thus the keyword new is preceded with virtual, otherwise the override modifier would give us an error in class C. As X in class B is marked as new method, having nothing to do with the class A, class C inherits a new which also has nothing to do with the class A. The X in class C is related to the X of class B and not of class A. Thus the X of class A gets invoked.

In the second case object b looks like class B now but in turn is initialized to an object of class C. C# first looks at class B. Here X is new and virtual both, which makes it a unique method X. Sadly, the X in C has the override modifier which sees to it that the X of C hides the X of B. This calls the X of C instead. If we remove the override modifier from X in class C, the default will be new, that cuts off the relation from the X of B. Thus, as it is, a new method, the X of B gets invoked.
A virtual method cannot be marked by the modifiers static, abstract or override. A non virtual method is said to be invariant. This means that the same method gets called always, irrespective of whether one exists in the base class or derived class. In a virtual method the run-time type of the object decides on which method to be called and not the compile-time type as is in the case of non virtual methods. For a virtual method there exists a most derived implementation which gets always gets called.

Runtime Polymorphism with four classes:



OK! We did a lot of coding. How about if I tell you that we are going to add one more class to our code, yes that is class ClassD. So we go more deep into concept of Polymorphism and Inheritance.
We add one more class to the three classes solution on which we were working on (remember?). So our new class is named ClassD.

Let's take our new class into action,

```
/// <summary>
   /// Class A
   /// </summary>
   public class ClassA
   {
      public virtual void XXX()
      {
         Console.WriteLine("ClassA XXX");
      }
   }

   /// <summary>
   /// ClassB
   /// </summary>
   public class ClassB:ClassA
   {
      public override void XXX()
      {
         Console.WriteLine("ClassB XXX");
      }
   }

   /// <summary>
   /// Class C
   /// </summary>
   public class ClassC : ClassB
   {
      public virtual new void XXX()
      {
         Console.WriteLine("ClassC XXX");
      }
   }

   /// <summary>
   /// Class D
   /// </summary>
   public class ClassD : ClassC
   {
      public override void XXX()
      {
         Console.WriteLine("ClassD XXX");
      }
   }

   /// <summary>
   /// Program: used to execute the method.
   /// Contains Main method.
   /// </summary>
   public class Program
   {
      private static void Main(string[] args)
      {
         ClassA a = new ClassD();
         ClassB b = new ClassD();
         ClassC c=new ClassD();
         ClassD d=new ClassD();

         a.XXX();
         b.XXX();
         c.XXX();
         d.XXX();

         Console.ReadKey();
      }
   }
```

Output:

```
ClassB XXX
ClassB XXX
ClassD XXX
ClassD XXX
```

Explanation:

One last explanation of virtual and override will be a bit complex.
The first output, ClassB XXX, is the outcome of the statement a.XXX();. We have the method XXX marked virtual in class ClassA. Therefore, when using new keyword, we now proceed to class ClassB and not ClassD as explained earlier. Here, XXX has an override and since C# knows that class ClassC inherits this function XXX. In the class ClassC, since it is marked as new, C# will now go back and do not proceed further to class ClassD. Finally the method XXX gets called from class ClassB as shown in the output above.

If we change the method XXX in class ClassC to override, then C# will proceed to class ClassD and call the XXX of class ClassD as it overrides the XXX of ClassC.

```
   /// <summary>
   /// Class C
   /// </summary>
   public class ClassC : ClassB
   {
      public override void XXX()
      {
         Console.WriteLine("ClassC XXX");
      }
   }
```

Remove the override from XXX in class ClassD and the method will get invoked from class ClassC as the default is new.

When we talk about object b, everything seems similar to object a, as it overrides the XXX of class ClassA. When we restore back the defaults, let's have a look at the third line. Object c here looks like ClassC. In class ClassC, XXX() is a new and therefore it has no connection with the earlier XXX methods. In class ClassD, we actually override the XXX of class ClassC and so the XXX of ClassD gets invoked. Just remove the override and then it will get invoked from class ClassC. The object d does not follow any of the above protocols as both the sides of the equal sign are of same data types.

Point to remember: An override method is a method that has the override modifier included on it. This introduces a new implementation of a method. We can't use the modifiers such as new, static or virtual along with override. But abstract is permitted.

Another example,

```
/// <summary>
   /// Class A
   /// </summary>
   public class ClassA
   {
      public virtual void XXX()
      {
         Console.WriteLine("ClassA XXX");
      }
   }

   /// <summary>
   /// ClassB
   /// </summary>
   public class ClassB:ClassA
   {
      public override void XXX()
      {
         base.XXX();
         Console.WriteLine("ClassB XXX");
      }
   }

   /// <summary>
   /// Class C
   /// </summary>
   public class ClassC : ClassB
   {
      public override void XXX()
      {
         base.XXX();
         Console.WriteLine("ClassC XXX");
      }
   }

   /// <summary>
   /// Class D
   /// </summary>
   public class ClassD : ClassC
   {
      public override void XXX()
```

```
        {
            Console.WriteLine("ClassD XXX");
        }
    }

    /// <summary>
    /// Program: used to execute the method.
    /// Contains Main method.
    /// </summary>
    public class Program
    {
        private static void Main(string[] args)
        {
            ClassA a = new ClassB();
            a.XXX();
            ClassB b = new ClassC();
            b.XXX();
            Console.ReadKey();
        }
    }
```

Output:

```
ClassA XXX
ClassB XXX
ClassA XXX
ClassB XXX
ClassC XXX
```

When we use the reserved keyword base, we can access the base class methods. Here no matter XXX is virtual or not, it will be treated as non virtual by the keyword base. Thus the base class XXX will always be called. The object a already knows that XXX is virtual. When it reaches to ClassB, it sees base.XXX() and hence it calls the XXX method of ClassA. But in second case, it first goes to class ClassC, here it calls the base.XXX, i.e. the method XXX of class ClassB, which in return invokes method XXX of class ClassA.

The infinite loop:

```
/// <summary>
    /// Class A
    /// </summary>
    public class ClassA
    {
        public virtual void XXX()
        {
            Console.WriteLine("ClassA XXX");
        }
```

```
  }

  /// <summary>
  /// ClassB
  /// </summary>
  public class ClassB:ClassA
  {
    public override void XXX()
    {
      ((ClassA)this).XXX();
      Console.WriteLine("ClassB XXX");
    }
  }


  /// <summary>
  /// Program: used to execute the method.
  /// Contains Main method.
  /// </summary>
  public class Program
  {
    private static void Main(string[] args)
    {
      ClassA a = new ClassB();
      a.XXX();

    }
  }
```

Output:

Error: {Cannot evaluate expression because the current thread is in a stack overflow state.}

In this kind of case no casting will stop the infinite loop. Therefore even though *this* is being cast to a class ClassA, it will always call XXX from class ClassB and not ClassA. So we get no output.

1. **Summary:**

Let's summarize all the point's to remember we got in the big article.

1. In C#, a smaller object could be equated to a bigger object.
2. The override modifier is needed as the derived class methods will get first priority and be called upon.
3. These modifiers like new and override can only be used if the method in the base class is a virtual method. Virtual means that the base class is granting us permission to invoke the method from the derived class and not the base class. But, we have to add the modifier override if our derived class method has to be called.

4. If the base class object declared the method virtual and the derived class used the modifier override, the derived class method will get called. Otherwise the base class method will get executed. Therefore for virtual methods, the data type created is decided at run time only.
5. All the methods not marked with virtual are non virtual, and the method to be called is decided at compile time, depending upon the static data type of the object.
6. An override method is a method that has the override modifier included on it. This introduces a new implementation of a method. We can't use the modifiers such as new, static or virtual along with override. But abstract is permitted.

Conclusion:

In this article I have tried to categorize the sections under different headings for the sake of readability.
In this article we learnt the concept of run time polymorphism and inheritance. We covered most of the scenarios by doing hands-on lab.

About Author

Akhil Mittal works as a Sr. Analyst in **Magic Software** and have an experience of more than 8 years in C#.Net. He is a codeproject and a c-sharpcorner MVP (Most Valuable Professional). Akhil is a B.Tech (Bachelor of Technology) in Computer Science and holds a diploma in Information Security and Application Development from CDAC. His work experience includes Development of Enterprise Applications using C#, .Net and Sql Server, Analysis as well as Research and Development. His expertise is in web application development. He is a MCP (Microsoft Certified Professional) in Web Applications (MCTS-70-528, MCTS-70-515) and .Net Framework 2.0 (MCTS-70-536).