Diving Into OOP

Akhil Mittal

Sr. Analyst (Magic Software)

C# corner MVP

INDEX

Page no

About Author

Akhil Mittal is a C# Corner MVP, author, blogger and currently working as a Sr. Analyst in Magic Software (http://www.magicsw.com) and have an experience of more than 8 years in C#.Net. He has done B.Tech in Computer Science and holds a diploma in Information Security and Application Development and also having experience in Development of Enterprise Applications using C#, .Net and Sql Server, Analysis as well as Research and Development. He is a MCP in Web Applications (MCTS-70-528, MCTS-70-515) and .Net Framework 2.0 (MCTS-70-536).

Akhil Mittal

Sr. Analyst (Magic Software)

C# corner MVP

Diving Into OOP

Diving Into OOP: Polymorphism and Inheritance (Early Binding/Compile Time Polymorphism)

OOP

1. What is OOP and what is the advantage of OOP?

OOP stands for "Object-Oriented Programming". Remember, it's OOP, not OOPS; "S" may stand for system, synopsis, structure and so on. OOP is a programming approach (usually called a programming paradigm or a methodology) entirely based on objects, instead of just functions and procedures like in procedural languages. It is like a programming language model organized around objects rather than "actions" and data rather than logic. An "object" in an OOP language refers to a specific type, or "instance", of a class. Each object has a structure exactly similar to other objects in a class, but can have individual properties/values. An object can also invoke methods specific to that object.

OOP makes it easier for developers to structure and organize software programs. Individual objects can be modified without affecting other aspects of the program therefore it is also easier to update and change programs written in object-oriented languages. Since the nature of software programs have grown larger over the years, OOP has made developing these large programs more manageable and readable.

2. OOP Concepts

The following are OOP concepts explained in brief, then we'll take the topics in detail.

1) Data Abstraction: Data Abstraction is a concept in which the internal and superfluous details of the implementation of a logic is hidden from an end user (who is using the program). A user can use any of the data and methods from the class without knowing about how this is created or what the complexity behind it is. In terms of a real world example, when we drive a bike and change the gears we don't need to care about how it works internally, like how a lever is pulled or how a chain is set.

2) Inheritance: Inheritance is a very popular concept in OOP .This provides a developer an advantage called reusability of code. Suppose a class is written having functions with specific logic, then we can derive that class into our newly created class and we don't need to write the logic again for derived class functions, we can use them as they are.

3) Data Encapsulation: Wrapping up of a member data and member functions of a class in a single unit is called encapsulation. The visibility of the member functions and data members is set via access modifiers used in the class.

4) Polymorphism: Poly means many and morphism means form. The concept refers to the many possible forms of behaviours of an object.

5) Message Communication: Message Communication means when an object es the call to a method of a class for execution.

3. Polymorphism

In this we will cover nearly all the scenarios of compile-type polymorphism, the use of the params keyword in detail and case studies or hands-on to multiple possible combinations of the thoughts coming to our mind while coding.

Method Overloading or Early Binding or Compile Time Polymorphism:

1. Let’s create a simple console application named InheritanceAndPolymorphism and add a class named Overload.cs and add three methods named DisplayOverload having varying parameters as follows.

Overload.cs

```
Public class Overload
{
  public void DisplayOverload(int a)
   {
     System.Console.WriteLine("DisplayOverload " + a);
   }
public void DisplayOverload(string a)
   {
     System.Console.WriteLine("DisplayOverload " + a);
   }
public void DisplayOverload(string a, int b)
   {
    System.Console.WriteLine("DisplayOverload " + a + b);
   }
}
```

In the main method in the Program.cs file, add the following code:

Program.cs

```
Class Program
{
   static void Main(string[] args)
    {
            Overload overload = new Overload();
            overload.DisplayOverload(100);
            overload.DisplayOverload("method overloading");
            overload.DisplayOverload("method overloading", 100);
            Console.ReadKey();
          }
      }
```

Now when you run the application, the output is:

Output:
```
DisplayOverload 100
DisplayOverload method overloading
DisplayOverload method overloading100
```

The class Overload contains three methods named DisplayOverload, they only differ in the datatype of the parameters they consist of. In C# we can have methods with the same name, but the datatypes of their parameters differ. This feature of C# is called method overloading. Therefore we need not remember many method names if a method differs in behavior, only providing different parameters to the methods can call a method individually.

Point to remember: C# recognizes the method by its parameters and not only by its name.
A signature signifies the full name of the method. So the name of a method or its signature is the original method name + the number and data types of its individual parameters.

If we run the project using the following code:

```
1. public void DisplayOverload() { }
2. public int DisplayOverload(){ }
```

We certainly get a compile time error as:

Error: Type 'InheritanceAndPolymorphism.Overload' already defines a member called 'DisplayOverload' with the same parameter types

Here we had two functions who differ only in the data type of the value that they return, but we got a compile time error, therefore another point to remember is:

Point to remember: The return value/parameter type of a method is never a part of the method signature if the names of the methods are the same. So this is not polymorphism.

If we run the project using the following code:

1. **static void** DisplayOverload(**int** a) {   }
2. **public void** DisplayOverload(**int** a) {   }
3. **public void** DisplayOverload(**string** a){ }

We again get a compile time error as in the following:

Error: Type 'InheritanceAndPolymorphism.Overload' already defines a member called 'DisplayOverload' with the same parameter types.

Can you differenciate with the modification done in code above, that we now have two DisplayOverload methods that accept an int (integer). The only difference is that one method is marked static. Here the signature of the methods will be considered the same as modifiers such as static are also not considered to be a part of the method signature.

Point to remember: Modifiers such as static are not considered to be a part of the method signature. If we run the program as in the following code, consider that the method signature is different now.

```
private void DisplayOverload(int a) {   }
private void DisplayOverload(out int a)
 {
   a = 100;
 }
private void DisplayOverload(ref int a) {   }
```

We again get a compile time error as in the following:

Error: Cannot define overloaded method 'DisplayOverload' because it differs from another method only on ref and out.

The signature of a method not only consists of the data type of the parameter but also the type/kind of parameter such as ref or out and so on. Method DisplayOverload takes an int with differing access modifiers, in other words out/ref and so on; the signature on each is different.

Point to remember: The signature of a method consists of its name, number and types of its formal parameters. The return type of a function is not part of the signature. Two methods cannot have the same signature and also non-members cannot have the same name as members.

4. Role of the Params Parameter in Polymorphism

A method can be called by any of four types of parameters; they are:

- by value,
- by reference,
- As an output parameter,
- Using parameter arrays.

As explained earlier the parameter modifier is never a part of the method signature. Now let's focus on Parameter Arrays.

A method declaration means creating a separate declaration space in memory. So anything created will be lost at the end of the method.

Running the following code:

```
public void DisplayOverload(int a, string a)  {   }
public void Display(int a)
 {
  string a;
 }
```

Results in a compile time error as in the following:

Error1: The parameter name 'a' is a duplicate.

Error2: A local variable named 'a' cannot be declared in this scope because it would provide a different meaning to 'a', that is already used in a 'parent or current' scope to denote something else.

Point to remember: Parameter names should be unique. And also we cannot have a parameter name and a declared variable name in the same function.

In the case of by value, the value of the variable is ed and in the case of ref and out, the address of the reference is ed.

When we run the following code:

Overload.cs

```
public class Overload
{
private string name = "Akhil";
public void Display()
{
Display2(ref name, ref name);
System.Console.WriteLine(name);         }
private void Display2(ref string x, ref string y)
{
System.Console.WriteLine(name);
x = "Akhil 1";
System.Console.WriteLine(name);
y = "Akhil 2";
System.Console.WriteLine(name);
name = "Akhil 3";
}
}
```

Program.cs

```
class Program
{
   static void Main(string[] args)
  {
   Overload overload = new Overload();
   overload.Display();
   Console.ReadKey();
   }
}
```

We get out put as,

Output:

```
Akhil
```

```
Akhil 1
Akhil 2
Akhil3
```

We are allowed to the same ref parameter as many times as we want. In the method Display the string name has a value of Akhil. Then by changing the string x to Akhil1, we are actually changing the string name to Akhil1 since the name is ed by reference.

Variables x and name refer to the same string in memory. Changing one changes the other. Again changing y also changes the name variable since they refer to the same string anyways. Thus variables x, y and name refer to the same string in memory.

When we run the following code:

Overload.cs

```
public class Overload
{
    public void Display()
    {
    DisplayOverload(100, "Akhil", "Mittal", "OOP");
    DisplayOverload(200, "Akhil");
    DisplayOverload(300);
    }
    private void DisplayOverload(int a, params string[] parameterArray)
    {
        foreach (string str in parameterArray)
        Console.WriteLine(str + " " + a);
        }
    }
```

Program.cs

```
class Program
{
    static void Main(string[] args)
    {
        Overload overload = new Overload();
        overload.Display();
        Console.ReadKey();
    }
}
```

We get the output:

Output:

```
Akhil 100
Mittal 100
OOP 100
Akhil 200
```

We will often get into a scenario where we would like to n number of parameters to a method. Since C# is very particular in parameter ing to methods, if we an int where a string is expected, it immediately breaks down. But C# provides a mechanism for ing n number of arguments to a method; we can do it using the params keyword.

Point to remember: This params keyword can only be applied to the last argument of the method. So the n number of parameters can only be at the end.

In the case of the method DisplayOverload, the first argument must be an integer, the rest can be from zero to an infinite number of strings.

If we add a method like:

```
private void DisplayOverload(int a, params string[] parameterArray, int b) { }
```

We get a compile time error as in the following:

Error: A parameter array must be the last parameter in a formal parameter list

Thus is is proved that the params keyword will be the last parameter in a method, this is already stated in the latest point to remember.

Overload.cs

```
public class Overload
{
    public void Display()
    {
       DisplayOverload(100, 200, 300);
       DisplayOverload(200, 100);
       DisplayOverload(200);
     }
     private void DisplayOverload(int a, params int[] parameterArray)
     {
        foreach (var i in parameterArray)
        Console.WriteLine(i + " " + a);
      }
 }
```

Program.cs

```
class Program
{
     static void Main(string[] args)
     {
        Overload overload = new Overload();
        overload.Display();
        Console.ReadKey();
      }
```

```
        }
```

When we run the code we get:

```
200 100
300 100
100 200
```

Therefore, Point to Remember: C# is very smart to recognize if the penultimate argument and the params have the same data type.

The first integer is stored in the variable a, the rest are made part of the array parameterArray.

1. **private void** DisplayOverload(**int** a**, params string**[][] parameterArray**) { }**
2. **private void** DisplayOverload**(int** a**, params string**[,] parameterArray**) { }**

For the preceding written code, we again get a compile time error and a new point to remember as well.

Error: The parameter array must be a single dimensional array

Point to remember: The error above says it all.

The data type of the params argument must be a single dimensional array. Therefore is allowed but not [,]. We also are not allowed to combine the params keyword with ref or out.

Overload.cs

```
public class Overload
{
      public void Display()
     {
         string[] names = {"Akhil", "Ekta", "Arsh"};
```

```
                DisplayOverload(3, names);
            }
            private void DisplayOverload(int a, params string[] parameterArray)
            {
                foreach (var s in parameterArray)
                Console.WriteLine(s + " " + a);
            }
    }
```

Program.cs

```
    class Program
    {
        static void Main(string[] args)
        {
            Overload overload = new Overload();
            overload.Display();
            Console.ReadKey();
         }
    }
```

Output

```
Akhil 3
Ekta 3
Arsh 3
```

We are therefore allowed to a string array instead of individual strings as arguments. Here names is a string array that has been initialized using the short form. Internally when we call the function DisplayOverload, C# converts the string array into individual strings.

Overload.cs

```
    public class Overload
    {
    public void Display()
```

```
{
string [] names = {"Akhil","Arsh"};
DisplayOverload(2, names, "Ekta");
}
private void DisplayOverload(int a, params string[] parameterArray)
{
foreach (var str in parameterArray)
Console.WriteLine(str + " " + a);
}
}
```

Program.cs

```
class Program
{
static void Main(string[] args)
{
Overload overload = new Overload();
overload.Display();
Console.ReadKey();
}
}
```

Output:

Error: The best overloaded method match for 'InheritanceAndPolymorphism.Overload.DisplayOverload(int, params string[])' has some invalid arguments

Error: Argument 2: cannot convert from 'string[]' to 'string'

So, we get two errors.

For the preceding code, C# does not permit mix and match. We assumed that the last string “Ekta” would be added to the array of string names or convert the names to individual strings and then the string “Ekta” would be added to it. Quite logical.
Internally before calling the function DisplayOverload, C# accumulates all the individual parameters and converts them into one big array for the params statement.

Overload.cs

```
public class Overload
{
    public void Display()
    {
        int[] numbers = {10, 20, 30};
        DisplayOverload(40, numbers);
        Console.WriteLine(numbers[1]);
     }
     private void DisplayOverload(int a, params int[] parameterArray)
     {
        parameterArray[1] = 1000;
      }
}
```

Program.cs

```
class Program
{
     static void Main(string[] args)
    {
        Overload overload = new Overload();
        overload.Display();
        Console.ReadKey();
     }
}
```

Output: 1000

We see that the output produced is the proof of concept. The member parameterArray[1] of array has an initial value of 20 and in the method DisplayOverload, we changed it to 1000.

So the original value changes, this shows that the array is given to the method DisplayOverload and that provides the proof.

Overload.cs

```
public class Overload
{
    public void Display()
    {
       int number = 102;
       DisplayOverload(200, 1000, number, 200);
       Console.WriteLine(number);
     }
     private void DisplayOverload(int a, params int[] parameterArray)
    {
        parameterArray[1] = 3000;
     }
}
```

Program.cs

```
class Program
{
    static void Main(string[] args)
    {
       Overload overload = new Overload();
       overload.Display();
       Console.ReadKey();
    }
}
```

Output: 102

In the preceding mentioned scenario C# creates an array containing 1000 102 and 200. We now change the second member of array to 3000 that has nothing to do with the variable number. As DisplayOverload has no knowledge of numebr, so how can DisplayOverload change the value of the int number? Therefore it remains the same.

Overload.cs

```
public class Overload
{
    public void Display()
    {
        DisplayOverload(200);
        DisplayOverload(200, 300);
        DisplayOverload(200, 300, 500, 600);
     }
     private void DisplayOverload(int x, int y)
    {
        Console.WriteLine("The two integers " + x + " " + y);
    }
    private void DisplayOverload(params int[] parameterArray)
    {
       Console.WriteLine("parameterArray");
    }
}
```

Program.cs

```
class Program
{
    static void Main(string[] args)
    {
      Overload overload = new Overload();
      overload.Display();
      Console.ReadKey();
     }
}
```

Output:

```
parameterArray
The two integers 200 300
parameterArray
```

Now we talk about method overloading. C# is extremely talented though partial. It does not appreciate the params statement and treats it as a stepchild. When we invoke DisplayOverload only with one integer, C# can only call the DisplayOverload that takes a params as a parameter since it matches only one int. An array can contain one member too. The fun is with the DisplayOverload that is called with two ints now. So here we have a dilemma. C# can call the params DisplayOverload or DisplayOverload with the two ints. As said earlier, C# treats the params as a second class member and therefore chooses the DisplayOverload with two ints.When there are more than two ints like in the third method call, C# is void of choice but to grudgingly choose the DisplayOverload with the params. C# opts for the params as a last resort before flagging an error.

Now for an example that is a bit trickier, yet important.

Overload.cs

```
public class Overload
{
    public static void Display(params object[] objectParamArray)
    {
    foreach (object obj in objectParamArray)
    {
      Console.Write(obj.GetType().FullName + " ");
    }
      Console.WriteLine();
    }
}
```

Program.cs

```
class Program
{
   static void Main(string[] args)
  {
    object[] objArray = { 100, "Akhil", 200.300 };
```

```
        object obj = objArray;
        Overload.Display(objArray);
        Overload.Display((object)objArray);
        Overload.Display(obj);
        Overload.Display((object[])obj);
        Console.ReadKey();
      }
}
```

Output:

```
System.Int32 System.String System.Double
System.Object[]
System.Object[]
System.Int32 System.String System.Double
```

In the first instance we are ing the method Display an array of objects that looks like an object. Since all the classes are derived from a common base class object, we can do that. The method Display gets an array of objects, objectParamArray. In the foreach object the class has a method named GetType that returns an object that looks like Type, that too has a method named FullName that returns the name of the type. Since each of three types are displayed. In the second method call of Display we are casting objArray to an object. Since there is no conversion available for converting an object to an object array, in other words object [], only a one-element object [] is created. It's the same case in the third invocation and the last explicitly casts to an object array.

For proof of concept, see the following.

Overload.cs

```
public class Overload
{
    public static void Display(params object[] objectParamArray)
    {
        Console.WriteLine(objectParamArray.GetType().FullName);
        Console.WriteLine(objectParamArray.Length);
        Console.WriteLine(objectParamArray[0]);
    }
}
```

Program.cs

```
class Program
{
    static void Main(string[] args)
    {
       object[] objArray = { 100, "Akhil", 200.300 };
       Overload.Display((object)objArray);
       Console.ReadKey();
     }
}
```

Output:

```
System.Object[]
1
System.Object[]
```

5. Conclusion

Diving Into OOP series we learned about compile-time polymorphism, it is also called early binding or method overloading. We catered most of the scenarios specific to polymorphism.We also learned about the use of powerful the params keyword and its use in polymorphism.

To sum up, let's list all the points to remember once more.

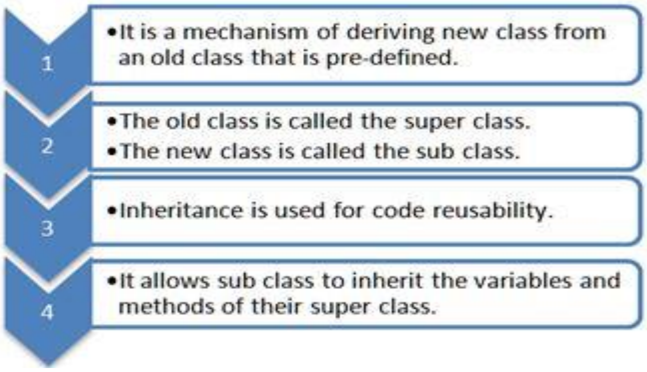
1. C# recognizes the method by its parameters and not only by its name.
2. The return value/parameter type of a method is never the part of method signature if the names of the methods are the same. So this is not polymorphism.
3. Modifiers such as static are not considered as part of the method signature.

4. The signature of a method consists of its name, number and types of its formal parameters. The return type of a function is not part of the signature. Two methods cannot have the same signature and also non-members cannot have the same name as members.
5. Parameter names should be unique. And also we cannot have a parameter name and a declared variable name in the same function.
6. In case of by value, the value of the variable is ed and in the case of ref and out, the address of the reference is ed.
7. This params keyword can only be applied to the last argument of the method. So the n number of parameters can only be at the end.
8. C# is very smart to recognize if the penultimate argument and the params have the same data type.
9. A Parameter Array must be a single dimensional array.

Diving Into OOP: Polymorphism and Inheritance

Introduction

Second part of the series will focus solely on inheritance in OOP. Let's define inheritance using some bullet points.



1
- It is a mechanism of deriving new class from an old class that is pre-defined.

2
- The old class is called the super class.
- The new class is called the sub class.

3
- Inheritance is used for code reusability.

4
- It allows sub class to inherit the variables and methods of their super class.

Inheritance in Action:

Let's do some hands-on. Create a console application and name it InheritanceAndPolymorphism. Add a class named ClassA and a class named ClassB, with the following code:

ClassA

```
class ClassA
{
}
```

ClassB

```
class ClassB
{
  public int x = 100;
  public void Display1()
  {
    Console.WriteLine("ClassB Display1");
  }
  public void Display2()
  {
    Console.WriteLine("ClassB Display2");   }
 }
```

We see class ClassA is empty and we added two methods in the class ClassB, Display1 and Display2. We also have a variable x declared and defined with a value 100.

Now in the main method of Program.cs, write the following code:

Program.cs

```
class Program
{
  static void Main(string[] args)
  {
    ClassA a = new ClassA();
     a.Display1();
   }
}
```

If we run the code, we immediately result in the compile time error:

Error: 'InheritanceAndPolymorphism.ClassA' does not contain a definition for 'Display1' and no extension method 'Display1' accepting a first argument of type

'InheritanceAndPolymorphism.ClassA' could be found (are you missing a using directive or an assembly reference?)

That is very obvious, we don't have a definition of the Display1 method in ClassA, nor can we access the same method using a ClassA instance because it is not derived from any such class like ClassB that contains a Display1 method. The class ClassA does not contain any code or variable defined . An empty class does not throw any error as we are able to instantiate an object that looks like an instance of ClassA. The error comes about because the class ClassA has no method called Display1. However the class ClassB has a method named Display1. Guess how fun it could be if we are allowed to access all the code of classB from ClassA itself.

Just derive the class ClassA from ClassB using the ":" operator as in the code shown below:

ClassA

```
class ClassA:ClassB
{
}
```

ClassB

```
class ClassB
{
  public int x = 100;
  public void Display1()
  {
    Console.WriteLine("ClassB Display1");
  }
  public void Display2()
  {
    Console.WriteLine("ClassB Display2");
  }
}
```

Program.cs

```
class Program
{
  static void Main(string[] args)
  {
    ClassA a = new ClassA();
    a.Display1();
    Console.ReadKey();
  }
}
```

And now run the code as it was, we get an output now.

Output:

ClassB Display1

In other words, now ClassA can access the inherited public methods of ClassB. The error vanishes and the Display1 in ClassB is invoked. If after the name of a class we specify: ClassB, in other words the name of another class then many changes occur at once. ClassA is now said to have been derived from ClassB. What that means is that all the code we wrote in ClassB can now be accessed and used in ClassA. It is if we actually wrote all the code that is contained in ClassB in ClassA. If we had created an instance that looks like that of ClassB then everything that the instance could do, now an instance of ClassA can also do. But we have not written a line of code in ClassA. We are made to believe that ClassA has one variable x and two functions Display1 and Display2 since ClassB contains these two functions. Therefore we enter into the concepts of inheritance where ClassB is the base class and ClassA is the derived class.

Child inherit properties from father
Derived class inherits properties from its base class

Let's use another scenario. Suppose we get into a situation where ClassA also has a method of the same name as of in ClassB. Let's define a method Derive1 in ClassA too, so our code for classA becomes:

```
class ClassA:ClassB
{
  public void Display1()
  {
    System.Console.WriteLine("ClassA Display1");
  }
}
```

ClassB

```
class ClassB
{
  public int x = 100;
  public void Display1()
  {
    Console.WriteLine("ClassB Display1");
  }
  public void Display2()
  {
    Console.WriteLine("ClassB Display2");
  }
}
```

Now if we run the application using the following code snippet for the Program class:

```
class Program
{
  static void Main(string[] args)
  {
    ClassA a = new ClassA();
    a.Display1();
    Console.ReadKey();
  }
}
```

The question is what will happen now? What will be the output? Will there be any output or any compilation error. Ok, let's run it and we get the output:

ClassA Display1

But did you notice one thing, we also got a warning when we run the code,

Warning: 'InheritanceAndPolymorphism.ClassA.Display1()' hides inherited member 'InheritanceAndPolymorphism.ClassB.Display1()'. Use the new keyword if hiding was intended.

Point to remember: No one can stop a derived class from having a method with the same name already declared in its base class.
So, ClassA undoubtedly can contain the Display1 method, that is already defined with the same name in ClassB.

When we invoke a.Display1(), C# first checks whether the class ClassA has a method named Display1. If it does not find it, it checks in the base class. Earlier Display1 method was only available in the base class ClassB and hence got executed. Here since it is there in ClassA, it gets called from ClassA and not ClassB.

Point to remember: Derived classes get a first chance at execution, then the base class.

The reason for this is that the base class may have a number of mehtods and for various reasons we may not be satisfied with what they do. We should have the full right to have our copy of the method to be called. In other words the derived classes methods override the ones defined in the baseclass.

What happens if we call the base class Display1 method too with the base keyword in the derived class, in other words by using base.Display1(), so our ClassA code will be:

ClassA

```
class ClassA:ClassB
{
  public void Display1()
  {
    Console.WriteLine("ClassA Display1");
    base.Display1();
  }
}
```

ClassB

```
class ClassB
{
  public int x = 100;
  public void Display1()
  {
    Console.WriteLine("ClassB Display1");
  }
  public void Display2()
  {
    Console.WriteLine("ClassB Display2");
  }
}
```

Program.cs

```
class Program
{
  static void Main(string[] args)
  {
    ClassA a = new ClassA();
    a.Display1();
    Console.ReadKey();
  }
}
```

Output:

```
ClassA Display1
ClassB Display1
```

We see here first our ClassA Display1 method is called and then the ClassB Display1 method.

Now if you want the best of both the classes then you may want to call the base classes (ClassB) Display1 first and then yours or vice versa. To do this, C# gives you a free keyword, called base. The keyword base can be used in any of derived class. It means call the method off the base class.

Thus base.Display1 will call the method Display1 from ClassB, the base class of ClassA as defined earlier.

Point to remember: A reserved keyword named “base” can be used in a derived class to call the base class method.

What if we call the Display2 method from the base class with an instance of the derived class ClassA?

```
/// <summary>
/// ClassB: acting as base class
/// </summary>
class ClassB
{
  public int x = 100;
  public void Display1()
  {
    Console.WriteLine("ClassB Display1");
  }
  public void Display2()
  {
    Console.WriteLine("ClassB Display2");
  }
}
/// <summary>
/// ClassA: acting as derived class
/// </summary>
class ClassA : ClassB
{
  public void Display1()
  {
    Console.WriteLine("ClassA Display1");
```

```
        base.Display2();
    }
}
/// <summary>
/// Program: used to execute the method.
/// Contains Main method.
/// </summary>
class Program
{
    static void Main(string[] args)
    {
        ClassA a = new ClassA();
        a.Display1();
        Console.ReadKey();
    }
}
```

Output:

```
ClassA Display1
ClassB Display2
```

In the code above we only made just a small change, base.Display1 was replaced by base.Display2. In this specific scenario method Display2 from the class ClassB is called. Base is usually a very general purpose. It lets us access members of the base class from the derived class as explained earlier. We cannot use the base in ClassB since ClassB is not derived from any class as per our code. So its done that the base keyword can only be used in derived classes.

Let us use another case.

```
/// <summary>
/// ClassB: acting as base class
/// </summary>
```

```
class ClassB
{
  public int x = 100;
  public void Display1()
  {
    Console.WriteLine("ClassB Display1");
  }
}

/// <summary>
/// ClassA: acting as derived class
/// </summary>

class ClassA : ClassB
{
  public void Display2()
  {
    Console.WriteLine("ClassA Display2");
  }
}

/// <summary>
/// Program: used to execute the method.
/// Contains Main method.
/// </summary>

class Program
{
  static void Main(string[] args)
  {
    ClassB b = new ClassB();
    b.Display2();
    Console.ReadKey();
  }
}
```

Output:

Error: 'InheritanceAndPolymorphism.ClassB' does not contain a definition for 'Display2' and no extension method 'Display2' accepting a first argument of type 'InheritanceAndPolymorphism.ClassB' could be found (are you missing a using directive or an assembly reference?)

Point to remember: Inheritance does not work backwards.

So we got an error. Since we see, ClassA is derived from ClassB, in other words ClassB is the base class. Therefore class ClassA can use all the members of class ClassB. Inheritance does not have backwords compatibility, whatever members ClassA contains does not permeate upwards to ClassB. When we tried to access the Display2 method of classA from the instance of class ClassB, it cannot provide it to class ClassB and thus an error occurs.

Point to remember: Except constructors and destructors, a class inherits everything from its base class.

If a class ClassC is derived from class ClassB, which in turn has been derived from class ClassA, then ClassC will inherit all the members declared in ClassB and also of ClassA. This is called transitive concept in inheritance. A derived class may inherit all the members of the base class but it cannot remove members off that base class. A derived class can however hide members of the base class by creating methods by the same name. The original member/method of the base class remains unmodified and unaffected by whatever happens in the derived class. It remains unchanged in the base class, in other words simply not visible in the derived class.

A class member could be of two types, in other words either a static member that directly belongs to a class or an instance member that is accessed through an instance of that class and belongs to that specific instance only. An instance member is accessible only through the object of the class and not directly by the class. The default member declared in the class are non-static, we just need to make them static using the static keyword.
All classes derive from a common base class named *Object*. So Object is the mother of all classes.

If we do not derive any class from any other class then it's the responsibility of C# to add: object by itself to the class definition. Object is the only class that is not derived from any other class. It is the ultimate base class for all the classes.

Suppose ClassA is derived from ClassB as in our case, but ClassB is not derived from any class.

```
public class ClassB
{
}
public class ClassA : ClassB
{
}
```

C# automatically adds :object to ClassB i.e., the code at compile time becomes,

```
public class ClassB:object
{
}
public class ClassA : ClassB
{
}
```

But as per theory we say ClassB is the direct base class of ClassA, so the classes of ClassA are ClassB and object.

Let's go for another case,

```
public class ClassW : System.ValueType
{
}
public class ClassX : System.Enum
{
}
public class ClassY : System.Delegate
{
}
public class ClassZ : System.Array
{
}
```

Here we have defined four classes, each derived from a built in class in C#, let's run the code.

We get many compile time errors as in the following:

Errors:

'InheritanceAndPolymorphism.ClassW' cannot derive from special class 'System.ValueType'
'InheritanceAndPolymorphism.ClassX' cannot derive from special class 'System.Enum'
'InheritanceAndPolymorphism.ClassY' cannot derive from special class 'System.Delegate'
'InheritanceAndPolymorphism.ClassZ' cannot derive from special class 'System.Array'

Don't be scared.

Did you notice the word special class. Our classes defined cannot inherit from special built-in classes in C#.

Point to remember: In inheritance in C#, custom classes cannot derive from special built-in C# classes like System.ValueType, System.Enum, System.Delegate, System.Array and so on.

One more case:

```
public class ClassW
{
}
public class ClassX
{
}
public class ClassY : ClassW, ClassX
{
}
```

In the preceding case, we see three classes, ClassW, ClassX and ClassY. ClassY is derived from ClassW and ClassX. Now if we run the code, what would we get?

Compile time Error: Class 'InheritanceAndPolymorphism.ClassY' cannot have multiple base classes: 'InheritanceAndPolymorphism.ClassW' and 'ClassX'

So one more **Point to remember**: A class can only be derived from one class in C#. C# does not support multiple inheritance by means of class*.

*Multiple inheritance in C# can be accomplished by the use of interfaces, we are not discussing interfaces in this article.

We are not allowed to derive from more than one class, thus every class can have only one base class. In another case, suppose we try to write the following code:

```
public class ClassW:ClassY
{
}
public class ClassX:ClassW
{
}
public class ClassY : ClassX
{
}
```

The code is quite readable and simple, ClassW is derived from ClassY, ClassX is derived from ClassW, and ClassY in turn is derived from ClassX. So no problem of multiple inheritance, so our code should build successfully, let's compile the code.What do we get? Again a compile time error as in the following:

Error: Circular base class dependency involving 'InheritanceAndPolymorphism.ClassX' and 'InheritanceAndPolymorphism.ClassW'

Point to remember: Circular dependency is not allowed in inheritance in C#. ClassX is derived from ClassW that was derived from ClassY and ClassY was again derived from ClassX, that caused a circular dependency in three classes, yet that is logically impossible.

Equalizing the Instances/Objects:

Let's directly start with a real case.

ClassB

```
public class ClassB
{
  public int b = 100;
}
```

ClassA

```
public class ClassA
{
  public int a = 100;
}
```

Program.cs

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

We are here trying to equate two objects or two instances of each of two classes. Let's compile the code.

We get a compile-time error.

Error:

Cannot implicitly convert type 'InheritanceAndPolymorphism.ClassB' to 'InheritanceAndPolymorphism.ClassA'
Cannot implicitly convert type 'InheritanceAndPolymorphism.ClassA' to 'InheritanceAndPolymorphism.ClassB'

Inheritance And Polymorphism is the namespace that I used for my console application, so no need to be scared of that word, just ignore it.

C# works on rules, it will never allow you to equate objects of various independent classes to each other. Therefore we cannot equate an object classA of ClassA to classB of ClassB or vice versa. It does not matter whether the classes contain a similar structure and their variables are initialized to similar integer values, even if we do:

```
public class ClassB
{
  public int a = 100;
}
public class ClassA
{
  public int a = 100;
}
```

I just changed int b of ClassB to int a. In this case too, to equate an object is not allowed and not possible.

C# is also very particular when it comes to dealing with data types.
There is however one way to to this. By this way that we'll discuss, one of the errors will disappear. The only time we are allowed to equate dissimilar data types is only when we derive from them .Check out the code mentioned below. Let's discuss this in detail, when we create an object of ClassB by declaring new, we are creating two objects at one go, one that looks like ClassB and the other that looks like an object, in other words derived from the Object class (in other words ultimate base class). All classes in C# are finally derived from object. Since ClassA is derived from ClassB, when we declare new ClassA, we are creating 3 objects, one that looks like ClassB, one that looks like ClassA and finally that looks like the object class.

```
public class ClassB
{
  public int b = 100;
}
public class ClassA:ClassB
{
  public int a = 100;
}

/// <summary>
/// Program: used to execute the method.
/// Contains Main method.
/// </summary>
```

```
public class Program
{
  private static void Main(string[] args)
  {
    ClassB classB = new ClassB();
    ClassA classA = new ClassA();
    classA = classB;
    classB = classA;
  }
}
```

We just derived ClassA from ClassB, this is something we can do, we learned a lot about this in this article. Now when we compile the code, we get:

Error: Cannot implicitly convert type 'InheritanceAndPolymorphism.ClassB' to 'InheritanceAndPolymorphism.ClassA'. An explicit conversion exists (are you missing a cast?)

Like I said, C# is very particular about equating objects.

Thus when we write classA = classB, classA looks like ClassA instance, ClassB and object and classB looks like ClassB, there is a match at ClassB.Result? No error Smile | :) . Even though classB and classA have the same values, using classB we can only access the members of ClassB, even though had we used classA we could access ClassA also. We have devalued the potency of classB. The error occurs at classA = classB. The class ClassA has ClassB and more. We cannot have a base class on the right and a derived class on the left. classB only represents a ClassB whereas classA expects a ClassA which is a ClassA and ClassB.

Point to remember: We can equate an object of a base class to a derived class but not vice versa.

Another code **snippet:**

```
public class ClassB
{
  public int b = 100;
}
```

```
public class ClassA:ClassB
{
  public int a = 100;
}
/// <summary>
/// Program: used to execute the method.
/// Contains Main method.
/// </summary>
public class Program
{
  private static void Main(string[] args)
  {
    ClassB classB = new ClassB();
    ClassA classA = new ClassA();
    classB=classA;
    classA = (ClassA)classB;
  }
}
```

Although we violated a C# rule of equating objects, we did not get any compiler error because of the cast we did to the object. A () is called a cast. Within the brackets the name of the class is put. A cast basically proves to be a great leveller. When we intend to write classA = classB, C# expects the right-hand side of the equal to to be a classA, in other words a ClassA instance . But it finds classB in other words a ClassB instance. So when we apply the cast , we actually try to convert an instance of ClassB to an instance of ClassA. This approach satisfies the rules of C# on only equating similar object types. Remember, it is only for the duration of the line that classB becomes a ClassA and not a ClassB.

Now if we remove ClassB as a base class to class ClassA as in following code and try to typecast classA to ClassB object then something happens.

```
public class ClassB
{
  public int b = 100;
}
Public class ClassA // Removed ClassB as base class
```

```
{
  public int a = 100;
}

/// <summary>
/// Program: used to execute the method.
/// Contains Main method.
/// </summary>

public class Program
{
  private static void Main(string[] args)
  {
    ClassB classB = new ClassB();
    ClassA classA = new ClassA();
    classB = (ClassB)classA;
    classA = (ClassA)classB;
  }
}
```

Output

Error:
Cannot convert type 'InheritanceAndPolymorphism.ClassA' to 'InheritanceAndPolymorphism.ClassB'
Cannot convert type 'InheritanceAndPolymorphism.ClassB' to 'InheritanceAndPolymorphism.ClassA'

***'InheritanceAndPolymorphism' :** Namespace I used in my application , so ignore that.

So we see that casting only works if one of the two classes is derived from one another. We cannot cast any two objects to each other.

The following is one last example.

```
/// <summary>
/// Program: used to execute the method.
/// Contains Main method.
/// </summary>

public class Program
{
   private static void Main(string[] args)
   {
      int integerA = 10;
      char characterB = 'A';
      integerA = characterB;
      characterB = integerA;
   }
}
```

We run the code,

Output:

Error: Cannot implicitly convert type 'int' to 'char'. An explicit conversion exists (are you missing a cast?)

Point to remember: We cannot implicitly convert an int to a char but a char can be converted to an int.

Conclusion

In this part of our series we learned about inheritance. We took various scenarios and practical examples back to back for a deep to understanding. Next we'll be discussing runtime polymorphism. Inheritance plays a very important role in runtime polymorphism.

The following is a list of all our points to remember.

1. No one can stop a derived class to have a method with the same name already declared in its base class.
2. Derived classes get a first chance at execution, then the base class.
3. A reserved keyword named "base" can be used in a derived class to call the base class method.
4. Inheritance does not work backwards.
5. Except constructors and destructors, a class inherits everything from its base class.
6. In inheritance in C#, custom classes cannot derive from special built-in C# classes like System.ValueType, System.Enum, System.Delegate, and System.Array and so on.
7. A class can only be derived from one class in C#. C# does not support multiple inheritance by means of class.
8. Circular dependency is not allowed in inheritance in C#. ClassX is derived from ClassW that was derived from ClassY and ClassY was again derived from ClassX, that caused circular dependency in three classes that is logically impossible.
9. We can equate an object of a base class to a derived class but not vice versa.
10. We cannot implicitly convert an int into a char, but char can be converted to int.

Diving Into OOP: Polymorphism and Inheritance (Dynamic Binding/Run Time Polymorphism)

Introduction:

This part of the article series will focus more on runtime polymorphism also called late Binding. We'll use the same technique of learning, less theory and more hands-on. We'll use small code snippets to learn the concept more deeply. Mastering this concept is like learning more than 50% of OOP.

Runtime Polymorphism or Late Binding or Dynamic Binding:

In simple C# language, in runtime polymorphism or method overriding we can override a method in a base class by creating a similar method in the derived class; this can be done using the 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");
    }
}
```

Program.cs

```
/// <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 number 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 become:

```
/// <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");
  }
}
```

Program.cs

```
/// <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, in other words run the code, what do we get?

Output

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

But with the compiler 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#, we can equate an object of a base class to a derived class but not vice versa.

Class ClassB is the super-class of class ClassA. That means ClassA is the derived class and ClassB is the base class. The class ClassA comprises ClassB and something more. So we can conclude that an object of ClassA is bigger than an object of ClassB. Since ClassA is inherited from ClassB, it contains its own methods and properties and 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 is 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 understand that it will call them from ClassB.

Object x looks like that of ClassA, in other words the derived class. It is initialized to an object that looks like ClassA. When we call AAA, the BBB and CCC methods through x, calls AAA, BBB and CCC from ClassA.

Now there is a 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 that does not provide 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 an override behind AAA and new behind the BBB methods of ClassA, in other words 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 the 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 the error after we added these two modifiers to 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 the 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 necessary since 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 the virtual modifier to 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 that literally means, forget about the data type of z which is ClassB, call AAA from ClassA since it overrides the AAA of the base class. The override modifier is needed since 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 in fact telling C# that this AAA is similar to the AAA of the one in the base class.

The new keyword acts the exact opposite to the 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, BBB, as in the base class, but that is only a coincidence. Since z looks like ClassB, the BBB of ClassB is 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 need to add the modifier override if our derived class method must 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 the reverse will result in an error. This leads to an instance of a base class being initialized to an instance of the derived class. So the question is now, 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 declares the method virtual and the derived class uses the modifier override, the derived class method will be called. Otherwise the base class method will be executed. Therefore for virtual methods, the data type created is decided only at run time.

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 preceding rules would apply. Whenever we have a mismatch, we always need rules to resolve the mismatch. So the result could be a scenario where an object to a base class can call a method in the derived class.

The object y that looks like it is of ClassB but is initialized here to the derived class 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 is called from the 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 that has nothing to do with the one in the base class.

They only accidentally share the same name. So since there is no method called BBB (as it is a new BBB) in the derived class, the one from the base class is 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 the class ClassA and not the one in the base class, 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 that 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 is called from ClassB. In the case of method BBB, C# now looks into the class ClassC. Here it does not find a method named BBB and so ultimately propagates and now looks into class ClassA. Therefore the preceding rules repeat on and on and it is called from the class ClassB. In the case of x.CCC, in the class ClassC, it is already marked new by default and therefore this method has nothing to do with the one declared in the class ClassB. So the one from ClassC is not called but the one from class ClassB where it is marked as override.

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

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

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

The last object z looks like 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 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() is called from the class ClassC. In the case of BBB(), BBB() is marked virtual in the class ClassB and new in ClassA, but since there is no method BBB in ClassC, none of the modifier matters at all in this case. Finally it gets invoked from class ClassA. Finally for method CCC, in class ClassC it is marked as new. Hence it has no relation with the CCC in class ClassA that results in the method CCC getting invoked from ClassA but not ClassB.

The following is 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 preceding example the code is very much self explanatory, the output that 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 since 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 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. Since 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();
    }
}
```

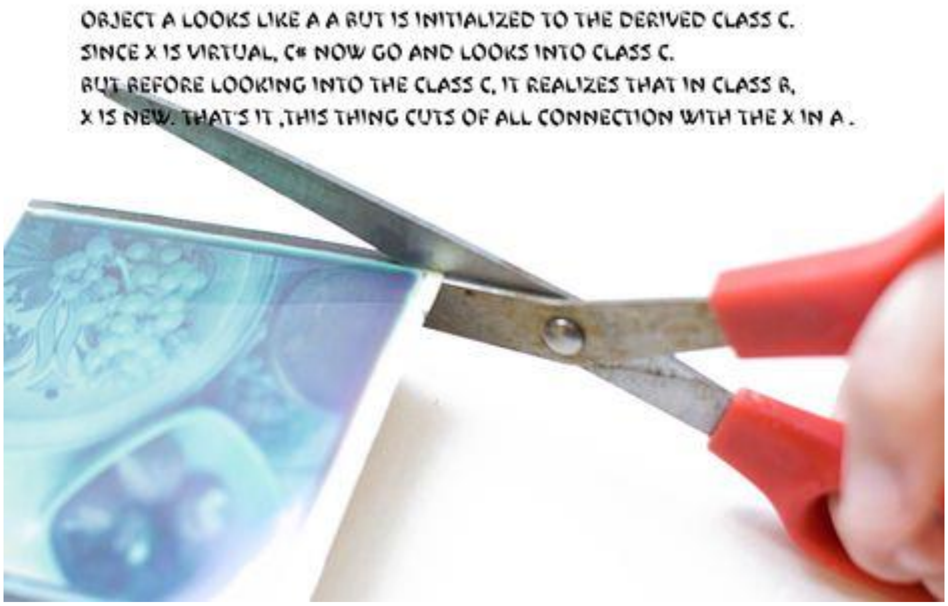
Output:

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

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

Output:

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



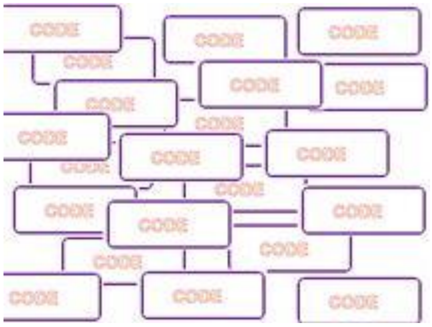
OBJECT A LOOKS LIKE A A BUT IS INITIALIZED TO THE DERIVED CLASS C.
SINCE X IS VIRTUAL, C# NOW GO AND LOOKS INTO CLASS C.
BUT BEFORE LOOKING INTO THE CLASS C, IT REALIZES THAT IN CLASS B,
X IS NEW. THAT'S IT ,THIS THING CUTS OF ALL CONNECTION WITH THE X IN A .

Yes, that's the problem with virtual methods. Sometimes they are too confusing; the result is entirely different from what we expect. Object a looks like a A but is initialized to the derived class C. Since X is virtual, C# now 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 connections 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. Since X in class B is marked as a new method, having nothing to do with the class A, class C inherits a new that 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 is 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, that makes it a unique method X. Sadly; the X in C has the override modifier that 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 then 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 is always called, irrespective of whether one exists in the base class or derived class. In a virtual method the run-time type of the object determines which method is 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 that is always called.

Runtime Polymorphism with four classes



OK! We did a lot of coding. How about if I tell you that we will add one more class to our code, yes that is class ClassD. So we dive deeper into the concepts of Polymorphism and Inheritance.

We add one more class to the three-class 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 the 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 does not proceed further to class ClassD. Finally the method XXX is 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 since 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 since the default is new.

When we talk about object b, everything seems similar to object a, since it overrides the XXX of class ClassA.

When we restore 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 is invoked. Just remove the override and then it will be invoked from class ClassC. The object d does not follow any of the preceding protocols since both sides of the equal sign are of the 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.

The following is 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 whether or not XXX is virtual, 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 ClassB, it sees base.XXX() and hence it calls the XXX method of ClassA. But in the second case, it first goes to class ClassC, where it calls the base.XXX, in other words the method XXX of class ClassB, that in turn invokes the method XXX of the 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.

Summary:

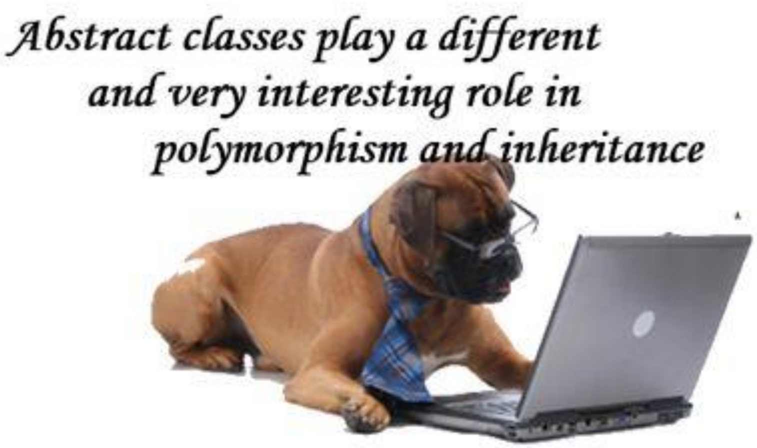
Let's summarize all the points to remember from this big article.

1. In C#, we can equate an object of a base class to a derived class but not vice versa.
2. The override modifier is necessary since 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 need to add the modifier override if our derived class method must 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 be called. Otherwise the base class method will be 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.

Diving Into OOP: Polymorphism and Inheritance (All About Abstract Classes in C#)

Introduction:

We have learned a lot about polymorphism and Inheritance. In this series “Diving Into OOP”, we'll discuss the hottest and most exciting topic of OOP in C#, Abstract Classes. The concept of Abstract classes is the same for any other language, but in C# we deal with it in a bit of a different way. Abstract classes play a different and very interesting role in polymorphism and inheritance. We'll cover all the aspects of abstract classes with our hands-on lab and theory as an explanation to what output we get.



2. Abstract Classes:

The following definition is from the **MSDN**:

"The abstract keyword enables you to create classes and class members that are incomplete and must be implemented in a derived class. An abstract class cannot be instantiated. The purpose of an abstract class is to provide a common definition of a base class that multiple derived classes can share. For example, a class library may define an abstract class that is used as a parameter to many of its functions, and require programmers using that library to provide their own implementation of the class by creating a derived class.

Abstract classes may also define abstract methods. This is accomplished by adding the keyword abstract before the return type of the method."

3. Abstract classes in action:

Add a console application named "InheritanceAndPolymorphism" in your Visual Studio. You'll get a class named Program; just add one more class named ClassA. Note that ClassA should be marked abstract and add the following code to ClassA and the Program class:

```
using System;

namespace InheritanceAndPolymorphism
{
    public abstract class ClassA
    {
    }

     /// <summary>
     /// Program: used to execute the method.
    /// Contains Main method.
    /// </summary>

   public class Program
   {
        private static void Main(string[] args)
        {
           ClassA classA = new ClassA();
           Console.ReadKey();
        }
    }
}
```

Compile the code.

Output:

Compile time error: Cannot create an instance of the abstract class or interface 'InheritanceAndPolymorphism.ClassA'

Point to remember: We cannot create an object of abstract class using the new keyword.

Now we go into understanding the concept. No power can stop abstract keyword to be written before a class. It acts as a modifier to the class. We cannot create an object of abstract class using the new keyword. It seems that the class is useless for us since we cannot use it for other practical purposes as we previously did.

4. Non abstract method definition in abstract class:

Let's add some code to our abstract class as in the following:

```
/// <summary>
/// Abstract class ClassA
/// </summary>

public abstract class ClassA
{
   public int a;
   public void XXX()
   {
   }
}

/// <summary>
/// Program: used to execute the method.
/// Contains Main method.
/// </summary>

public class Program
{
   private static void Main(string[] args)
   {
      ClassA classA = new ClassA();
      Console.ReadKey();
   }
}
```

We again see the error that we encountered earlier. Again it reminds us that we cannot use new if we have already used an abstract modifier.

5. Abstract class acting as a base class:

Let's add one more class as in the following:

```
/// <summary>
/// Abstract class ClassA
/// </summary>


   public abstract class ClassA
   {
      public int a;
      public void XXX()
   {
  }
}

/// <summary>
/// Derived class.
/// Class derived from abstract class ClassA
/// </summary>

public class ClassB:ClassA
{
}

/// <summary>
/// Program: used to execute the method.
/// Contains Main method.
/// </summary>

public class Program
{
   private static void Main(string[] args)
   {
     ClassB classB = new ClassB();
     Console.ReadKey();
   }
}
```

We get no error. A class can be derived from an abstract class. Creating an object of ClassB does not give us an error.

Point to remember: A class can be derived from an abstract class.

Point to remember: A class derived from an abstract class can create an object.

6. Non abstract method declaration in abstract class:

Another scenario:

```
/// <summary>
/// Abstract class ClassA
/// </summary>
public abstract class ClassA
{
   public int a;
   public void XXX()
   {

   }

   public void YYY();
}

/// <summary>
/// Derived class.
/// Class derived from abstract class ClassA.
/// </summary>

public class ClassB:ClassA
{

}
```

```
/// <summary>
/// Program: used to execute the method.
/// Contains Main method.
/// </summary>

public class Program
{
   private static void Main(string[] args)
   {
      ClassB classB = new ClassB();
      Console.ReadKey();
   }
}
```

We just declared a method named YYY() in our abstract class ClassA. Compile the code; we get:

Output:

Compile time error: 'InheritanceAndPolymorphism.ClassA.YYY()' must declare a body because it is not marked abstract, extern, or partial.

InheritanceAndPolymorphism is the namespace I used for my console application so you can ignore that, no need to confuse it with the logic.

In the code above, we just added a method declaration in the abstract class. An abstract method indicates that the actual definition or code of the method is created somewhere else. The method prototype declared in abstract class must also be declared abstract as per the rules of C#.

7. Abstract method declaration in abstract class:

Just make the method YYY() as abstract in ClassA as in the following:

```
/// <summary>
/// Abstract class ClassA
/// </summary>
public abstract class ClassA
{
   public int a;
   public void XXX()
   {
   }
```

```
    abstract public void YYY();
}
/// <summary>
/// Derived class.
/// Class derived from abstract class ClassA.
/// </summary>
public class ClassB:ClassA
{
}
/// <summary>
/// Program: used to execute the method.
/// Contains Main method.
/// </summary>
public class Program
{
    private static void Main(string[] args)
    {
       ClassB classB = new ClassB();
       Console.ReadKey();
    }
}
```

Output:

Compiler error: 'InheritanceAndPolymorphism.ClassB' does not implement inherited abstract member 'InheritanceAndPolymorphism.ClassA.YYY()'.

Point to remember: If we declare any method as abstract in our abstract class, then it's the responsibility of the derived class to provide the body of that abstract method, unless a body is provided for that abstract method, we cannot create an object of that derived class.

In the previously specified scenario, we declared method YYY() as abstract in ClassA. Since ClassB derives from ClassA, now it becomes the responsibility of ClassB to provide the body of that abstract method, else we cannot create an object of ClassB.

8. Abstract method implementation in derived class:

Now provide a body of method YYY() in ClassB, let's see what happens,

```
/// <summary>
/// Abstract class ClassA
/// </summary>
    public abstract class ClassA
    {
        public int a;
        public void XXX()
    {
}

abstract public void YYY();
}

/// <summary>
/// Derived class.
/// Class derived from abstract class ClassA.
/// </summary>
public class ClassB:ClassA
{
    public void YYY()
    {
    }
}

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

Every thing seems fine now, but now compile the code.

Output:
There are two compile-time errors this time.

Compile time error: 'InheritanceAndPolymorphism.ClassB' does not implement inherited abstract member 'InheritanceAndPolymorphism.ClassA.YYY()'

Compile time warning: 'InheritanceAndPolymorphism.ClassB.YYY()' hides inherited member 'InheritanceAndPolymorphism.ClassA.YYY()'. To make the current member override that implementation, add the override keyword. Otherwise add the new keyword.

We have been continuously trying to compile our code, but with no success until now. The compiler error indicates clearly that both of our base and derived classes contain the same method named YYY().

If both our derived class and base class contains the method with the same name, an error will always occur. The only way to overcome this error is to derive a class explicitly and add the modifier override to its method signature. We have already discussed such scenarios in our previous parts of the articles of Diving Into OOP series.
Let's add the override keyword before the derived class method YYY().

```
/// <summary>
/// Abstract class ClassA
/// </summary>
public abstract class ClassA
{
   public int a;
   public void XXX()
   {
   }
   abstract public void YYY();
}

/// <summary>
/// Derived class.
/// Class derived from abstract class ClassA.
/// </summary>
public class ClassB:ClassA
{
   public override void YYY()
   {
   }
}

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

We get no warning or error now.

9. Abstract method implementation in derived class with different return type:

Let's just change the return type of the method YYY() in the derived class as in the following:

```
/// <summary>
/// Abstract class ClassA
/// </summary>
public abstract class ClassA
{
    public int a;
    public void XXX()
    {
    }
    abstract public void YYY();
}

/// <summary>
/// Derived class.
/// Class derived from abstract class ClassA.
/// </summary>
public class ClassB:ClassA
{
    public override int YYY()
    {
    }
}

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

We changed the return type of method YYY from void to int in the derived class. Compile the code.

Output:

Compile time error: 'InheritanceAndPolymorphism.ClassB.YYY()': return type must be 'void' to match overridden member 'InheritanceAndPolymorphism.ClassA.YYY()'

Therefore there is one more constraint.

Point to remember: When we override an abstract method from a derived class, we cannot change the parameters ed to it or the return type irrespective of the number of methods declared as abstract in the abstract class.

Let's see the implementation of the second line mentioned in “**points to remember**”:

```
/// <summary>
/// Abstract class ClassA
/// </summary>
public abstract class ClassA
{
   public int a;
   public void XXX()
   {
   }
   abstract public void YYY();
   abstract public void YYY1();
   abstract public void YYY2();
   abstract public void YYY3();
}

/// <summary>
/// Derived class.
/// Class derived from abstract class ClassA.
/// </summary>
public class ClassB:ClassA
{
   public override void YYY()
   {
   }
}

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

Compiler errors:

1. 'InheritanceAndPolymorphism.ClassB' does not implement inherited abstract member 'InheritanceAndPolymorphism.ClassA.YYY3()'
2. 'InheritanceAndPolymorphism.ClassB' does not implement inherited abstract member 'InheritanceAndPolymorphism.ClassA.YYY2()'
3. 'InheritanceAndPolymorphism.ClassB' does not implement inherited abstract member 'InheritanceAndPolymorphism.ClassA.YYY1()'

If we implement these three methods in a derived class then we'll get no error.

Point to remember: An abstract class means that the class is incomplete and cannot be directly used. An abstract class can only be used as a base class for other classes to derive from.

10. Variable initialization in abstract class:

Therefore as seen earlier, we get an error if we use a new keyword on an abstract class. If we do not Initialize a variable in an abstract class like we used a, it will automatically have a default value of 0; that is what the compiler kept warning us about. We can initialize an int variable a of ClassA to any value we wish. The variables in an abstract class act similar to that in any other normal class.

11. Power of abstract class:

Whenever a class remains incomplete, in other words we do not have the code for some methods, we mark those methods abstract and the class is marked abstract as well. And so we can compile our class without any error or blocker. Any other class can then derive from our abstract class but they need to implement the abstract, in other words our incomplete methods from the abstract class.

Abstract therefore enables us to write code for a part of the class and allows the others (derived classes) to complete the rest of the code.

12. Abstract method in non abstract class:

Let's take another code block.

```
/// <summary>
/// Abstract class ClassA
/// </summary>
public class ClassA
{
   public int a;
   public void XXX()
   {
   }
   abstract public void YYY();
}

/// <summary>
/// Derived class.
/// Class derived from abstract class ClassA.
/// </summary>
public class ClassB:ClassA
{
   public override void YYY()
   {
   }
}

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

Compile the code.

Output:

Compiler error: 'InheritanceAndPolymorphism.ClassA.YYY()' is abstract but it is contained in non-abstract class 'InheritanceAndPolymorphism.ClassA'

We just removed the abstract keyword from class ClassA. The error clearly conveys a message that if a single method is marked abstract in a class then the class will need to be abstract as well.

Point to remember: If a class has even a single abstract method then the class must be declared abstract as well.

Point to remember: An abstract method also cannot use the modifiers such as static or virtual.

We can only have the abstract method in an abstract class. Any class that derives from an abstract class must provide an implementation to its abstract method. By default the modifier new is added to th derived class method, that makes it a new/different method.

13. Abstract base method:

```
/// <summary>
/// Abstract class ClassA
/// </summary>
public abstract class ClassA
{
   public int a;
   public void XXX()
   {
   }
   abstract public void YYY();
}

/// <summary>
/// Derived class.
/// Class derived from abstract class ClassA.
/// </summary>
public class ClassB:ClassA
{
   public override void YYY()
   {
      base.YYY();
   }
}

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

Output:

Compile time error: Cannot call an abstract base member:

'InheritanceAndPolymorphism.ClassA.YYY()'

We cannot call the method YYY() from the base class ClassA since it does not carry any implementation/code along with it and has also been declared abstract. Common sense prevails and C# of course does not allow us to call a method that does not contain code.

14. Abstract class acting as derived as well as base class:

Let's modify our code a bit, and prepare our class structure something as follows:

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

/// <summary>
/// Derived abstract class.
/// Class derived from base class ClassA.
/// </summary>
public abstract class ClassB:ClassA
{
    public new abstract void XXX();
}
public class ClassC:ClassB
{
    public override void XXX()
    {
       System.Console.WriteLine("ClassC XXX");
    }
}

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

Compile the code, and run.

Output:

```
ClassA XXX
ClassC XXX
```

We created a base class named ClassA that is not abstract and added a virtual method XXX to it. Since the method is non-abstract but marked virtual, it must be overridden in its deriving class. We added one more class named ClassB and marked that class abstract, note that this class is derived from ClassA. So this class has a choice to override the method marked as virtual in the base class.But we'll do something different and tricky,

We marked the XXX method in this derived class as new abstract and did not provide any body to this method. Now what? We will add one more class ClassC that will derive from ClassB. Class C has no choice but to override the method XXX. Therefore we override the method XXX in Class C.

In the main method we created two objects of ClassA, classA = new ClassC() and ClassB, classB = new ClassC().

The first object looks like that of ClassC but refers to ClassA and the second one again seems to be like ClassC but refers to ClassB.

In the case of classA.XXX() will definitely first look into the class ClassA. Here it finds the method XXX marked virtual. These kinds of scenarios we have already taken n number of times in our earlier articles where we discussed runtime polymorphism. C# will then crawl over to class ClassB. Here it gets shocked that the method XXX() is abstract, in other words there is no code or implementation for method XXX() and also that it is a method marked as new, thus severing all links with the base class. And so flow halts and all and the method XXX() from ClassA is executed.

In the case of b.XXX()(), since the method is new, the links to the base class becomes broken, we are left with no choice but to invoke the method from ClassC as it says override.

We cannot replace the modifier new with the keyword override for the method XXX() in the abstract class ClassB.

Let's replace the override modifier in ClassC with “new” as in the following:

```
public class ClassC:ClassB
{
   public new void XXX()
   {
     System.Console.WriteLine("ClassC XXX");
   }
}
```

Output:

Compile time error: 'InheritanceAndPolymorphism.ClassC' does not implement inherited abstract member 'InheritanceAndPolymorphism.ClassB.XXX()'. The error indicates that there is no code for the method XXX. Remember the XXX() of class ClassA has nothing to do at all with that of ClassB and ClassC.

Point to remember: Virtual methods run slower that non virtual methods.

15. Can an abstract class be sealed?

Let's take this final question into consideration, let's test this too with an example as in the following:

```
/// <summary>
/// sealed abstract class ClassA
/// </summary>
public sealed abstract class ClassA
{
   public abstract void XXX()
   {
     Console.WriteLine("ClassA XXX");
```

```
    }
}

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

Compile the code.

Output:

Compile time error: 'InheritanceAndPolymorphism.ClassA': an abstract class cannot be sealed or static

And so we get two points to remember.

Point to remember: Abstract class cannot be sealed class.

Point to remember: Abstract class cannot be a static class.

16. Points to remember

Let's sum up all the points to remember.

1. We cannot create an object of an abstract class using the new keyword.
2. A class can be derived from an abstract class.
3. A class derived from an abstract class can create an object.
4. If we declare any method as abstract in our abstract class, then it's the responsibility of the derived class to provide the body of that abstract method, unless a body is provided for that abstract method, we cannot create an object of that derived class.
5. When we override an abstract method from a derived class, we cannot change the parameters ed to it or the return type irrespective of the number of methods declared as abstract in an abstract class.
6. An abstract class means that the class is incomplete and cannot be directly used. An abstract class can only be used as a base class for other classes to derive from.
7. If a class has even a single abstract method, then the class must be declared abstract as well.
8. An abstract method also cannot use the modifiers such as static or virtual.
9. Virtual methods run slower that non-virtual methods.
10. An abstract class cannot be a sealed class.
11. An abstract class cannot be a static class.

18. Conclusion:

With this we have completed our understanding of inheritance and polymorphism. We have covered nearly all the aspects of polymorphism and inheritance.

Diving Into OOP: All About Access Modifiers in C# (C# Modifiers/Sealed/Constants/Readonly Fields)

Introduction:

We have already covered nearly all the aspects of Inheritance and Polymorphism in C#. I will highlight nearly all the aspects/scenarios of access modifiers in C#. We'll learn by doing a hands-on lab, not just by theory. We'll cover my favourite topic Constants in a very different manner by categorizing the sections in the form of "Labs".

"Access modifiers (or access specifiers)
are keywords in object-oriented languages that set the
accessibility of classes, methods, and other members.
Access modifiers are a specific part of programming language
syntax used to facilitate the encapsulation of components."

Access Modifiers:

Let us take the definition from Wikipedia this time:

"Access modifiers (or access specifiers) are keywords in object-oriented languages that set the accessibility of classes, methods and other members. Access modifiers are a specific part of programming language syntax used to facilitate the encapsulation of components."

Like the definition says, we can control the accessibility of our class methods and members through access modifiers, let us understand this in detail by taking every access modifier one by one.

Public, Private, Protected at class level:

Whenever we create a class we always want to have the scope to decide who can access certain members of the class. In other words, we would sometimes need to restrict access to the class members. The one thumb rule is that members of a class can freely access each other. A method in one class can always access another method of the same class without any restrictions. When we talk about the default behavior, the same class is allowed complete access but no else is provided access to the members of the class. The default access modifier is private for class members.

Point to remember: The default access modifier is private for class members.

Let's do a hands-on lab. Just open your Visual Studio and add a console application in C# named AccessModifiers, you'll get a Program class file by default. In the same file add a new class named Modifiers and add the following code to it:

```
using System;

namespace AccessModifiers
{
    class Modifiers
    {
        static void AAA()
        {
            Console.WriteLine("Modifiers AAA");
        }

        public static void BBB()
        {
            Console.WriteLine("Modifiers BBB");
            AAA();
        }
    }





class Program
    {
        static void Main(string[] args)
        {
            Modifiers.BBB();
        }
    }

}
```

So, your Program.cs file becomes like as shown in the code snippet above. We added a class Modifiers and two static methods AAA and BBB. Method BBB is marked as public. We call the method BBB from the Main method.The method is called directly by the class name because it is marked static.

When we run the application, we get the output as follows:

Output:

```
Modifiers BBB
Modifiers AAA
```

BBB is marked public and so anyone is allowed to call and run it. Method AAA is not marked with any access modifier that automatically makes it private, that is the default. The private modifier has no effect on members of the same class and so method BBB is allowed to call method AAA. Now this concept is called member access.

Modify the Program class and try to access AAA as:

```
class Program
{
   static void Main(string[] args)
   {
      Modifiers.AAA();
      Console.ReadKey();
   }
}
```

Output:

'AccessModifiers.Modifiers.AAA()' is inaccessible due to its protection level So, since the method AAA is private therefore no one else can have access to it except Modifiers class.

Now mark the AAA method as protected, our class looks as in the following.

Modifiers

```
class Modifiers
{
   protected static void AAA()
   {
       Console.WriteLine("Modifiers AAA");
   }

   public static void BBB()
   {
       Console.WriteLine("Modifiers BBB");
       AAA();
   }
}
```

Program

```
class Program
{
    static void Main(string[] args)
    {
        Modifiers.AAA();
        Console.ReadKey();
    }
}
```

Output:

'AccessModifiers.Modifiers.AAA()' is inaccessible due to its protection level Again the same output. We cannot access the method AAA even after we introduced a new modifier named protected. But BBB can access the AAA method because it lies in the same class.

Modifiers in Inheritance:

Let's add one more class and make a relation of base and derived class to our existing class and add one more method to our base class. So our class structure will look something like the following.

Modifiers Base Class

```
class ModifiersBase
{
    static void AAA()
    {
       Console.WriteLine("ModifiersBase AAA");
    }
    public static void BBB()
    {
      Console.WriteLine("ModifiersBase BBB");
    }
    protected static void CCC()
    {
       Console.WriteLine("ModifiersBase CCC");
    }
}
```

Modifiers Derive Class

```
class ModifiersDerived:ModifiersBase
{
   public static void XXX()
   {
       AAA();
       BBB();
       CCC();
   }
}
```

Program Class

```
class Program
{
    static void Main(string[] args)
     {
        ModifiersDerived.XXX();
        Console.ReadKey();
      }
}
```

Output:

'AccessModifiers.ModifiersBase.AAA()' is inaccessible due to its protection level.

Now in this case we are dealing with a derived class. Whenever we mark a method with the specifier, protected, we are actually telling C# that only derived classes can access that method and no one else can. Therefore in the method XXX we can call CCC because it is marked protected, but it cannot be called from anywhere else including the Main function. The method AAA is made private and can be called only from the class ModifiersBase. If we remove AAA from method XXX, the compiler will give no error.

Therefore now we are aware of three important concepts. Private means only the same class has access to the members, public means everybody has access and protected lies in between where only derived classes have access to the base class method.

All the methods for example reside in a class. The accessibility of that method is decided by the class in which it resides as well as the modifiers on the method. If we are allowed an access to a member, then we say that the member is accessible, else it is inaccessible.

Internal modifier at class level:

Let's take one other scenario. Create a class library with a name “AccessModifiersLibrary” in your Visual Studio. Add a class named ClassA in that class library and mark the class as internal, the code will be as shown below.

AccessModifiersLibrary.ClassA

```
namespace AccessModifiersLibrary
{
   internal class ClassA
   {
   }
}
```

Now compile the class and leave it. Its DLL will be generated in the ~\AccessModifiersLibrary\bin\Debug folder.

Now in your console application “AccessModifiers” was created earlier. Add the reference of the AccessModifiersLibrary library by adding its compiled DLL as a reference to AccessModifiers.

In Program.cs of the AccessModifiers console application, modify the Program class like shown below.

AccessModifiers.Program

```
using AccessModifiersLibrary;

namespace AccessModifiers
{
    class Program
    {
        static void Main(string[] args)
        {
            ClassA classA;
        }
    }

}
```

And compile the code.

Output:

Compile time error: 'AccessModifiersLibrary.ClassA' is inaccessible due to its protection level.

We encountered this error because the access specifier internal means that we can only access ClassA from AccessModifiersLibrary.dll and not from any other file or code. An internal modifier means that access is limited to the current program only. So try never to create a component and mark the class internal as no one would be able to use it.

And what if we remove the field internal from ClassA, will the code compile? As in the following.

AccessModifiersLibrary.ClassA

```
namespace AccessModifiersLibrary
{
    class ClassA
    {
    }
}
```

AccessModifiers.Program

```
using AccessModifiersLibrary;

namespace AccessModifiers
{
    class Program
    {
        static void Main(string[] args)
        {
            ClassA classA;
        }
    }
}
```

Output:

Compile time error: 'AccessModifiersLibrary.ClassA' is inaccessible due to its protection level.

We again got the same error. We should not forget that by default if no modifier is specified, the class is internal. So our class ClassA is internal by default even if we do not mark it with any access modifier, so the compiler result remains the same.

Had the class ClassA been marked public, everything would have gone smoothly without any error.

Point to remember: A class marked as internal can only have its access limited to the current assembly only.

Namespaces with modifiers:

Just for fun, let's mark the namespace of the AccessModifiers class library as public in the Program class.

Program:

```
public namespace AccessModifiers
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Compile the application.

Output:

Compile time error: A namespace declaration cannot have modifiers or attributes.

Point to remember: Namespaces as we see by default can have no accessibility specifiers at all. They are by default public and we cannot add any other access modifier including public again too.

Private Class:

Let's do one more experiment and mark the class Program as private, so our code becomes:

```
namespace AccessModifiers
{
    private class Program
    {
        static void Main(string[] args)
        {

        }
    }
}
```

Compile the code.

Output:

Compile time error: Elements defined in a namespace cannot be explicitly declared as private, protected, or protected internal.

So, **point to remember:** A class can only be public or internal. It cannot be marked as protected or private. The default is internal for the class.

Access modifiers for the members of the class:

Now here is a big statement, that the members of a class can have all the preceding explained access modifiers, but the default modifier is private.

Point to remember: Members of a class can be marked with all the access modifiers and the default access modifier is private.

What if we want to mark a method with two access modifiers?

```
namespace AccessModifiers
{
    public class Program
    {
        static void Main(string[] args)
        {
        }

        public private void Method1()
        {

        }
    }
}
```

Compile the code.

Output:

Compile time error: More than one protection modifier.

Therefore we can't mark a member with more than one access modifier often. But there are such scenarios too, we'll cover them in the next sections. Already defined types like int and object have no accessibility restrictions. They can be used anywhere and everywhere.

Internal class and public method:

Create a class library with a class named ClassA marked internal and have a public method MethodClassA(), as in the following:

```
namespace AccessModifiersLibrary
{
   internal class ClassA
   {
      public void MethodClassA(){}
   }
}
```

Add the reference of class library to our console application. Now in Program.cs of the console application, try to access that method MethodClassA of ClassA.

Program

```
using AccessModifiersLibrary;

 namespace AccessModifiers
{
   public class Program
   {
      public static void Main(string[] args)
      {
         ClassA classA = new ClassA();
         classA.MethodClassA();
      }

   }
}
```

Output:

Compile time errors:

'AccessModifiersLibrary.ClassA' is inaccessible due to its protection level The type 'AccessModifiersLibrary.ClassA' has no constructors defined

'AccessModifiersLibrary.ClassA' is inaccessible due to its protection level
'AccessModifiersLibrary.ClassA' does not contain a definition for 'MethodClassA' and no extension method 'MethodClassA' accepting a first argument of type 'AccessModifiersLibrary.ClassA' could be found (are you missing a using directive or an assembly reference?)

So many errors. The errors are self-explanatory though. Even the method MethodClassA of ClassA is public; it could not be accessed in the Program class due to the protection level of ClassA, in other words internal. The type enclosing the method MethodClassA is internal, so no matter if the method is marked public, we cannot access it in any other assembly.

Public class and private method:

Let's make the class ClassA as public and method as private.

AccessModifiersLibrary.ClassA

```
namespace AccessModifiersLibrary
{
   public class ClassA
   {
      private void MethodClassA(){}
   }
}
```

Program:

```
using AccessModifiersLibrary;

 namespace AccessModifiers
{
   public class Program
   {
      public static void Main(string[] args)
      {
         ClassA classA = new ClassA();
         classA.MethodClassA();
      }
```

```
    }
}
```

Output on compilation:

'AccessModifiersLibrary.ClassA' does not contain a definition for 'MethodClassA' and no extension method 'MethodClassA' accepting a first argument of type 'AccessModifiersLibrary.ClassA' could be found (are you missing a using directive or an assembly reference?)

Now that we have marked our class Public, we still can't access the private method. So for accessing a member of the class, the access modifier of the class as well as the method is very important.

Public class and internal method:

Make ClassA as public and MethodClassA as internal.

AccessModifiersLibrary.ClassA

```
namespace AccessModifiersLibrary
{
    public class ClassA
    {
        Internal void MethodClassA(){}
    }
}
```

Program:

```
using AccessModifiersLibrary;

 namespace AccessModifiers
{
    public class Program
    {
        public static void Main(string[] args)
        {
            ClassA classA = new ClassA();
            classA.MethodClassA();
        }

    }
}
```

Output on compilation:

'AccessModifiersLibrary.ClassA' does not contain a definition for 'MethodClassA' and no extension method 'MethodClassA' accepting a first argument of type 'AccessModifiersLibrary.ClassA' could be found (are you missing a using directive or an assembly reference?)

So an internal marked member means that no one from outside that DLL can access the member.

Protected internal:

In the class library make three classes ClassA, ClassB and ClassC and place the code somewhat like this.

```
namespace AccessModifiersLibrary
{
    public class ClassA
    {
        protected internal void MethodClassA()
        {

        }
    }

    public class ClassB:ClassA
    {
        protected internal void MethodClassB()
        {
            MethodClassA();
        }
    }

    public class ClassC
    {
        public void MethodClassC()
        {
            ClassA classA=new ClassA();
            classA.MethodClassA();
        }
    }
}
```

And in Program class in our console application, call the MethodClassC of ClassC.

Program:

```
using AccessModifiersLibrary;

 namespace AccessModifiers
{
    public class Program
    {
        public static void Main(string[] args)
        {
            ClassC classC=new ClassC();
            classC.MethodClassC();
        }
    }
}
```

Compiler output: The code successfully compiles with no error.

Protected internal modifier indicates two things, that either the derived class or the class in the same file can have access to that method, therefore in the above mentioned scenario, the derived class

ClassB and the class in the same file, in other words ClassC, can access that method of ClassA marked as protected internal.

Point to remember: Protected internal means that the derived class and the class within the same source code file can have access.

Protected member:

In our Program.cs in the console application, place the following code:

```
namespace AccessModifiers
{
    class AAA
    {
        protected int a;
        void MethodAAA(AAA aaa,BBB bbb)
        {
            aaa.a = 100;
            bbb.a = 200;
        }
    }
     class BBB:AAA
     {
         void MethodBBB(AAA aaa, BBB bbb)
         {
             aaa.a = 100;
             bbb.a = 200;
         }
     }
    public class Program
    {
        public static void Main(string[] args)
        {
        }
    }
}
```

Compiler Output:

Cannot access protected member 'AccessModifiers.AAA.a' via a qualifier of type 'AccessModifiers.AAA'; the qualifier must be of type 'AccessModifiers.BBB' (or derived from it)

Class AAA contains a protected member, in other words a. But to the same class no modifiers make sense. However as a is protected, in the derived class method MethodBBB, we cannot access it through AAA since aaa.a gives us an error. However bbb which looks like BBB does not give an error. To check this out, comment out the line aaa.a=100 in MethodBBB (). This means that we cannot access the protected members from an object of the base class, but from the objects of a derived class only. This is in spite of the fact that a is a member of AAA, in other words the base class. Even so, we still cannot access it. We also cannot access a from the method Main.

Accessibility Priority in inheritance:

Program:

```
namespace AccessModifiers

{
    class AAA
    {

    }
    public class BBB:AAA
     {

     }
    public class Program
    {
        public static void Main(string[] args)
        {
        }
    }
}
```

Compiler Output:

Compile time error: Inconsistent accessibility: base class 'AccessModifiers.AAA' is less accessible than class 'AccessModifiers.BBB'

The error again gives us one more point to remember.

Point to remember: between public and internal, public always allows greater access to its members.

The class AAA is by default marked internal and BBB that derives from AAA is made public explicitly. We got an error since the derived class BBB must have an access modifier that allows greater access than the base class access modifier. Here internal seems to be more restrictive than public.

But if we reverse the modifiers to both the classes in other words ClassA marked as public and ClassB internal or default, we eliminate the error.

Point to remember: The base class always allows more accessibility than the derived class.

Another scenario.

Program:

```
namespace AccessModifiers

  class AAA
  {

  }
  public class BBB
   {
     public AAA MethodB()
     {
        AAA aaa= new AAA();
        return aaa;
     }
   }
  public class Program
  {
     public static void Main(string[] args)
     {
     }
  }
```

Compiler output: Inconsistent accessibility: return type 'AccessModifiers.AAA' is less accessible than method 'AccessModifiers.BBB.MethodB()'.

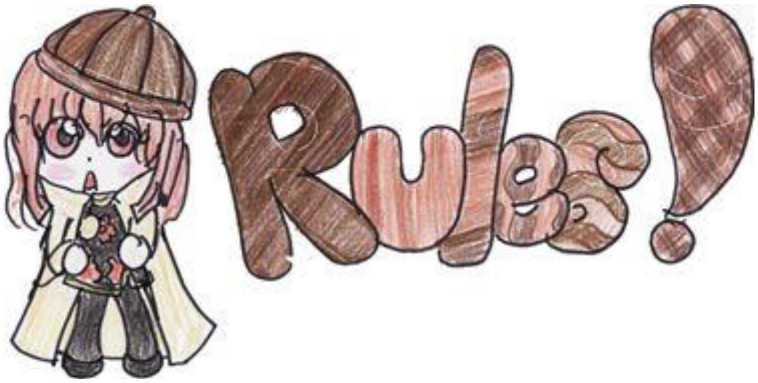
Here the accessibility of AAA is internal that is more restrictive than public. The accessibility of method MethodB is public that is more than that of the typeAAA. Now the error occurred because return values of a method must have greater accessibility than that of the method itself, which is not true in this case.

Point to remember: The return values of a method must have greater accessibility than that of the method itself.

Program

```
namespace AccessModifiers
{
    class AAA
    {

    }
    public class BBB
    {
        public AAA aaa;
    }
    public class Program
    {
        public static void Main(string[] args)
        {

        }
    }
}
```

Compiler Output: Inconsistent accessibility: field type 'AccessModifiers.AAA' is less accessible than field 'AccessModifiers.BBB.aaa'.



Now the rules are the same for everyone. The class AAA or data type aaa is internal. The aaa field is public and that makes it more accessible than AAA that is internal. So we got the error.

Change the code to:

```
namespace AccessModifiers
{
    class AAA
    {

    }
    public class BBB
    {
         AAA a;
    }
    public class Program
    {
        public static void Main(string[] args)
        {
        }
    }
}
```

The output compilation results in no error.

We learned a lot about these access modifiers like, public, private, protected, internal and protected internal. We also learned about their priority of access and usage, let's summarize their details in a tabular format for revision. Later we'll move to other types as well.

Tables taken from: msdn

Declared accessibility	**Meaning**
public	Access is not restricted.
protected	Access is limited to the containing class or types derived from the containing class.
internal	Access is limited to the current assembly.
protected internal	Access is limited to the current assembly or types derived from the containing class.
private	Access is limited to the containing type.

“Only one access modifier is allowed for a member or type, except when you use the protected internal combination.

Access modifiers are not allowed on namespaces. Namespaces have no access restrictions.

Depending on the context in which a member declaration occurs, only certain declared accessibilities are permitted. If no access modifier is specified in a member declaration, a default accessibility is used.

Top-level types, that are not nested in other types, can only have internal or public accessibility. The default accessibility for these types is internal. Nested types, that are members of other types, can have declared accessibilities as indicated in the following table.”

Members of	**Default member accessibility**	**Allowed declared accessibility of the member**
enum	Public	None
class	Private	public protected internal private protected internal
interface	Public	None
struct	Private	public internal private

Sealed Classes

“Sealed” is a special class of access modifier in C#. If a class is marked as sealed, no other class can derive from that sealed class. In other words a class marked as sealed can't act as a base class to any other class.

Program:

```
namespace AccessModifiers
{
   sealed class AAA
   {

   }
   class BBB:AAA
   {

   }
   public class Program
   {
      public static void Main(string[] args)
      {

      }
   }
}
```

Compiler Output: 'AccessModifiers.BBB': cannot derive from sealed type 'AccessModifiers.AAA'

Hence proved.

Point to remember: A class marked sealed can't act as a base class for any other class.

Access the members of the sealed class.

Program:

```
using System;
namespace AccessModifiers
{
    sealed class AAA
    {
        public int x = 100;
        public void MethodA()
        {
            Console.WriteLine("Method A in sealed class");
        }

    }
    public class Program
    {
        public static void Main(string[] args)
        {
            AAA aaa=new AAA();
            Console.WriteLine(aaa.x);
            aaa.MethodA();
            Console.ReadKey();
        }
    }
}
```

Compiler **Output:**

```
100
```

Method A in sealed class

So, as we discussed, the only difference between a sealed and a non-sealed class is that the sealed class cannot be derived from. A sealed class can contain variables, methods and properties like a normal class does.

Point to remember: Since we cannot derive from sealed classes, the code from the sealed classes cannot be overridden.

Constants:

Lab 1
Our Program class in the console application is as follows.

Program:

```
public class Program
 {
    private const int x = 100;
    public static void Main(string[] args)
    {
       Console.WriteLine(x);
       Console.ReadKey();
    }
 }
```

Output: 100

We see, a constant maked variable or a const variable behaves like a member variable in C#. We can provide it an initial value and can use it anywhere we want.

Point to remember: We need to initialize the const variable at the time we create it. We are not allowed to initialize it later in our code or program.

Lab 2

```
 using System;
 namespace AccessModifiers
{
   public class Program
   {
       private const int x = y + 100;
       private const int y = z - 10;
       private const int z = 300;

       public static void Main(string[] args)
       {
          System.Console.WriteLine("{0} {1} {2}",x,y,z);
           Console.ReadKey();
       }
   }
}
```

Can you guess the output? What ? Is it a compiler error?

Output:

```
390 290 300
```

Shocked? A constant field can no doubt depend upon another constant. C# is very smart to realize that to calculate the value of variable x marked const, it first needs to know the value of y variable. y's value depends upon another const variable z, whose value is set to 300. Thus C# first evaluates z to 300 then y becomes 290, in other words z -1 and finally x takes on the value of y, in other words 290 + 100 resulting in 390.

Lab 3
Program:

```
using System;
```

```
namespace AccessModifiers
{
   public class Program
   {
      private const int x = y + 100;
      private const int y = z - 10;
      private const int z = x;

      public static void Main(string[] args)
      {
        System.Console.WriteLine("{0} {1} {2}",x,y,z);
         Console.ReadKey();
      }
   }
}
```

Output: The evaluation of the constant value for 'AccessModifiers.Program.x' involves a circular definition.

We just assigned z=x from our previous code and it resulted in an error. The value of const x depends upon y and y in turn depends upon the value of z, but we see the value z depends upon x since x is assigned directly to z, it results in a circular dependency.

Point to remember: Like classes const variables cannot be circular, in other words they cannot depend on each other.

Lab 4

A const is a variable whose value once assigned cannot be modified, but its value is determined at compile time only.

```
using System;
namespace AccessModifiers
{
   public class Program
   {
      public const ClassA classA=new ClassA();
      public static void Main(string[] args)
      {
      }
   }

  public class ClassA
   {
    }
}
```

Output:

Compile time error: 'AccessModifiers.Program.classA' is of type 'AccessModifiers.ClassA'. A const field of a reference type other than string can only be initialized with null.

Point to remember: A const field of a reference type other than string can only be initialized with null.

If we assign the value to null in the Program class as in the following:

```
using System;
namespace AccessModifiers
{
    public class Program
    {
        public const ClassA classA=null;
        public static void Main(string[] args)
        {
        }
   }

   public class ClassA
    {

    }
}
```

Then the error will vanish. The error disappears since we now initialize classA to an object that has a value that can be determined at compile time, null. We can never change the value of classA, so it will always be null. Normally we do not have consts as a classA reference type since they have a value only at runtime.

Point to remember: One can only initialize a const variable to a compile-time value, in other words a value available to the compiler while it is executing.

new() actually gets executed at runtime and therefore does not get a value at compile time. So this results in an error.

Lab 5

Class A

```
public class ClassA
{
    public const int aaa = 10;
}
```

Program:

```
public class Program
{
   public static void Main(string[] args)
   {
      ClassA classA=new ClassA();
      Console.WriteLine(classA.aaa);
      Console.ReadKey();
   }
}
```

Output:

Compile time error: Member 'AccessModifiers.ClassA.aaa' cannot be accessed with an instance reference; qualify it with a type name instead.

Point to remember: A constant by default is static and we can't use the instance reference, in other words a name to reference a const. A const must be static since no one will be allowed to make any changes to a const variable.

Just mark the const as static.

```
using System;
namespace AccessModifiers
{
   public class ClassA
   {
      public static const int aaa = 10;
   }
```

```
    public class Program
    {
        public static void Main(string[] args)
        {
            ClassA classA=new ClassA();
            Console.WriteLine(classA.aaa);
            Console.ReadKey();
        }
    }

}
```

Output:

Compile time error: The constant 'AccessModifiers.ClassA.aaa' cannot be marked static. C# tells us frankly that a field already static by default cannot be marked as static.

Point to remember: A const variable cannot be marked as static.

Lab 6

```
using System;
namespace AccessModifiers
{
    public class ClassA
    {
        public const int xxx = 10;
    }



 public class ClassB:ClassA
    {
        public const int xxx = 100;
    }

    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine(ClassA.xxx);
            Console.WriteLine(ClassB.xxx);
            Console.ReadKey();
        }
    }
}
```

Output:

```
10
100
```

Compiler Warning: 'AccessModifiers.ClassB.xxx' hides inherited member 'AccessModifiers.ClassA.xxx'. Use the new keyword if hiding was intended.

We can always create a const with the same name in the derived class as another const in the base class. The const variable of class ClassB xxx will hide the const xxx in class ClassA for the class ClassB only.

Static Fields

Point to remember: A variable in C# can never have an uninitialized value.
Let's discuss this in detail.

Lab 1
Program:

```
using System;
namespace AccessModifiers
{
    public class Program
    {
        private static int x;
        private static Boolean y;
        public static void Main(string[] args)
        {
            Console.WriteLine(x);
            Console.WriteLine(y);
            Console.ReadKey();
        }
    }
}
```

Output:

```
0
False
```

Point to remember: Static variables are always initialized when the class is loaded first. An int is given a default value of zero and a bool is given a default to False.

Lab 2

Program:

```
using System;
namespace AccessModifiers
{
   public class Program
   {
      private  int x;
      private  Boolean y;
      public static void Main(string[] args)
      {
         Program program=new Program();
         Console.WriteLine(program.x);
         Console.WriteLine(program.y);
         Console.ReadKey();
      }
   }

}
```

Output:

```
0
False
```

Point to remember: An instance variable is always initialized at the time of creation of its instance.

An instance variable is always initialized at the time of creation of its instance. The keyword new will create an instance of the class Program. It will allocate memory for each of the non-static, in other words instance variables, and then initialize each of them to their default values as well.

Lab 3

Program:

```
using System;
namespace AccessModifiers
{
   public class Program
   {
      private static int x = y + 10;
      private static int y = x + 5;
      public static void Main(string[] args)
      {
         Console.WriteLine(Program.x);
         Console.WriteLine(Program.y);
         Console.ReadKey();
      }
   }
}
```

Output:

```
10
15
```

The output is self explanatory. C# always initializes static variables to their initial value after creating them. Variables x and y are therefore given a default of zero value. C# now realizes that these variables declared need to be assigned some values. C# does not read all the lines at once but only one at a time. It will now read the first line and since the variable y has a value of 0, x will get a value of 10. Then at the next line, y is the value of x + 5. The variable x has a value of 10 and so y now becomes 15. Since C# does not see both lines at the same time, it does not notice the circularity of the preceding definition.

Lab 4

Program:

```
using System;
namespace AccessModifiers
{
   public class Program
```

```
    {
         int x = y + 10;
         int y = x + 5;
        public static void Main(string[] args)
        {

        }
    }
}
```

Output:

Compile time error: A field initializer cannot reference the non-static field, method, or property 'AccessModifiers.Program.y'.

A field initializer cannot reference the non-static field, method, or property 'AccessModifiers.Program.x'

The lab we did in Lab 3 does not work for instance variables since the rules of an instance variable are quite different than that of static variables. The initializer of an instance variable must be determined at the time of creation of the instance. The variable y does not have a value at this point in time. It can't refer to variables of the same object at the time of creation. So we can refer to no instance members to initialize an instance member.

Readonly Fields

Readonly fields are one of the most interesting topics of OOP in C#.

Lab 1
Program:

```
using System;
namespace AccessModifiers
{
    public class Program
    {
        public static readonly int x = 100;

        public static void Main(string[] args)
        {
            Console.WriteLine(x);
            Console.ReadKey();
        }
    }
}
```

Output: 100

Wow, we get no error, but remember not to use a non-static variable inside a static method else we'll get an error.

Lab 2
Program:

```
using System;
namespace AccessModifiers
{
    public class Program
    {
        public static readonly int x = 100;

        public static void Main(string[] args)
        {
            x = 200;
            Console.WriteLine(x);
            Console.ReadKey();
        }
    }
}
```

Output:

Compile time error: A static readonly field cannot be assigned to (except in a static constructor or a variable initializer).
We cannot change the value of a readonly field except in a constructor.

Point to remember: A static readonly field cannot be assigned to (except in a static constructor or a variable initializer)

Lab 3
Program:

```
using System;
namespace AccessModifiers
{
    public class Program
    {
        public static readonly int x;

        public static void Main(string[] args)
        {

        }
    }
}
```

Here we find one difference between const and readonly, unlike const, readonly fields need not need to be initialized at the time of creation.

Lab 4
Program:

```
using System;
namespace AccessModifiers
{
    public class Program
    {
        public static readonly int x;

        static Program()
        {
            x = 100;
            Console.WriteLine("Inside Constructor");
        }

        public static void Main(string[] args)
        {
            Console.WriteLine(x);
            Console.ReadKey();
        }
    }
}
```

Output:

```
Inside Constructor
100
```

One more major difference between const and readonly is seen here. A static readonly variable can be initialized in the constructor as well, like we saw in the above mentioned example.

Lab 5
Program:

```
using System;
namespace AccessModifiers
{
    public class ClassA
    {

    }
    public class Program
    {

        public readonly ClassA classA=new ClassA();
        public static void Main(string[] args)
        {

        }
    }
}
```

We have already seen this example in the const section. The same code gave an error with const does not give an error with readonly fields. So we can say that readonly is a more generic const and it makes our programs more readable as we refer to a name and not a number. Is 10 more intuitive or priceofcookie easier to understand? The compiler would for efficiency convert all const's and readonly fields to the actual values.

Lab 6
Program:

```
using System;
namespace AccessModifiers
{
    public class ClassA
    {
        public int readonly x= 100;
    }
    public class Program
    {
      public static void Main(string[] args)
        {
        }
    }
}
```

Output:

Compile time error: Member modifier 'readonly' must precede the member type and name

Invalid token '=' in class, struct, or interface member declaration

Wherever we need to place multiple modifiers, remind yourself that there are rules that decide the order of access modifiers, that comes first. Now here the readonly modifier precedes the data type int, we already discussed in the very start of the article. This is just a rule that must always be remembered.

Lab 7

Program:

```
using System;
namespace AccessModifiers
{
    public class ClassA
    {
        public readonly int x= 100;

        void Method1(ref int y)

     }
}

       void Method2()
        {
            Method1(ref x);
        }
    }
    public class Program
    {
        public static void Main(string[] args)
        {
        }
    }
}
```

Output:

Compile time error: A readonly field cannot be ed ref or out (except in a constructor).

A readonly field can't be changed by anyone except a constructor. The method Method1 expects a ref parameter that if we have forgotten allows you to change the value of the original. Therefore C# does not permit a readonly as a parameter to a method that accepts a ref or an out parameters.

Summary

Let's recall all the points that we must remember.

1. The default access modifier is private for class members.
2. A class marked as internal can only have its access limited to the current assembly only.
3. Namespaces as we see by default can have no accessibility specifiers at all. They are by default public and we cannot add any other access modifier including public again too.
4. A class can only be public or internal. It cannot be marked as protected or private. The default is internal for the class.
5. Members of a class can be marked with all the access modifiers and the default access modifier is private.
6. Protected internal means that the derived class and the class within the same source code file can have access.
7. Between public and internal, public always allows greater access to its members.
8. Base class always allows more accessibility than the derived class.
9. The return values of a method must have greater accessibility than that of the method itself.
10. A class marked sealed can't act as a base class to any other class.
11. Since we cannot derive from sealed classes, the code from the sealed classes cannot be overridden.

12. We need to initialize the const variable at the time we create it. We are not allowed to initialize it later in our code or program.
13. Like classes const variables cannot be circular, in other words they cannot depend on each other.
14. A const field of a reference type other than string can only be initialized with null.
15. One can only initialize a const variable to a compile-time value, in other words a value available to the compiler while it is executing.
16. A constant by default is static and we can't use the instance reference, in other words a name to reference a const. A const must be static since no one will be allowed to make any changes to a const variable.
17. A const variable cannot be marked as static.
18. A variable in C# can never have an uninitialized value.
19. Static variables are always initialized when the class is loaded first. An int is given a default value of zero and a bool is given a default of false.
20. An instance variable is always initialized at the time of creation of its instance.
21. A static readonly field cannot be assigned to (except in a static constructor or a variable initializer)

Conclusion:

With this we completed nearly all the scenarios of access modifiers. We did many hands-on labs to clarify the concepts.

Diving Into OOP: Understanding Enums in C# (A Practical Approach)

Introduction:

This article of the series “Diving into OOP” will explain the enum datatype in C#. We'll learn by doing a hands-on lab, not just by theory. We'll explore the power of enum and will cover almost every scenario in which we can use an enum. We'll follow a practical approach of learning to understand this concept. We may come across complex examples to understand the concept more deeply.

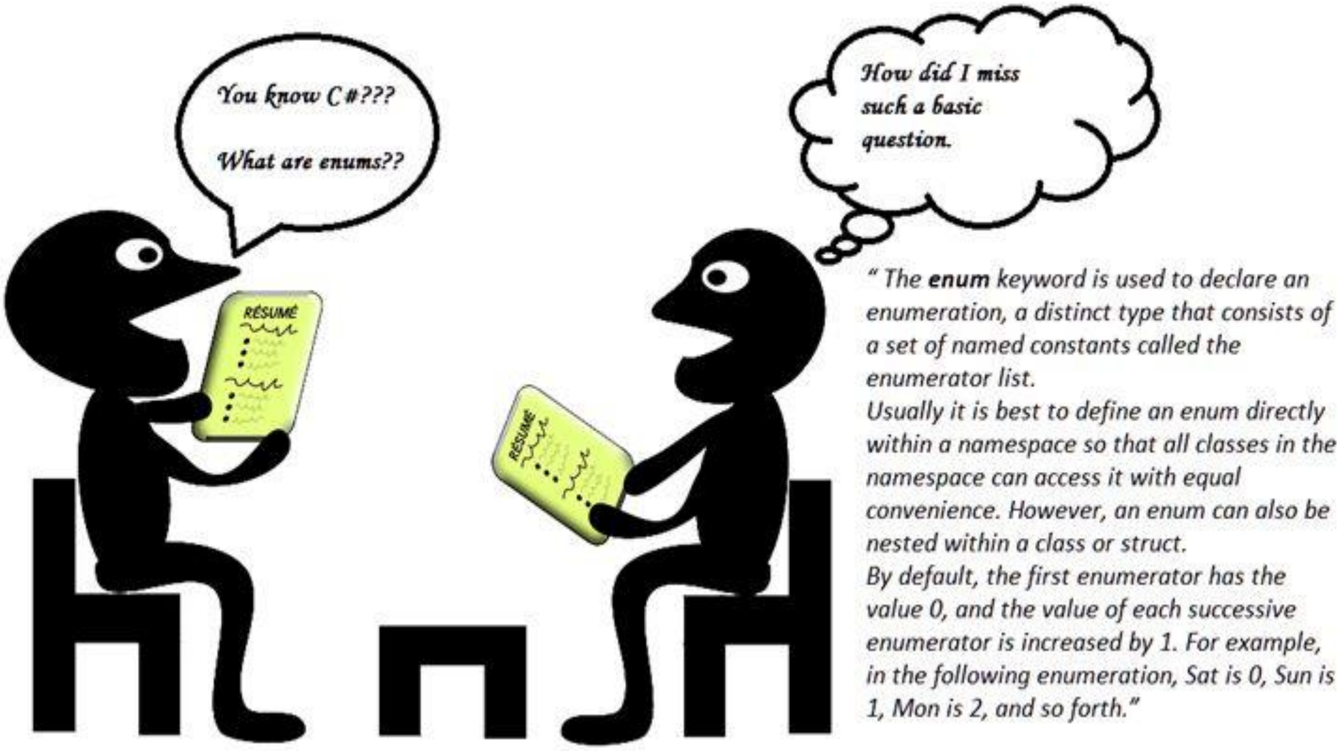
Enums (The definition)

Let's start with the definition taken from the MSDN:

“The enum keyword is used to declare an enumeration, a distinct type that consists of a set of named constants called the enumerator list.

Usually it is best to define an enum directly within a namespace so that all classes in the namespace can access it with equal convenience. However, an enum can also be nested within a class or struct.

By default, the first enumerator has the value 0 and the value of each successive enumerator is increased by 1. For example, in the following enumeration, Sat is 0, Sun is 1, Mon is 2 and so forth."



" The **enum** keyword is used to declare an enumeration, a distinct type that consists of a set of named constants called the enumerator list.
Usually it is best to define an enum directly within a namespace so that all classes in the namespace can access it with equal convenience. However, an enum can also be nested within a class or struct.
By default, the first enumerator has the value 0, and the value of each successive enumerator is increased by 1. For example, in the following enumeration, Sat is 0, Sun is 1, Mon is 2, and so forth."

A Practical Approach

Enum plays almost the same responsibility as the class does, in other words creating a new data type and it exists at the same level as class, interfaces or structs. Just open your Visual Studio and add a console application named Enums. You'll get a Program class.

Note: Each and every code snippet in this article is tried and tested.

Declare an enum at the same level as of the Program class, call it Color.

Program.cs

```
namespace Enums
{
   class Program
```

```
    {
        static void Main(string[] args)
        {
        }
    }

    enum Color
    {
        Yellow,
        Blue,
        Brown,
        Green
    }
}
```

In the preceding example, we created a new datatype using an enum. The datatype is Color having four distinct values Yellow, Blue, Brown and Green. The text that we write inside the declared enum could be anything of your wish; it just provides a custom enumerated list to you.

Modify your main program as shown below.

```
using System;
namespace Enums
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(Color.Yellow);
            Console.ReadLine();
        }
    }
  enum Color
    {
        Yellow,
        Blue,
        Brown,
        Green
    }
}
```

Run the program.

Output: Yellow

Now if we just typecast Color.Yellow to int, what do we get?

```
using System;
namespace Enums
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine((int)Color.Yellow);
         Console.ReadLine();
      }
   }
   enum Color
   {
      Yellow,
      Blue,
      Brown,
      Green
   }
}
```

Output: 0

We see that enums are called static variables, so an enum can be considered here as static objects. Therefore other enums in the preceding example can be declared in the same way as Yellow, like Blue can be declared as Color.Blue. The output in the above two examples we see is 0 when we type cast and Yellow without typecasting, hence we see here that its behaviour is very similar to an array where Yellow has a value 0, similarly Blue has a value 1, Brown: 2, Green:3.

Therefore when we do Color.Yellow, it's like displaying a number 0, so from this we can infer that an enum represents a constant number, therefore an enum type is a distinct type having named constants.

A point to remember: An enum represents a constant number and an enum type is known as a distinct type having named constants.

Underlying data type

Program.cs

```
using System;
namespace Enums
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine((byte)Color.Yellow);
            Console.WriteLine((byte)Color.Blue);
            Console.ReadLine();
        }
    }
    enum Color:byte
    {
        Yellow,
        Blue,
        Brown,
        Green
    }
}
```

Output:

```
0
1
```

Note: Each and every code snippet in this article is tried and tested.

The only change we did here is that we specified the type to the underlying enum that we declared. The default datatype for the enum is int, here we have specified the data type as byte and we get the result.

There are more data types that can be specified for enum like long, ulong, short, ushort, int, uint, byte and sbyte.

Point to remember: We can't declare char as an underlying data type for enum objects because char stores Unicode characters, but enum objects data type can only be a number.

Inheritance in enum

Program.cs

```
using System;
namespace Enums
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine((byte)Color.Yellow);
            Console.WriteLine((byte)Color.Blue);
            Console.ReadLine();
        }
    }



 enum Color:byte
    {
        Yellow,
        Blue,
        Brown,
        Green
    }
    enum Shades:Color
    {

    }
}
```

Output:

Compile time error: Type byte, sbyte, short, ushort, int, uint, long, or ulong expected. We clearly see here enums can't be derived from any other type except that specified in the error.

Point to remember: an enum can't be derived from any other type except that of type byte, sbyte, short, ushort, int, uint, long, or ulong

Let's derive a class from an enum, call it class Derived, so our code is as in the following.

Program.cs:

```
class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine((byte)Color.Yellow);
            Console.WriteLine((byte)Color.Blue);
            Console.ReadLine();
        }
    }
```

Enum:

```
enum Color:byte
{
    Yellow,
    Blue,
    Brown,
    Green
}
```

Derived.cs:

```
class Derived:Color
    {

    }
```

Compile the code.

Output: Compile time error: 'Enums.Derived': cannot derive from sealed type 'Enums.Color'

So, the point to remember is: By default an enum is a sealed class and therefore conforms to all the rules that a sealed class follows, so no class can derive from an enum, in other words a sealed type.

Can System.Enum be a base class to enum?

Program.cs:

```
using System;
namespace Enums
{
   internal enum Color: System.Enum
   {
      Yellow,
      Blue
   }
   internal class Program
   {
      private static void Main(string[] args)
      {

      }
   }
}
```

Output:

Compile time error: Type byte, sbyte, short, ushort, int, uint, long, or ulong expected

Point to remember: The enum type is implicitly derived from System.Enum and so we cannot explicitly derive it from System.Enum.

To add more, an enum is also derived from the three interfaces IComparable, IFormattable and IConvertible.

A. IComparable - Let's check.

Program.cs:

```
using System;
namespace Enums
{
   internal enum Color
```

```
    {
        Yellow,
        Blue,
        Green
    }

    internal class Program
    {
        private static void Main(string[] args)
        {
            Console.WriteLine(Color.Yellow.CompareTo(Color.Blue));
            Console.WriteLine(Color.Blue.CompareTo(Color.Green));
            Console.WriteLine(Color.Blue.CompareTo(Color.Yellow));
            Console.WriteLine(Color.Green.CompareTo(Color.Green));
            Console.ReadLine();
        }
    }
}
```

Output:

```
-1
-1
 1
 0
```

Sometimes we may get into situations where we have a large number of enums defined and we want to compare the values of the enum to each other to check if they are smaller, larger or an equal value to one another.

Since all enums implicitly derive from an Enum class that implements the interface IComparable, so they all have a method CompareTo(), that we just use in the above example. The method being non-static must be used through a member. Yellow has the value 0, Blue has 1 and Green has 2. In the first statement, when Color.Yellow is comared to Color.Blue, the value of Yellow is less than Blue hence -1 is returned. The same applies for the second statement when Color.Blue is compared to Color.Green. Green has a larger value, in other words 2 than that of Color.Blue having value 1 only. In the third statement, in other words the revers of the first statement, we get the result of comparison as 1, because Blue is greater than Yellow. In the last statement where Color.Green compares to itself, we undoubtedly get the value 0.

So the value -1 means the value is smaller, 1 means the value is larger and 0 means equal values for both the enum members.

The following is another comparison example.

Program.cs:

```
using System;
namespace Enums
{
    enum Color
   {
      Yellow,
      Blue,
      Green
   }

   internal class Program
   {
      private static void Main(string[] args)
      {
         int myColor = 2;
         if(myColor== Color.Green)
         {
            Console.WriteLine("my color");
         }
        Console.ReadLine();
      }
   }
}
```

Output: Compile time error : Operator '==' cannot be applied to operands of type 'int' and 'Enums.Color'

In the above example we tried to compare an int type to a Enum type and that resulted in a compile time error. Since an enum acts as an individual data type so it cannot be directly compared to an int, however, we can typecast the enum type to an int to do the comparison, like in the following example.

Program.cs:

```
using System;
namespace Enums
{
     enum Color
    {
        Yellow,
        Blue,
        Green
    }

    internal class Program
    {
        private static void Main(string[] args)
        {
            int myColor = 2;
            if(myColor== (int)Color.Green)
            {
                Console.WriteLine("my color");
            }
           Console.ReadLine();
        }
    }
}
```

Output: my color

B. IFormattable

Program.cs:

```
using System;
namespace Enums
{
    internal enum Color
    {
        Yellow,
        Blue,
        Green
    }

    internal class Program
    {
        private static void Main(string[] args)
        {
            System.Console.WriteLine(Color.Format(typeof(Color), Color.Green, "X"));
            System.Console.WriteLine(Color.Format(typeof(Color), Color.Green, "d"));
            Console.ReadLine();
        }
    }
}
```

Output: 00000002

```
              2
```

Format is the method derived from the IFormatter interface. It's a static method so can be used directly with the enum class defined as Color. It's first parameter is the type of the enum class, the second is the member that must be formatted and the third is the format, in other words hexadecimal or decimal, like we used in the above example and we got a positive result output too.

C. IConvertible

```
using System;
namespace Enums
{
     enum Color
    {
        Yellow,
        Blue,
        Green
    }


internal class Program
    {
        private static void Main(string[] args)
        {
            string[] names;
            names = Color.GetNames(typeof (Color));
            foreach (var name in names)
            {
                Console.WriteLine(name);
            }
            Console.ReadLine();
        }
    }
}
```

Output:

```
Yellow
Blue
Green
```

Note: Each and every code snippet in this article is tried and tested.

GetNames is a static method that accepts Type, in other words an instance of type as a parameter and in return gives an array of strings. Like in the above example, we had an array of 3 members in our enum, therefore thier names are displayed one by one.

The following is another example.

Program.cs

```
using System;
namespace Enums
{
    enum Color
   {
      Yellow,
      Blue,
      Green
   }

   internal class Program
   {
      private static void Main(string[] args)
      {
        Console.WriteLine(Color.Blue.ToString());
        Console.WriteLine(Color.Green.ToString());
        Console.ReadLine();
      }
   }
}
```

Output:

```
Blue
Green
```

As we see in the above example, we converted an enum type to a starting type and got an output too, so, numerous predefined conversion methods can be used to convert an enum from one data type to another.

Point to remember: numerous predefined conversion methods can be used to convert an enum from one data type to another.

Duplicity, default values and initialization

Program.cs:

```
using System;
namespace Enums
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine((byte)Color.Yellow);
         Console.WriteLine((byte)Color.Blue);
         Console.ReadLine();
      }
   }

   enum Color
   {
      Yellow,
      Blue,
      Brown,
      Green,
      Blue

   }
}
```

Output:

Compile time error: The type 'Enums.Color' already contains a definition for 'Blue'

In the preceding example we just repeated the enum member Blue of Color and we got a compile time error, hence we now know that an enum cannot contain two members having the same name. By default if the first value is not specified then the first member takes the value 0 and increments it by one for succeeding members.

Let's use one more example.

Program.cs:

```
using System;
namespace Enums
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine((int)Color.Yellow);
         Console.WriteLine((int)Color.Blue);
         Console.WriteLine((int)Color.Brown);
         Console.WriteLine((int)Color.Green);

         Console.ReadLine();
      }
   }

   enum Color
   {
      Yellow =2,
      Blue,
      Brown=9,
      Green,
   }
}
```

Output:

```
2
3
9
10
```

Surprise! We can always specify the default constant value to any enum member, here we see, we specified the value 2 for yellow, so as per the law of enum, the value of blue will be incremented by one and gets the value 3. We again specified 9 as a default value to Brown and so its successor Green gets incremented by one and gets that value 10.

Moving on to another example,

Program.cs:

```
using System;
namespace Enums
{
    class Program
    {
        static void Main(string[] args)
        {

        }
    }

    enum Color:byte
    {
        Yellow =300 ,
        Blue,
        Brown=9,
        Green,
    }
}
```

Output: Compile time error - Constant value '300' cannot be converted to a 'byte'

We just derived our enum from byte, we know we can do that. We then change the value of Yellow from 2 to 300 and we resulted in a compile time error. Since here our underlying data type was byte, it is as simple as that, that we cannot specify the value to enum members that exceeds the range of underlying data types. The value 300 is beyond the range of byte. It is similar to assigning the beyond range value to a byte data type variable.

The following is another example.

Program.cs:

```
using System;
namespace Enums
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine((int)Color.Yellow);
            Console.WriteLine((int)Color.Blue);
            Console.WriteLine((int)Color.Brown);
            Console.WriteLine((int)Color.Green);

            Console.ReadLine();
        }
    }

    enum Color
    {
        Yellow = 2,
        Blue,
        Brown = 9,
        Green = Yellow
    }
}
```

```
Output: 2
        3
        9
        2
```

Here we initialized Green to Yellow and we did not get an error, so we see, more than one enum members can be initialized with the same constant value.

Point to remember: More than one enum members can be initialized with the same constant value.

Program.cs:

```
using System;
namespace Enums
{
    class Program
    {
        static void Main(string[] args)
        {
            Color.Yellow = 3;
        }
    }
    enum Color
    {
        Yellow = 2,
        Blue,
        Brown = 9,
        Green = Yellow
    }
}
```

Output:

Compile time error: The left-hand side of an assignment must be a variable, property or indexer

In the preceding example, we tried to initialize the enum member out of the scope of the defined enum, in other words in another class and got a compile time error. We must not forget that an enum acts as a constant, that cannot change its value.

Point to remember: An enum acts as a constant, so its value cannot be changed once initialized.

Readability

Program.cs:

```
using System;
namespace Enums
{
    internal enum Color
    {
        Yellow,
        Blue,
        Brown,
        Green
    }

    internal class Program
    {
        private static void Main(string[] args)
        {
            Console.WriteLine(CheckColor(Color.Yellow));
            Console.WriteLine(CheckColor(Color.Brown));
            Console.WriteLine(CheckColor(Color.Green));
            Console.ReadLine();
        }
        public static string CheckColor(Color color)
        {
            switch (color)
            {
                case Color.Yellow:
                    return "Yellow";
                case Color.Blue:
                    return "Blue";
                case Color.Brown:
                    return "Brown";
                case Color.Green:
                    return "Green";
                default:
                    return "no color";

            }
        }
    }
}
```

Output:

```
Yellow
Brown
Green
```

Here, in the above example we declared an enum Color containing various color members.There is a class named program that contains a static method named CheckColor, that has a switch statement checking the color on the basis of the ed parameter to the method, in other words an Enum Color. In the Main method we try to access that CheckColor method, ing various parameters. We see that the the switch statement in the CheckColor method can use any of the datatypes ed and in return the case statements use the name of that type and not the plain int number to compare the result. We see that this made our program more readable. So an enum plays an important role in making the program more readable, structured and easy to grasp.

Circular dependency

Program.cs:

```
using System;
namespace Enums
{
    internal enum Color
    {
        Yellow=Blue,
        Blue
    }

    internal class Program
    {
        private static void Main(string[] args)
        {
        }
    }
}
```

Output:

Compile time error: The evaluation of the constant value for 'Enums.Color.Yellow' involves a circular definition

Like constants we also cannot have circular dependency in enums. We assigned the value Blue to Yellow and Blue in turn is incremented by one as a next enum member, this results in a circular dependency of Blue to yellow and resulted in an error, C# is smart enough to catch these kinds of tricks.

Diving Deep

Let's take some complex scenarios-

Lab 1
Program.cs:

```
using System;
namespace Enums
{
     enum Color
    {

    }

    internal class Program
    {
       private static void Main(string[] args)
       {
          Color color = (Color) -1;
         Console.ReadLine();
       }
    }
}
```

Note: Each and every code snippet in this article is tried and tested.

Output:

Compile time error: To cast a negative value, you must enclose the value in parentheses
'Enums.Color' is a 'type' but is used like a 'variable'

In the preceding example, we are casting a negative value to enum, but the compiler says that while casting a negative value, we must keep that in parenthesis. It's not strange, since C# knows that "-" is also a unary operator, that use of the code above may create confusion for the compiler of whether we are using subtraction or we are typecasting a negative value. So always use parenthesis while typecasting negative values.

Lab 2

Program.cs:

```
using System;
namespace Enums
{
    enum Color
    {
      value__
    }

   internal class Program
    {
       private static void Main(string[] args)
```

```
        {

        }
    }
}
```

Output:

Compile time error: The enumerator name 'value__' is reserved and cannot be used

We clearly see here that we have value__ as a reserved member for the enumerator. The C# compiler like this keyword has a large number of reserved builtin keywords.

It may keep this reserved keyword to keep track of the enum members internally but not sure.

Summary: Let's recall all the points that we need to remember.

1. An enum represents for a constant number and an enum type is known as a distinct type having named constants.
2. We can't declare char as an underlying data type for enum objects because char stores Unicode characters, but enum objects data type can only be number.

3. An enum can't be derived from any other type except that of type byte, sbyte, short, ushort, int, uint, long, or ulong.
4. By default an enum is a sealed class and therefore conforms to all the rules that a sealed class follows, so no class can derive from an enum, in other words a sealed type.
5. The enum type is implicitly derived from System.Enum and so we cannot explicitly derive it from System.Enum.
6. An enum is also derived from the three interfaces IComparable, IFormattable and IConvertible.
7. A numerous predefined conversion methods can be used to convert an enum from one data type to another.
8. More than one enum member can be initialized a same constant value.
9. An enum acts as a constant, so its value cannot be changed once initialized.
10. The enumerator name "value__" is reserved and cannot be used.

Conclusion

With this we completed nearly all the scenarios related to enum. We did many of hands-on labs to clarify our concepts. These may also help you in cracking C# interviews.

Diving Into OOP - Properties in C# (A Practical Approach)

Table of Contents

Introduction

In this “Diving into OOP” will explain properties, their uses and indexers in C#. We'll follow the same way of learning with less theory and more practice. I'll try to explain the concept in-depth.

Properties (the definition)

Let's start with the definition taken from the MSDN.

“A property is a member that provides a flexible mechanism to read, write, or compute the value of a private field. Properties can be used as if they are public data members, but they are actually special methods called accessors. This enables data to be accessed easily and still helps promote the safety and flexibility of methods.”

Properties (the explanation)

As a C# programmer, I must say that properties are something a C# programmer is blessed to use in his code. If we analyze the internals of properties, they are very different from normal variables. Internally, properties are methods that do not have their own memory like variables have. We can leverage a property to write our custom code whenever we access a property. We can access the code when we call/execute properties or during of declaration too, but this is not possible with variables. A property in easy language is a class member and is encapsulated and abstracted from the end developer who is accessing the property. A property can contain lots of code that an end user does not know. An end user only cares to use that property like a variable.

Let's start with some coding now.

Lab 1

Create a simple console application and name it “Properties”. Add a class named “Properties” to it. You can choose the names of the projects and classes depending on your wishes. Now try to create a property in that class as shown below.

Properties.cs

```
namespace Properties
{
   public class Properties
   {
      public string Name{}
   }
}
```

Try to run/build the application, what do we get?

Output: *Error 'Properties.Properties.Name': property or indexer must have at least one accessor*

In the preceding example, we created a property named Name of type string. The error we got is very self-explanatory; it says a property must have an accessor, in other words a get or a set. This means we need something in our property that gets accessed, whether to set the property value or get the property value. Unlike variables, properties cannot be declared without an accessor.

Lab 2

Let's assign a get accessor to our property and try to access that property in the Main method of the Program.cs file.

Properties.cs

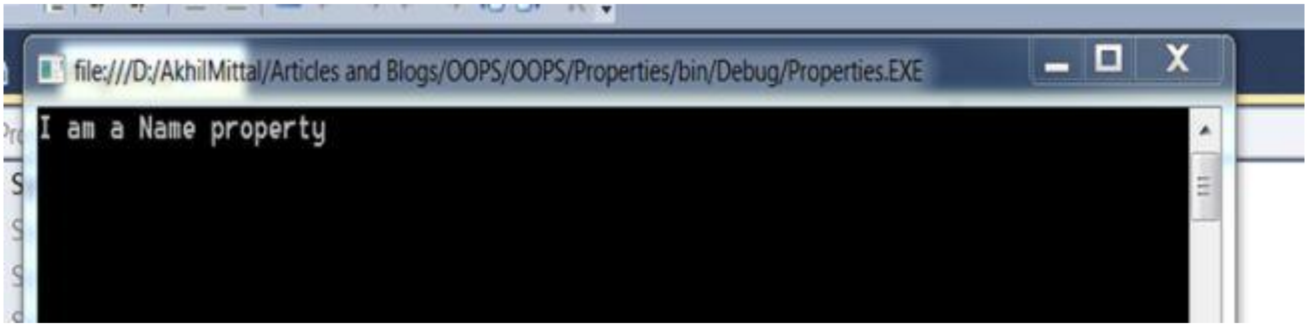
```
namespace Properties
{
   public class Properties
   {
      public string Name
      {
         get
         {
             return "I am a Name property";
         }
      }
   }
}
```

Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Properties
{
    class Program
    {
        static void Main(string[] args)
        {
            Properties properties=new Properties();
            Console.WriteLine(properties.Name);
            Console.ReadLine();
        }
    }
}
```

Try to run/build the application. What do you get?

Output:



file:///D:/AkhilMittal/Articles and Blogs/OOPS/OOPS/Properties/bin/Debug/Properties.EXE

```
I am a Name property
```

It says “I am a Name property”. This signifies that our get successor got called when I tried to access the property or tried to fetch the value of a Property.

Get accessor

Now declare one more property in the Properties class and name it Age that returns an integer. It calculates the age of a person by calculating the difference between his date of birth and current date.

Properties.cs

```
using System;

namespace Properties
{
    public class Properties
    {
        public string Name
        {
            get
            {
                return "I am a Name property";
            }
        }

        public int Age
        {
            get
            {
                DateTime dateOfBirth=new DateTime(1984,01,20);
                DateTime currentDate = DateTime.Now;
                int age = currentDate.Year - dateOfBirth.Year;
                return age;
            }
        }
    }
}
```

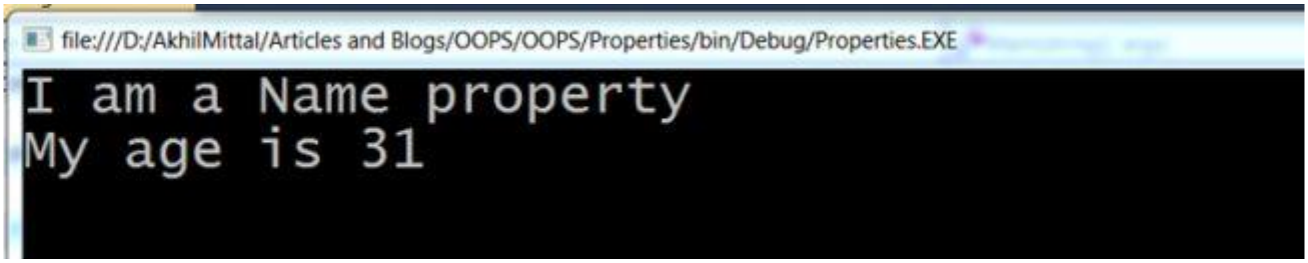
Call the “Age” property in the same way as done for Name.

Program.cs

```
using System;
namespace Properties
{
    class Program
    {
        static void Main(string[] args)
        {
            Properties properties=new Properties();
            Console.WriteLine(properties.Name);
            Console.WriteLine("My age is " + properties.Age);
            Console.ReadLine();
        }
    }
}
```

Try to run/build the application, what do we get?

Output:



file:///D:/AkhilMittal/Articles and Blogs/OOPS/OOPS/Properties/bin/Debug/Properties.EXE

```
I am a Name property
My age is 31
```

It returns the correct age subjective to the date of birth provided. Did you notince something here? Our property contains some code and logic to calculate age, but the caller, Program.cs, is not aware of the logic. It only cares about using that Property. Therefore we see that a property encapsulates and abstracts its functionality from the end user, in our case it's a developer.

Point to remember: The Get accessor is only used to read a property value. A property having only "get" cannot be set with any value from the caller. This means a caller/end user can only access that property in read mode.

Set accessor

Let's start with a simple example-

Lab 1

Properties.cs

```
using System;
namespace Properties
{
    public class Properties
    {
        public string Name
        {
            get { return "I am a Name property"; }
        }

        public int Age
        {
            get
            {
                DateTime dateOfBirth = new DateTime(1984, 01, 20);
                DateTime currentDate = DateTime.Now;
                int age = currentDate.Year - dateOfBirth.Year;
                Console.WriteLine("Get Age called");
                return age;
            }
            set
            {
                Console.WriteLine("Set Age called " + value);
            }
        }
    }
}
```

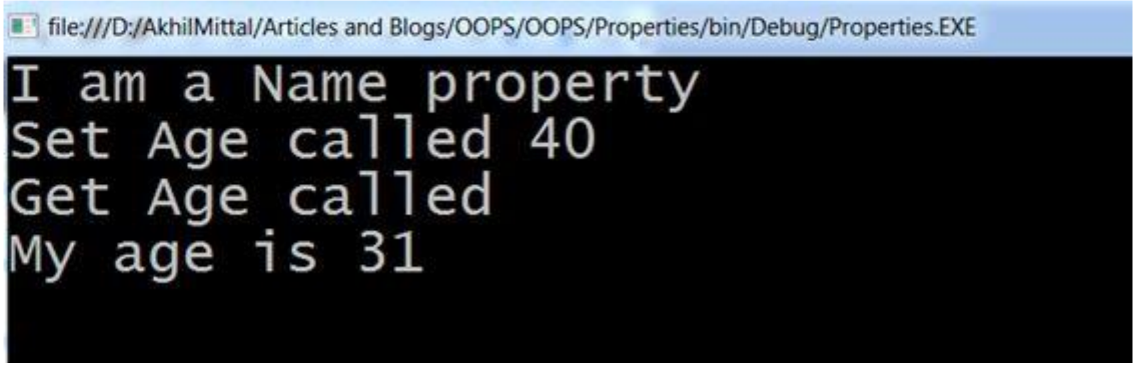
Call the “Age” property in the same way as done for Name.

Program.cs

```
using System;

namespace Properties
{
    class Program
    {
        static void Main(string[] args)
        {
            Properties properties=new Properties();
            Console.WriteLine(properties.Name);
            properties.Age = 40;
            Console.WriteLine("My age is " + properties.Age);
            Console.ReadLine();
        }
    }
}
```

Run the application-

Output:


file:///D:/AkhilMittal/Articles and Blogs/OOPS/OOPS/Properties/bin/Debug/Properties.EXE

```
I am a Name property
Set Age called 40
Get Age called
My age is 31
```

In the preceding example, I made a w minor changes in the get accessor, in other words just printing that control is in the “Get accessor” and introduced a “Set” in the Age property too. Everything else remains the same. Now when I call the Name property, it works as it worked earlier. Since we used “Set” we are now allowed to set the value of a property. When I do *properties.Age = 40;* that means I am setting the value 40 for that property. We can say a property can also be used to assign some value. In this case the Set accessor is called, as soon as we set a value to property. Later on when we access that same property, again our get accessor is called that returns the value with some custom logic. We have a drawback here. As we see, whenever we call get we get the same value and not the value that we assigned to that property, in other words because get has its custom fixed logic. Let's try to overcome this situation.

Lab 2

The example I am about to explain makes use of a private variable. But you can also use Automatic Properties to do it. I'll intentionally use a variable to clarify things.

Properties.cs

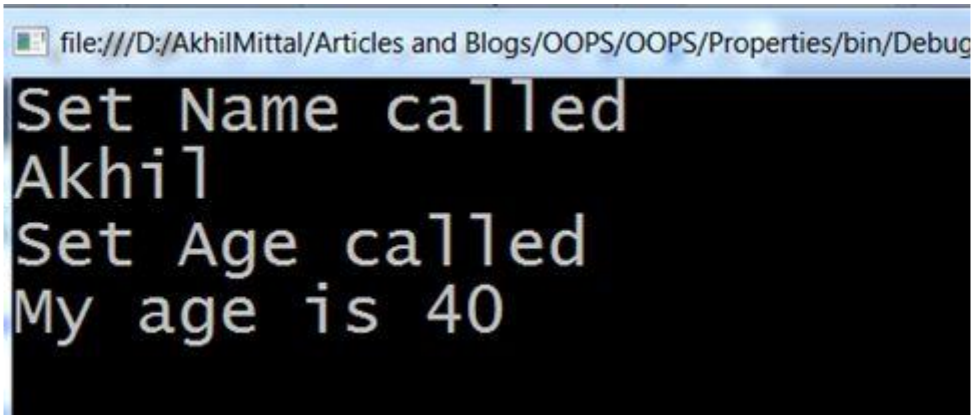
```
using System;
namespace Properties
{
   public class Properties
   {
      private string name;
      private int age;

      public string Name
      {
            get { return name; }
            set
            {
                Console.WriteLine("Set Name called ");
                name = value;
            }
        }

        public int Age
        {
            get { return age; }
            set
            {
                Console.WriteLine("Set Age called ");
                age = value;
            }
        }
    }
}
```

Program.cs

```
using System;
namespace Properties
{
    class Program
    {
        static void Main(string[] args)
        {
            Properties properties=new Properties();
            properties.Name = "Akhil";
            Console.WriteLine(properties.Name);
            properties.Age = 40;
            Console.WriteLine("My age is " + properties.Age);
            Console.ReadLine();
        }
    }
}
```

Run the application-

Output:



file:///D:/AkhilMittal/Articles and Blogs/OOPS/OOPS/Properties/bin/Debug

```
Set Name called
Akhil
Set Age called
My age is 40
```

Now you see, we get the same value that we assigned to the Name and Age properties. When we access these properties the get accessor is called and it returns the same value as we set them to. Here the properties internally use a local variable to hold and sustain the value.

In daily programming, we normally create a Public property that can be accessed outside the class. However the variable it is using internally could be private.

Point to remember: The variable used for a property should be of the same data type as the data type of the property.

In our case we used the variables name and age, they share the same datatype as their respective properties do. We don't use variables since there might be scenarios in which we do not have control over those variables, the end user can change them at any point of code without maintaining the change stack. Moreover one major use of properties is for the user to associate some logic or action when some change on the variable occurs, therefore when we use properties, we can easily track the value changes in the variable.
When using Automatic Properties, they do this internally, in other words we don't need to define an extra variable to do so, as shown below.

Lab 3

Properties.cs

```
using System;
namespace Properties
{
   public class Properties
```

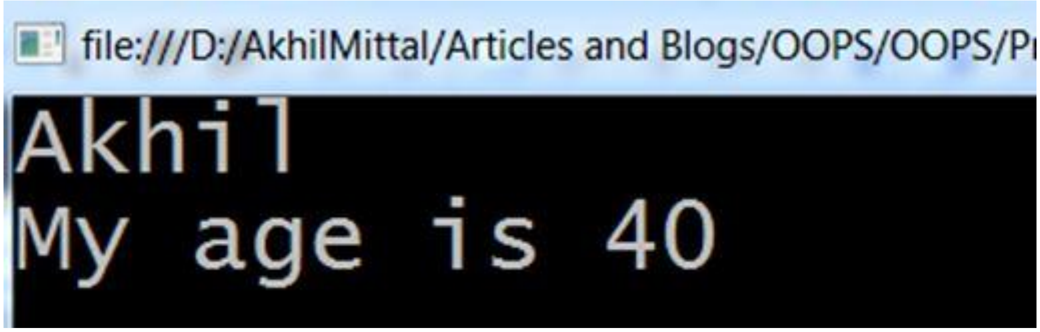
```
    {
        private string name;
        private int age;

        public string Name { get; set; }

        public int Age { get; set; }
    }
}
```

Program.cs

```
using System;

namespace Properties
{
    class Program
    {
        static void Main(string[] args)
        {
            Properties properties=new Properties();
            properties.Name = "Akhil";
            Console.WriteLine(properties.Name);
            properties.Age = 40;
            Console.WriteLine("My age is " + properties.Age);
            Console.ReadLine();
        }
    }
}
```

Run the application-

Output:



```
file:///D:/AkhilMittal/Articles and Blogs/OOPS/OOPS/Pı
Akhil
My age is 40
```

The following are automatic properties:

public string Name { get; set; }
public int Age { get; set; }

I hope now you understand how to define a property and use it.

Readonly

A property can be made read-only by only providing the get accessor. We do not provide a set accessor, if we do not want our property to be initialized or to be set from outside the scope of the class.

Properties.cs

```
using System;
namespace Properties
{
    public class Properties
    {
        private string name="Akhil";
        private int age=32;

        public string Name
        {
            get { return name; }
        }

        public int Age { get { return age; } }
    }
}
```

Program.cs

```
using System;
namespace Properties
{
   class Program
   {
      static void Main(string[] args)
      {
         Properties properties=new Properties();
         properties.Name = "Akhil";
         Console.WriteLine(properties.Name);
         properties.Age = 40;
         Console.WriteLine("My age is " + properties.Age);
         Console.ReadLine();
      }
   }
}
```

Build the application, we get the following output-

Error Property or indexer 'Properties.Properties.Age' cannot be assigned to -- it is read only
Error Property or indexer 'Properties.Properties.Name' cannot be assigned to -- it is read only

In the main method of the Program class, we tried to set the value of Age and Name property by:

1. properties.Name = "Akhil";
2. properties.Age = 40;

But since they aew readonly, in other words only have a get accessor, we encountered a compile time error.

Write-Only

A property can also be made write-only, in other words the reverse of read-only. In this case you'll be only allowed to set the value of the property but can't access it because we don't have a get accessor in it.

Properties.cs

```
using System;
```

```
namespace Properties
{
   public class Properties
   {
      private string name;
      private int age;

      public string Name
      {
         set { name=value; }
      }
      public int Age { set { age = value; } }
   }
}
```

Program.cs

```
using System;
namespace Properties
{
   class Program
   {
      static void Main(string[] args)
      {
         Properties properties=new Properties();
         properties.Name = "Akhil";
         Console.WriteLine(properties.Name);
         properties.Age = 40;
         Console.WriteLine("My age is " + properties.Age);
         Console.ReadLine();
      }
   }
}
```

Build the application, we get the following output-

Error The property or indexer 'Properties.Properties.Age' cannot be used in this context because it lacks the get accessor
Error The property or indexer 'Properties.Properties.Name' cannot be used in this context because it lacks the get accessor

In the preceding example, our property is marked only with a set accessor, but we tried to access those properties in our main program with:

```
Console.WriteLine(properties.Name);
Console.WriteLine("My age is " + properties.Age);
```

That means we tried to call the get accessor of the property that is not defined, so we again ended up in a compile-time error.

Insight of Properties in C#

Lab 1

Can we define properties as two different set of pieces? The answer is **no**.

Properties.cs

```
using System;
namespace Properties
{
   public class Properties
   {
      private string name;

      public string Name
      {
         set { name=value; }
      }

      public string Name
      {
         get { return name; }
      }
   }
}
```

Build the project, we get the following compile time error-

Error The type 'Properties.Properties' already contains a definition for 'Name'

Here I tried to create a single property segregated into two accessors. The compiler treats a property name as a single separate property, so we cannot define a property with two names having a different accessor.

Lab 2

Can we define properties the same as a previously defined variable? The answer is **no**.

Properties.cs

```
using System;
namespace Properties
{
   public class Properties
   {
      private string name;

      public string name
      {
         set { name=value; }
         get { return name; }
      }
   }
}
```

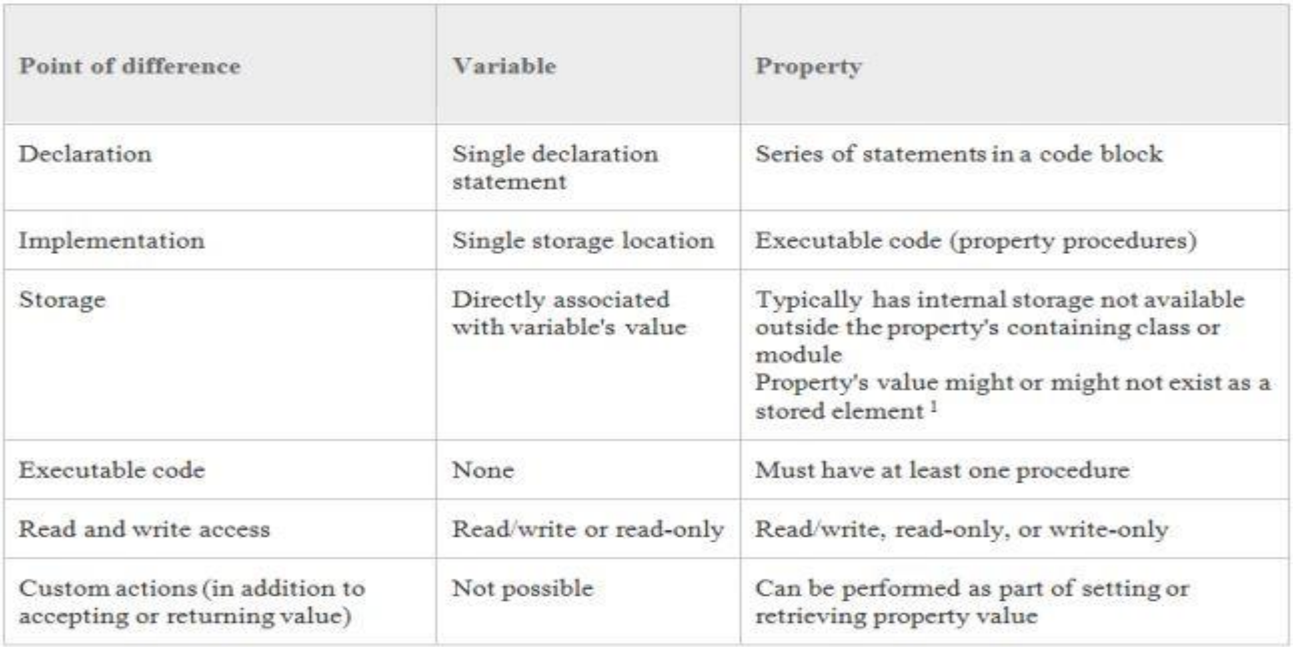
Build the project; we get the following compile time error-

Error The type 'Properties.Properties' already contains a definition for ‘name"

Again, we cannot have a variable and a property with the same name. They may differ on the grounds of case sensitivity, but they cannot share a common name with the same case because when they are accessed the compiler might be confused about whether you are trying to access a property or a variable.

Properties vs Variables

It is a misconception that variables are faster in execution than properties. I do not deny that but this may not be true in every case or can vary case to case. A property, as I explained, internally executes a function/method whereas a variable uses/initializes memory when used. At times properties are not slower than variables since the property code is internally rewritten to memory access.



Point of difference	Variable	Property
Declaration	Single declaration statement	Series of statements in a code block
Implementation	Single storage location	Executable code (property procedures)
Storage	Directly associated with variable's value	Typically has internal storage not available outside the property's containing class or module Property's value might or might not exist as a stored element [1]
Executable code	None	Must have at least one procedure
Read and write access	Read/write or read-only	Read/write, read-only, or write-only
Custom actions (in addition to accepting or returning value)	Not possible	Can be performed as part of setting or retrieving property value

Static Properties

Like variables and methods, a property can also be marked static.

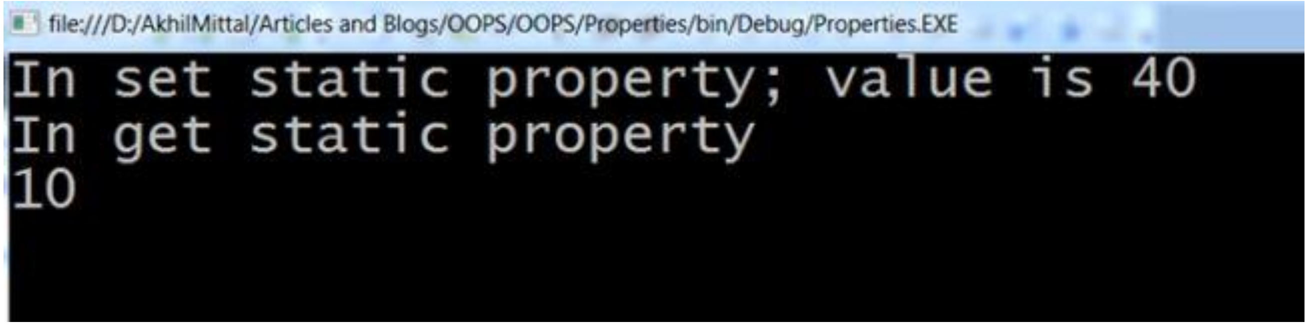
Properties.cs

```
using System;
namespace Properties
{
   public class Properties
   {
      public static int Age
      {
         set
         {
            Console.WriteLine("In set static property; value is " + value);
         }
         get
         {
            Console.WriteLine("In get static property");
            return 10;
         }
      }
   }
}
```

Program.cs

```
using System;
namespace Properties
{
    class Program
    {
        static void Main(string[] args)
        {
            Properties.Age = 40;
            Console.WriteLine(Properties.Age);
            Console.ReadLine();
        }
    }
}
```

Output:



file:///D:/AkhilMittal/Articles and Blogs/OOPS/OOPS/Properties/bin/Debug/Properties.EXE

```
In set static property; value is 40
In get static property
10
```

In the preceding example, I created a static Age property. When I tried to access it, you can see it is accessed via class name, like all static members are subjected to. So properties also inherit the static functionality like all C# members, no matter whether it is variable or a method. They'll be accessed via class name only.

Properties return type

Lab 1

Properties.cs

```
using System;
namespace Properties
```

```
{
  public class Properties
    {
        public void AbsProperty
        {
            get
            {
                Console.WriteLine("Get called");
            }
        }
    }
}
```

Compile the program- The output is a compile time error.

Error 'AbsProperty': property or indexer cannot have void type

Point to remember: A property cannot have a void return type.

Lab 2

Just try to return a value from a “set” accessor.

Properties.cs

```
using System;
namespace Properties
{
  public class Properties
    {
        public int Age
        {
            set { return 5; }
        }
    }
}
```

Compile the program-

Error Since 'Properties.Properties.Age.set' returns void, a return keyword must not be followed by an object expression

Here the compiler understands the “set” accessor as a method that returns void and takes a parameter to initialize the value. So set cannot be expected to return a value. If we just leave a return statement empty and remove 5, we do not get any error and the code compiles.

```
using System;
namespace Properties
{
  public class Properties
    {
        public int Age
        {
            set { return ; }
        }
    }
}
```

Value Keyword: We have a reserved keyword named value.

```
using System;
namespace Properties
{
  public class Properties
    {
        public string Name
        {
            set { string value; }
        }
    }
}
```

Just compile the preceding code, we get a compile time error as follows-

Error A local variable named 'value' cannot be declared in this scope because it would give a different meaning to 'value', that is already used in a 'parent or current' scope to denote something else

This signifies that “value” is a reserved keyword here. So one cannot declare a variable named value in the “set” accessor since it may give a different meaning to an already reserved keyword value.

Abstract Properties

Lab 1

Yes, we can also have abstract properties. Let's see how it works.

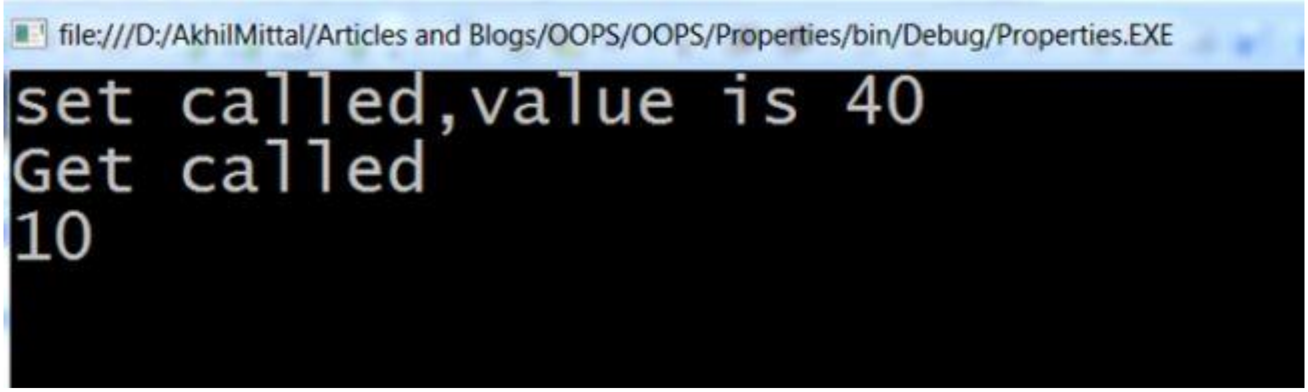
Properties.cs

```
using System;
namespace Properties
{
    public abstract class BaseClass
    {
        public abstract int AbsProperty { get; set; }
    }

    public class Properties : BaseClass
    {
        public override int AbsProperty
        {
            get
            {
                Console.WriteLine("Get called");
                return 10;
            }
            set { Console.WriteLine("set called,value is " + value); }
        }
    }
}
```

Program.cs

```
using System;
namespace Properties
{
   class Program
    {
        static void Main(string[] args)
        {
            Properties prop=new Properties();
            prop.AbsProperty = 40;
            Console.WriteLine(prop.AbsProperty);
            Console.ReadLine();
        }
    }
}
```

Output:



file:///D:/AkhilMittal/Articles and Blogs/OOPS/OOPS/Properties/bin/Debug/Properties.EXE

```
set called,value is 40
Get called
10
```

In the preceding example, I just created a base class named “BaseClass” and defined an abstract property named Absproperty. Since the property is abstract it follows the rules of being abstract as well. I inherited my “Properties” class from BaseClass and given the body to that abstract property. Since the property was abstract I need to override it in my derived class to add functionality to it. So I used the override keyword in my derived class.

In the base class, the abstract property has no body at all, neither for “get” nor for “set”, so we need to implement both of the accessors in our derived class, as shown in the “Properties” class.

Point to remember: If one does not mark a property defined in a derived class as override, it will by default be considered to be new.

Lab 2

Properties.cs

```
using System;
namespace Properties
{
    public abstract class BaseClass
    {
        public abstract int AbsProperty { get; }
    }

    public class Properties : BaseClass
    {
        public override int AbsProperty
        {
            get
            {
                Console.WriteLine("Get called");
                return 10;
            }
            set { Console.WriteLine("set called,value is " + value); }
        }
    }
}
```

Program.cs

```
using System;
namespace Properties
{
    class Program
    {
        static void Main(string[] args)
        {
            Properties prop=new Properties();
            prop.AbsProperty = 40;
            Console.WriteLine(prop.AbsProperty);
            Console.ReadLine();
        }
    }
}
```

Output: Compile time error.

Error 'Properties.Properties.AbsProperty.set': cannot override because 'Properties.BaseClass.AbsProperty' does not have an overridable set accessor

In the preceding lab example, I just removed “set” from the AbsProperty in the Base class. All the code remains the same. Now here we are trying to override the set accessor too in the derived class that is missing in the base class, therefore the compiler will not allow you to override a successor that is not declared in the base class, hence the result is a compile time error.

Point to remember: You cannot override an accessor that is not defined in a base class abstract property.

Properties in Inheritance

Just follow the code.

Properties.cs

```
using System;
namespace Properties
{
 public class PropertiesBaseClass
  {
    public int Age
    {
      set {}
    }
  }

  public class PropertiesDerivedClass:PropertiesBaseClass
  {
    public int Age
    {
      get { return 32; }
    }
  }
}
```

Program.cs

```
namespace Properties
{
  class Program
  {
    static void Main(string[] args)
    {
      PropertiesBaseClass pBaseClass=new PropertiesBaseClass();
      pBaseClass.Age = 10;
      PropertiesDerivedClass pDerivedClass=new PropertiesDerivedClass();
      ((PropertiesBaseClass) pDerivedClass).Age = 15;
      pDerivedClass.Age = 10;
    }
    }
}
```

As you can see in the preceding code, in the Properties.cs file I created two classes. One is Base, in other words PropertiesBaseClass and the second is Derived, in other words PropertiesDerivedClass. I intentionally declared a set accessor in the Base class and a get in the Derived class for the same property name, Age. Now this case may give you the feeling that when compiled, our code of property Age will become one, in other words it will take the set from the Base class and the get from the derived class and combine them into a single entity of Age property. But actually that is not the case. The compiler treats these properties differently and does not consider them to be the same. In this case the property in the derived class actually hides the property in the base class, they are not the same but independent properties.The same concept of method hiding applies here too. To use the property of the base class from a derived class object, you need to cast it to the base class and then use it.

When you compile the preceding code, you get a compile time error as follows-

Error Property or indexer 'Properties.PropertiesDerivedClass.Age' cannot be assigned to -- it is read only

In other words we can do ((*PropertiesBaseClass*) pDerivedClass).Age = 15;

But we cannot do pDerivedClass.Age = 10; because the derived class property has no "set" accessor.

Summary

Let's recall all the points that we need to remember-

- The variable used for a property should be the same data type as the data type of the property.
- A property cannot have a void return type.
- If one does not mark a property defined in the derived class as override, it will by default be considered as new.

- You cannot override an accessor that is not defined in a base class abstract property.
- A get accessor is only used to read a property value. A property having only a get cannot be set with any value from the caller.

Conclusion

We learned a lot about properties in C#. I hope you now understand properties well.

Diving Into OOP - Indexers in C# (A Practical Approach)

Table of Contents

Introduction

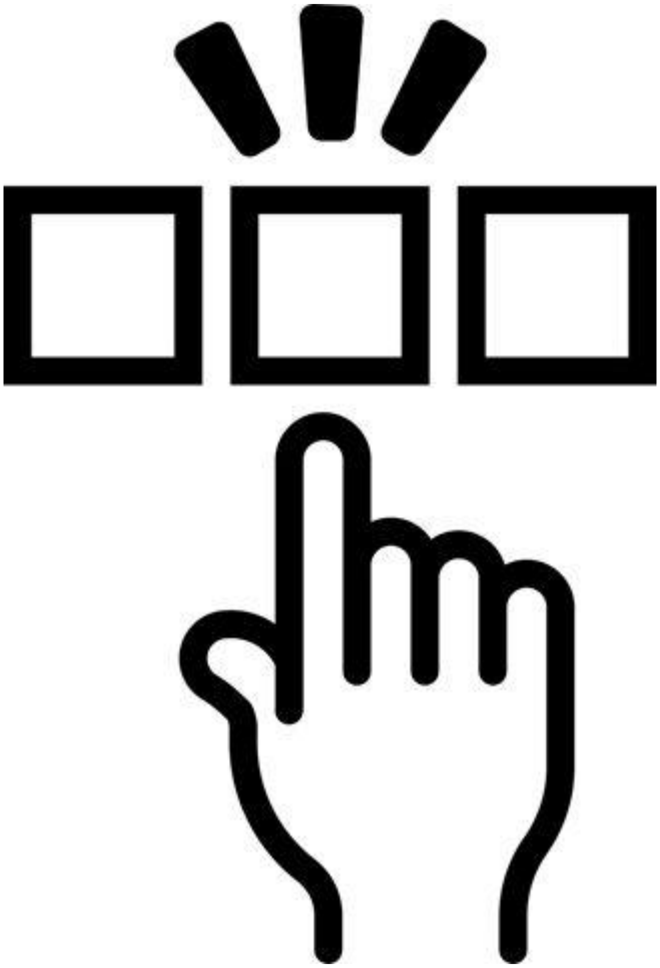
In my last article of this series we learned about properties in C#. This article of the series "Diving into OOP" will explain all about indexers in C#, their uses and practical implementation. We'll follow the same way of learning, in other words less theory and more practice. I'll try to explain the concept in-depth.

Indexers in C# (The definition)

Let's use the definition from: MSDN.

"*Indexers allow instances of a class or struct to be indexed just like arrays. Indexers resemble properties except that their accessors take parameters.*"

Indexers(The explanation)



As the definition says, indexers allow us to leverage the capability of accessing the class objects as an array. For a better understanding, create a console application named Indexers and add a class to it named Indexer. We'll use this class and project to learn Indexers. Make the class public, do not add any code for now and in Program.cs add following code.

Lab 1

```
namespace Indexers
{
   class Program
   {
      static void Main(string[] args)
      {
         Indexer indexer=new Indexer();
```

```
            indexer[1] = 50;
        }
    }
}
```

Compile the code. We get-

Error Cannot apply indexing with [] to an expression of type 'Indexers.Indexer'

I just created an object of the Indexer class and tried to use that object as an array. Since it was not actually an array, it produced a compile time error.

Lab 2

Indexer.cs

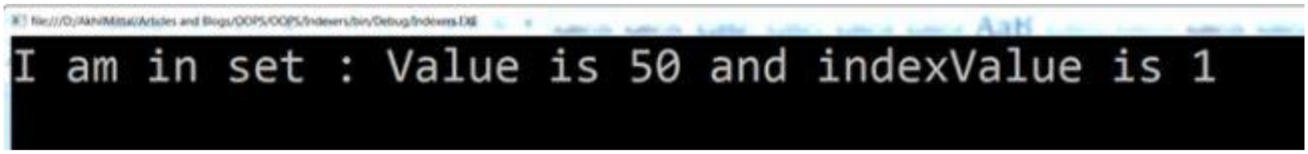
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Indexers
{
    public class Indexer
    {
        public int this[int indexValue]
        {
            set
            {
                Console.WriteLine("I am in set : Value is " + value + " and indexValue is " + indexValue);

                Console.ReadLine();
            }
        }
    }
}
```

Program.cs

```
namespace Indexers
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            Indexer indexer=new Indexer();
            indexer[1] = 50;
        }
    }
}
```

Output:


file:///D:/AkhilMittal/Articles and Blogs/OOPS/OOPS/Indexers/bin/Debug/Indexers.EXE

```
I am in set : Value is 50 and indexValue is 1
```

Here we just used an indexer to index my object of the class Indexer. Now my object can be used as an array to access various object values.

Implementation of indexers is derived from a property known as “this”. It takes an integer parameter indexValue. Indexers are different from properties. In properties, when we want to initialize or assign a value, the “set” accessor, if defined, is automatically called. And the keyword “value” in the “set” accessor was used to hold or keep track of the assigned value to our property. In the preceding example, *indexer[1] = 50;* calls the “set” accessor of the “this” property, in other words the indexer 50 becomes the value and 1 becomes the index of that value.

Lab 3

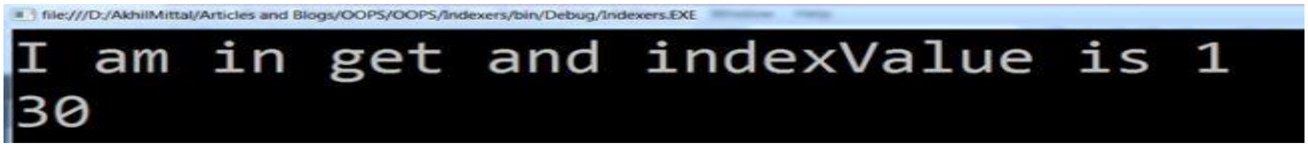
Indexer.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Indexers
{
    public class Indexer
    {
        public int this[int indexValue]
        {
            set
            {
                Console.WriteLine("I am in set : Value is " + value + " and indexValue is " + indexValue);

            }
             get
            {
                Console.WriteLine("I am in get and indexValue is " + indexValue);
                return 30;
            }
        }
    }
}
```

Program.cs

```
using System;
namespace Indexers
{
    class Program
    {
        static void Main(string[] args)
        {
            Indexer indexer=new Indexer();
            Console.WriteLine(indexer[1]);
            Console.ReadKey();
        }
    }
}
```

Output:


file:///D:/AkhilMittal/Articles and Blogs/OOPS/OOPS/Indexers/bin/Debug/Indexers.EXE

```
I am in get and indexValue is 1
30
```

In the preceding code snippet, I used get as well, to access the value of the indexer. Properties and Indexers work on the same set of rules. There is a bit of a difference on how we use them. When we do indexer[1] then that means the “get” accessor is called and when we assign some value to indexer[1] then the “set” accessor is called. When implementing indexer code we need to take care that when we access an indexer it is accessed in the form of a variable and that too is an array parameter.

Data-Types in Indexers

Lab 1

Indexer.cs

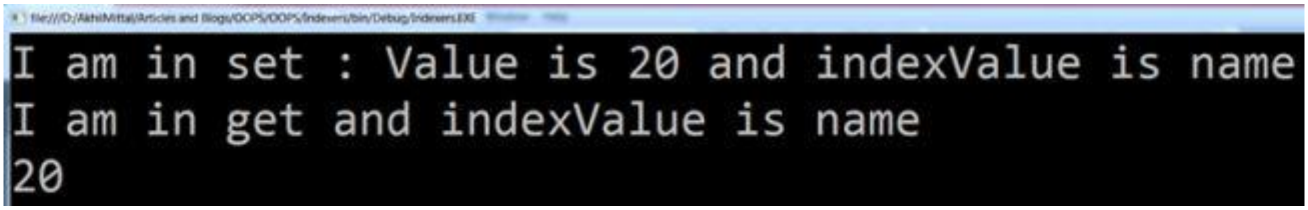
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Indexers
{
    public class Indexer
    {
        public int Index;
        public int this[string indexValue]
        {
            set
            {
          Console.WriteLine("I am in set : Value is " + value + " and indexValue is " + indexValue);
                Index = value;
            }
             get
            {
                Console.WriteLine("I am in get and indexValue is " + indexValue);
                return Index;
            }
        }
      }
 }
```

Program.cs

```
using System;

namespace Indexers
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            Indexer indexer=new Indexer();
            indexer["name"]=20;
            Console.WriteLine(indexer["name"]);
            Console.ReadKey();
        }
    }
}
```

Output


file:///D:/AkhilMittal/Articles and Blogs/OOPS/OOPS/Indexers/bin/Debug/Indexers.EXE

```
I am in set : Value is 20 and indexValue is name
I am in get and indexValue is name
20
```

The “this” property, in other words indexers, have a return value. In our example the return value was an integer. The square brackets along with “this” can also hold other data types and not only an integer. In the preceding example I tried to explain this using a string parameter type for “this” : public int this [string indexValue].

The string parameter “indexValue” has a value “name”, like we ed in the Main method of Program.cs. So one can have more than one indexer in a class deciding what should be the data type of the parameter value of an array. An indexer, like properties, follow the same rules of inheritance and polymorphism.

Indexers in interfaces

Like Properties and Methods, Indexers can also be declared in interfaces.

For practical implementation, just create an interface named IIndexers with the following code.

```
namespace Indexers
{
    interface IIndexers
    {
        string this[int indexerValue] { get; set; }
    }
}
```

Here, an indexer is declared with an empty get and set accessor, that returns string values.

Now we need a class that implements this interface. You can define a class of your choice and implement that using the IIndexers interface.

Indexer.cs

```
using System;

namespace Indexers
{
    public class IndexerClass:IIndexers
    {
        readonly string[] _nameList = { "AKhil","Bob","Shawn","Sandra" };

        public string this[int indexerValue]
        {
            get
            {
                return _nameList[indexerValue];
            }
            set
            {
                _nameList[indexerValue] = value;
            }
        }
    }
}
```

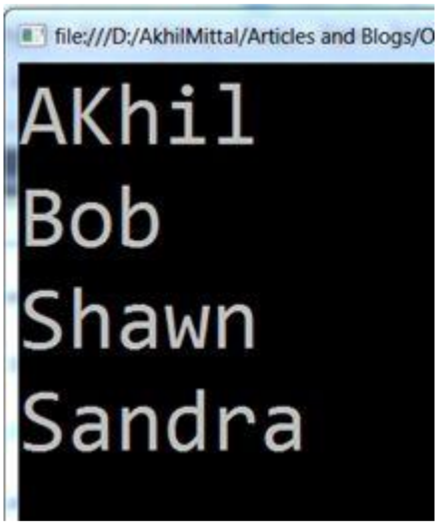
The class has a default array of strings that holds names. Now we can implement an interface defined indexer in this class to write our custom logic to fetch names on the base of indexerValue. Let's call this in our main method.

Program.cs

```
using System;

namespace Indexers
{
    class Program
    {
        static void Main(string[] args)
        {
           IIndexers iIndexer=new IndexerClass();
            Console.WriteLine(iIndexer[0]);
            Console.WriteLine(iIndexer[1]);
            Console.WriteLine(iIndexer[2]);
            Console.WriteLine(iIndexer[3]);
            Console.ReadLine();

        }
    }
}
```

Run the application.

Output:



file:///D:/AkhilMittal/Articles and Blogs/O

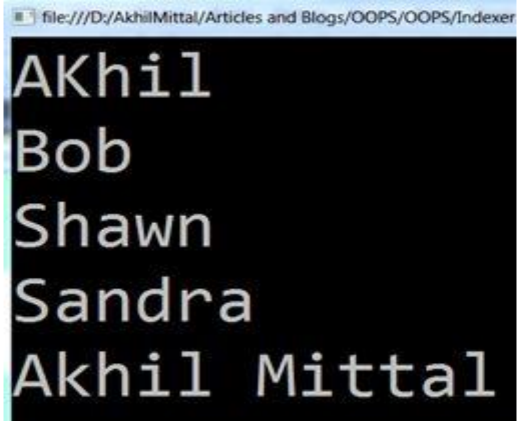
```
AKhil
Bob
Shawn
Sandra
```

In the main method, we took an interface reference to create an object of the Indexer class and we accessed that object array using indexer values like an array. It gives the names one by one.

Now if I want to access a “set” accessor as well, I can easily do that. To check this, just add two more lines where you set the value in the indexer.

```
1. iIndexer[2] = "t;Akhil Mittal";
2. Console.WriteLine(iIndexer[2]);
```

I set the value of the second element as a new name. Let's see the output.



file:///D:/AkhilMittal/Articles and Blogs/OOPS/OOPS/Indexer

```
AKhil
Bob
Shawn
Sandra
Akhil Mittal
```

Indexers in Abstract class

Like we used indexers in interfaces, we can also use indexers in abstract classes. I'll use the same logic of source code that we used in interfaces, so that you can relate how it works in abstract class as well. Just define a new class that should be abstract and should contain an abstract indexer with empty get and set.

AbstractBaseClass

```
namespace Indexers
{
    public abstract class AbstractBaseClass
    {
        public abstract string this[int indexerValue] { get; set; }
    }
}
```

Define the derived class, inheriting from the abstract class.

IndexerClass

We here use an override in the indexer to override the abstract indexer declared in the abstract class.

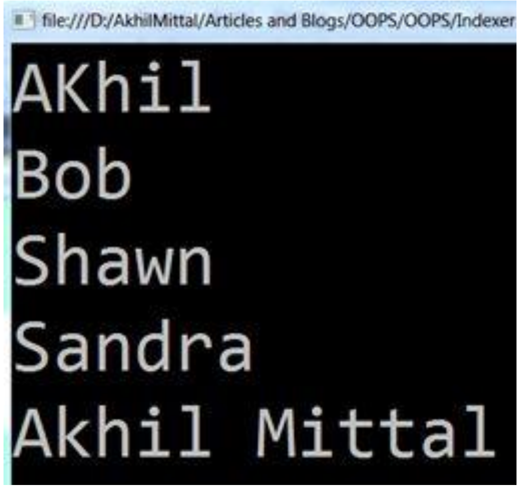
```
using System;
namespace Indexers
{
    public class IndexerClass:AbstractBaseClass
    {
        readonly string[] _nameList = { "AKhil","Bob","Shawn","Sandra" };

        public override string this[int indexerValue]
        {
            get
            {
                return _nameList[indexerValue];
            }
            set
            {
                _nameList[indexerValue] = value;
            }
        }
    }
}
```

Program.cs

We'll use a reference of an abstract class to create an object of the Indexer class.

```
using System;
namespace Indexers
{
    class Program
    {
        static void Main(string[] args)
        {
           AbstractBaseClass absIndexer=new IndexerClass();
            Console.WriteLine(absIndexer[0]);
            Console.WriteLine(absIndexer[1]);
            Console.WriteLine(absIndexer[2]);
            Console.WriteLine(absIndexer[3]);
            absIndexer[2] = "Akhil Mittal";
            Console.WriteLine(absIndexer[2]);
            Console.ReadLine();
        }
    }
}
```

Output:



```
AKhil
Bob
Shawn
Sandra
Akhil Mittal
```

All of the preceding code is self-explanatory. You can explore more scenarios by yourself for a better understanding.

Indexer Overloading

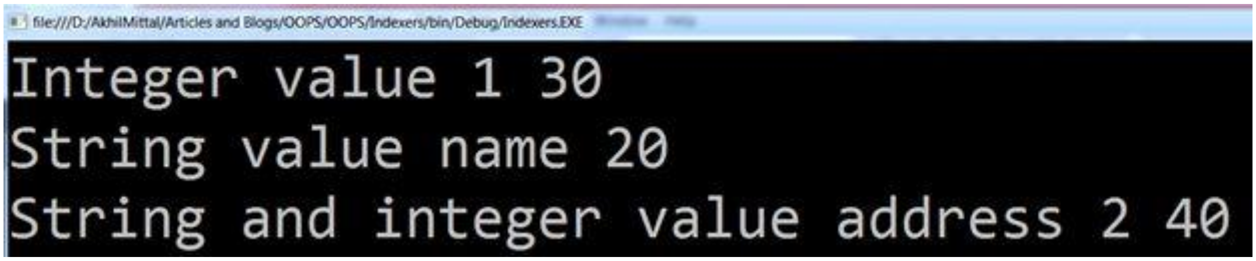
Indexer.cs

```
using System;
namespace Indexers
{
    public class Indexer
    {
        public int this[int indexerValue]
        {
            set
            {
                Console.WriteLine("Integer value " + indexerValue + " " + value);
            }
        }

        public int this[string indexerValue]
        {
            set
            {
                Console.WriteLine("String value " + indexerValue + " " + value);
            }
        }

        public int this[string indexerValue, int indexerintValue]
        {
            set
            {
                Console.WriteLine("String and integer value " + indexerValue + " " + indexerintValue + " "
 + value);
            }
        }
    }
}
```

Program.cs

```
using System;

namespace Indexers
{
    class Program
    {
        static void Main(string[] args)
        {
            Indexer indexer=new Indexer();
            indexer[1] = 30;
            indexer["name"]=20;
            indexer["address",2] = 40;
            Console.ReadLine();
        }
    }
}
```

Output:


file:///D:/AkhilMittal/Articles and Blogs/OOPS/OOPS/Indexers/bin/Debug/Indexers.EXE

```
Integer value 1 30
String value name 20
String and integer value address 2 40
```

In the preceding example, we see the indexer's signature is in the actual count of the actual parameters and data types irresepective of the names of the arguments/parameters or return value of the indexers. This allows us to overload indexers as we do in method overloading. You can read more about method over loading. Here now we have overloaded indexers that take integer, string integer and string combined as actual parameters. Like methods cannot be overloaded on the base of return types, so are indexers. Indexers follow the same methodology of overload like methods do.

Point to remember: Like indexers, we cannot overload properties. Properties are more like knowing by name and indexers on the other hand is more like knowing by signature.

Static Indexers

In the example that we discussed in the last section, just add a static keyword to the indexer signature.

```
public static int this[int indexerValue]
{
    set
    {
        Console.WriteLine("Integer value " + indexerValue + " " + value);
    }
}
```

Compile the program. We get a compile time error-

Error The modifier 'static' is not valid for this item

The error clearly indicates that an indexer cannot be marked static. An indexer can only be a class instance member but not static, on the other hand a property can be static too.

Point to remember: Properties can be static but indexers cannot be.

Inheritance/Polymorphism in Indexers

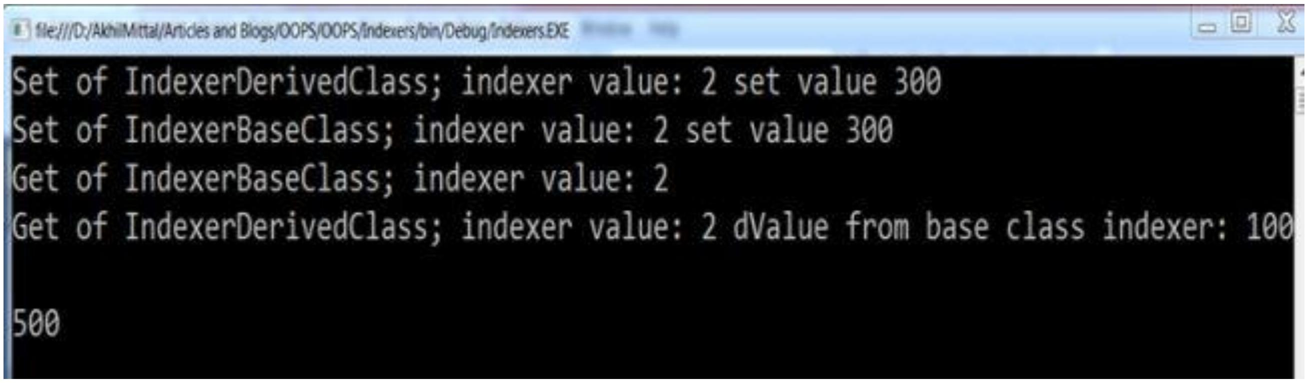
Indexer.cs

```
using System;
namespace Indexers
{
    public class IndexerBaseClass
    {
        public virtual int this[int indexerValue]
        {
            get
            {
                Console.WriteLine("Get of IndexerBaseClass; indexer value: " + indexerValue);
                return 100;
            }
            set
          {
                Console.WriteLine("Set of IndexerBaseClass; indexer value: " + indexerValue + " set value
 " + value);
            }
        }
    }
    public class IndexerDerivedClass:IndexerBaseClass
    {
        public override int this[int indexerValue]
        {
            get
            {
                int dValue = base[indexerValue];
                Console.WriteLine("Get of IndexerDerivedClass; indexer value: " + indexerValue + " dVal
ue from base class indexer: " + dValue);
                return 500;
            }
            set
            {
                Console.WriteLine("Set of IndexerDerivedClass; indexer value: " + indexerValue + " set va
lue " + value);
                base[indexerValue] = value;
            }

        }
    }
}
```

Program.cs

```
using System;
namespace Indexers
{
    class Program
    {
        static void Main(string[] args)
        {
            IndexerDerivedClass indexDerived=new IndexerDerivedClass();
            indexDerived[2] = 300;
            Console.WriteLine(indexDerived[2]);
            Console.ReadLine();

        }
    }
}
```

Output:



file:///D:/AkhilMittal/Articles and Blogs/OOPS/OOPS/Indexers/bin/Debug/Indexers.EXE

```
Set of IndexerDerivedClass; indexer value: 2 set value 300
Set of IndexerBaseClass; indexer value: 2 set value 300
Get of IndexerBaseClass; indexer value: 2
Get of IndexerDerivedClass; indexer value: 2 dValue from base class indexer: 100

500
```

The example code taken above explains run time polymorphism and inheritance in indexers. I created a base class named *IndexerBaseClass* having an indexer with its own get and set as was explained in prior examples. Thereafter a derived class is created named *IndexerDerivedClass*, this derives from IndexerBaseClass and overrides the "this" indexer from the base class. Note that the base class indexer is marked virtual, so we can override it in a derived class by marking it “override” in the derived class. The example makes a call to the indexer of the base class. Sometimes when we need to override code in a derived class in the derived class, we may require the base class indexer to be called first. This is just a situation. The same rule of run time polymorphism applies here, we declare the base class indexer and virtual and derived class one as override. In the “set” accessor of the derived class, we can call the base class indexer as *base[indexerValue]*. Also this value is used to initialize the derived class indexer as well. So the value is stored in the “value” keyword too. So, indexDerived[2] in the Main() method of Program.cs is replaced to base[2] in the “set” accessor. Whereas in the “get” accessor it is the reverse, we must put base[indexerValue] on the right hand side of the equal sign. The “get” accessor in the base class returns the value 100 that we get in the dValue variable.

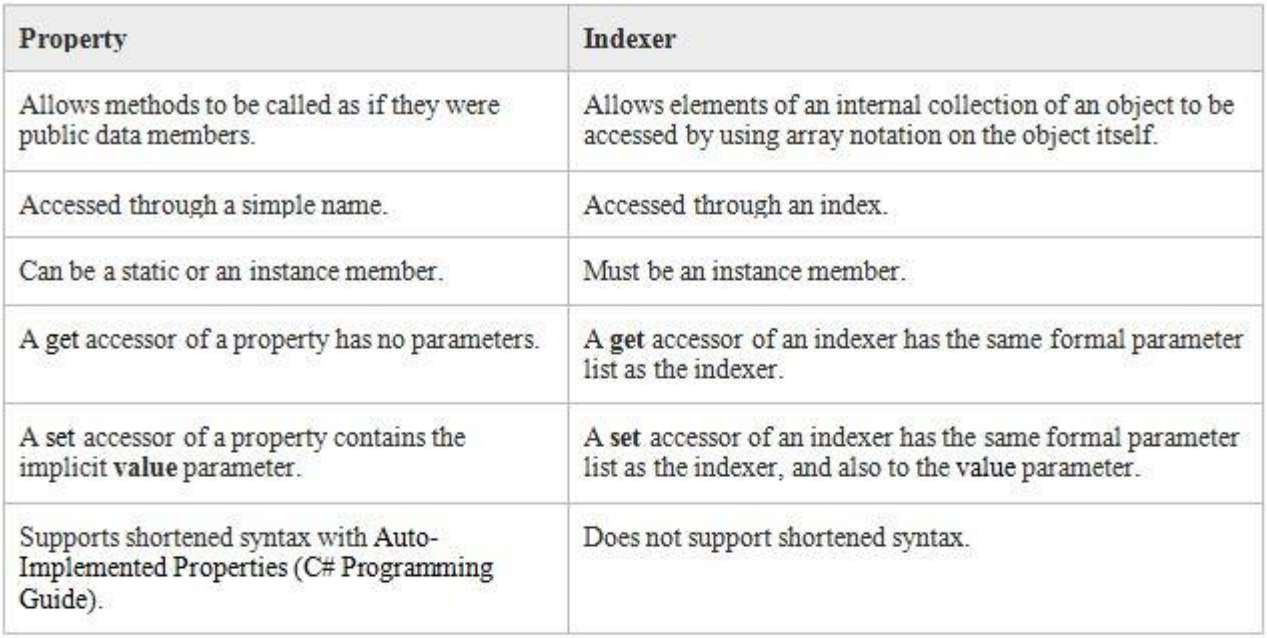
.NET Framework and Indexers

Indexers play a crucial role in the .NET Framework. Indexers are widely used in the .NET Framework builtin classes, libraries such as collections and enumerable. Indexers are used in collections that are searchable like Dictionary, Hashtable, List, Arraylist and so on.

Point to remember: Dictionary in C# largely uses indexers to have a staring parameter as an indexer argument.
Classes like ArrayList and List use indexers internally to provide functionality of arrays for fetching and using the elements.

Properties vs Indexers

I have already explained a lot about properties and indexers, to summarize, let me point to
an MSDN link for a better explanation.



Property	Indexer
Allows methods to be called as if they were public data members.	Allows elements of an internal collection of an object to be accessed by using array notation on the object itself.
Accessed through a simple name.	Accessed through an index.
Can be a static or an instance member.	Must be an instance member.
A get accessor of a property has no parameters.	A **get** accessor of an indexer has the same formal parameter list as the indexer.
A set accessor of a property contains the implicit **value** parameter.	A **set** accessor of an indexer has the same formal parameter list as the indexer, and also to the **value** parameter.
Supports shortened syntax with Auto-Implemented Properties (C# Programming Guide).	Does not support shortened syntax.

Table taken from MSDN.

Conclusion

With this article we completed nearly all the scenarios related to an indexer. We did many hands-on labs to clarify our concepts. I hope my readers now know by heart about these basic concepts and will never forget them. These may also help you in cracking C# interviews.