**Diving in OOP (Part 1) : Polymorphism and Inheritance(Early Binding/Compile Time Polymorphism).**

OOPS

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

OOP stands for “Object-Oriented Programming.” Remember, it’s OOP not OOPS,’S’ may stand for system, synopsis, structure etc. It is a programming approach 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. What are OOP Concepts?

Following are OOP concepts explained in brief, 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 method from the class without knowing about how this is created or what is the complexity behind it. In terms of a real world example, when we drive a bike and change the gears we don’t have to care about how internally its working, like how liver is pulled or how chain is set.
2. **Inheritance**: Inheritance is most popular Concept in OOP’s .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 have to write the logic again for derived class functions, we can use them as it is.
3. **Data Encapsulation:** Wrapping up of member data and member functions of a class in a single unit is called encapsulation. The visibility of the member functions,data members is set via access modifiers used in class.
4. **Polymorphism**: Poly means many and morphism means many function The Concepts Introduces in the form of Many behaviours of an object.
5. **Message Communication**: Message Communication means when an object passes the call to method of class for execution.

OK, we covered lots of theory, now it’s time for action. I hope that will be interesting. We’ll cover the topics in a series as follows,



1. [**Diving in OOP (Day 1): Polymorphism and Inheritance(Early Binding/Compile Time Polymorphism)**](https://codeteddy.com/2014/05/11/diving-in-oop-part-1-polymorphism-and-inheritanceearly-bindingcompile-time-polymorphism/)
2. [Diving in OOP (Day 2): Polymorphism and Inheritance (Inheritance)](https://codeteddy.com/2014/05/16/diving-in-oop-part-2-polymorphism-and-inheritance-inheritance/)
3. [Diving in OOP (Day 3): Polymorphism and Inheritance (Dynamic Binding/Run Time Polymorphism)](https://codeteddy.com/2014/05/18/diving-in-oop-day-3-polymorphism-and-inheritance-dynamic-bindingrun-time-polymorphism/)
4. [Diving in OOP (Day 4): Polymorphism and Inheritance (All about Abstarct classes in C#)](https://codeteddy.com/2014/06/05/diving-in-oop-day-4-polymorphism-and-inheritance-all-about-abstract-classes-in-c/)
5. [Diving in OOP (Day 5): All about access modifiers in C# (Public/Private/Protected/Internal/Sealed/Constants/Readonly Fields)](https://codeteddy.com/2014/07/11/diving-into-oop-day-5-all-about-c-access-modifiers-publicprivateprotectedinternalsealedconstantsstatic-and-readonly-fields/)
6. [Diving in OOP (Day 6): Understanding Enum in C# (A Practical Approach)](https://codeteddy.com/2014/09/11/diving-in-oop-day-6-understanding-enums-in-c-a-practical-approach/)
7. [Diving into OOP (Day 7): Properties in C# (A Practical Approach)](https://codeteddy.com/2015/07/07/diving-into-oop-day-7-properties-in-c-a-practical-approach/)
8. [Diving into OOP (Day 8): Indexers in C# (A Practical Approach)](https://codeteddy.com/2015/07/07/diving-into-oop-day-8-indexers-in-c-a-practical-approach/)
9. [Diving into OOP (Day 9): Understanding Events in C# (An Insight)](http://www.codeproject.com/Articles/1009930/Learning-Csharp-Day-Understanding-Events-in-Csharp)

3. Polymorphism:

In this article we will cover almost all the scenarios of compile type polymorphism, the use of **params** keyword in detail, and case study or hands on to different 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 DisplayOverloadhaving varied parameters as follows,

**Overload.cs**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/771455/Diving-in-OOP-Polymorphism-and-Inheritance-Part)

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 *Program.cs* file, add the following code,

**Program.cs**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/771455/Diving-in-OOP-Polymorphism-and-Inheritance-Part)

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 should differ. This feature of C# is called method overloading. Therefore, we need not to remember lots of 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 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 project using following code,

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/771455/Diving-in-OOP-Polymorphism-and-Inheritance-Part)

public void DisplayOverload() { }

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 comes,

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

If we run the project using following code,

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/771455/Diving-in-OOP-Polymorphism-and-Inheritance-Part)

static void DisplayOverload(int a) { }

public void DisplayOverload(int a) { }

public void DisplayOverload(string a){ }

We again get a compile time error,

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

Can you differentiate with the modification done in the above code, 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 same as modifiers such as static are also not considered to be a part of method signature.

**Point to remember:** Modifiers such as static are not considered as part of method signature.

If we run the program as per following code, considering the method signature is different now,

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/771455/Diving-in-OOP-Polymorphism-and-Inheritance-Part)

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,

**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 etc. Method DisplayOverload takes an int with different access modifiers i.e. out/ref etc, 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 can not have the same signature and also non-members cannot have the same name as members.

4. Role of Params Parameter in Polymorphism

A method can be called by four different types of parameters.

1. pass by value,
2. Pass by reference,
3. As an output parameter,
4. Using parameter arrays.

As explained earlier the parameter modifier is never the part of 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 following code,

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/771455/Diving-in-OOP-Polymorphism-and-Inheritance-Part)

public void DisplayOverload(int a, string a) { }

public void Display(int a)  
{  
string a;  
}

Results in compile time error,

**Error1: The parameter name ‘a’ is a duplicate**

**Error2: A local variable named ‘a’ cannot be declared in this scope because it would give a different meaning to ‘a’, which is already used in a ‘parent or current’ scope to denote something else**

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

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

When we run the following code,

**Overload.cs**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/771455/Diving-in-OOP-Polymorphism-and-Inheritance-Part)

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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/771455/Diving-in-OOP-Polymorphism-and-Inheritance-Part)

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 pass 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 as name is passed by reference. Variables x and name refer to the same string in memory. Changing one changes the other. Again changing y also changes name variable as 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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/771455/Diving-in-OOP-Polymorphism-and-Inheritance-Part)

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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/771455/Diving-in-OOP-Polymorphism-and-Inheritance-Part)

class Program

{

static void Main(string[] args)

{

Overload overload = new Overload();

overload.Display();

Console.ReadKey();

}

}

We get output,

**Output**

Akhil 100  
Mittal 100  
OOP 100  
Akhil 200

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

we can achieve it with the help ofparams 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 method DisplayOverload, the first argument has to be an integer, the rest can be from zero to an infinite number of strings.

If we add a method like ,

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/771455/Diving-in-OOP-Polymorphism-and-Inheritance-Part)

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

We get a compile time error as,

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

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

**Overload.cs**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/771455/Diving-in-OOP-Polymorphism-and-Inheritance-Part)

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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/771455/Diving-in-OOP-Polymorphism-and-Inheritance-Part)

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.

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/771455/Diving-in-OOP-Polymorphism-and-Inheritance-Part)

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

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

For the above 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:**same as error above.

The data type of the params argument must be a single dimensional array. Therefore [ ][ ]

is allowed but not [,]. We also not allowed to combine the params keyword with ref or out.

**Overload.cs**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/771455/Diving-in-OOP-Polymorphism-and-Inheritance-Part)

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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/771455/Diving-in-OOP-Polymorphism-and-Inheritance-Part)

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 pass a string array instead of individual strings as arguments. Here, names is a string array which has been initialized using the short form. Internally when we call the function DisplayOverload, C# converts the string array into individual strings.

**Overload.cs**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/771455/Diving-in-OOP-Polymorphism-and-Inheritance-Part)

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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/771455/Diving-in-OOP-Polymorphism-and-Inheritance-Part)

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 got two errors. 

For the above mentioned code, C# does not permit mix and match. We assumed that the last string “Ekta” would be added to the array of strings names or convert names to individual strings and then add the string “Ekta” 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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/771455/Diving-in-OOP-Polymorphism-and-Inheritance-Part)

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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/771455/Diving-in-OOP-Polymorphism-and-Inheritance-Part)

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, Hence proved.

**Overload.cs**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/771455/Diving-in-OOP-Polymorphism-and-Inheritance-Part)

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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/771455/Diving-in-OOP-Polymorphism-and-Inheritance-Part)

class Program

{

static void Main(string[] args)

{

Overload overload = new Overload();

overload.Display();

Console.ReadKey();

}

}

**Output**

102

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

**Overload.cs**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/771455/Diving-in-OOP-Polymorphism-and-Inheritance-Part)

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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/771455/Diving-in-OOP-Polymorphism-and-Inheritance-Part)

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’ll talk about method overloading. C# is extremely talented though partial. It does not appreciate the paramsstatement and treats it as a stepchild. When we invoke DisplayOverload only with one integer, C# can only call theDisplayOverload that takes a params as a parameter as 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 theparams DisplayOverload or DisplayOverload with the two ints. As discussed 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 a bit tricky example, yet important,

**Overload.cs**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/771455/Diving-in-OOP-Polymorphism-and-Inheritance-Part)

public class Overload

{

public static void Display(params object[] objectParamArray)

{

foreach (object obj in objectParamArray)

{

Console.Write(obj.GetType().FullName + " ");

}

Console.WriteLine();

}  
}

**Program.cs**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/771455/Diving-in-OOP-Polymorphism-and-Inheritance-Part)

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 passing the method Display an array of object that looks like object. Since all the classes are derived from a common base class object, we can do that. The method Display gets an array of objectsobjectParamArray. In the foreach object class has a method named GetType that returns an object that looks like Type, which too has a method named FullName that returns the name of the type. Since three different types displayed. In the second method call of Display we are casting objArray to an object. Since there is no conversion available from converting an object to an object array i.e. object [ ], so 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,

**Overload.cs**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/771455/Diving-in-OOP-Polymorphism-and-Inheritance-Part)

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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/771455/Diving-in-OOP-Polymorphism-and-Inheritance-Part)

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

In this article of our Diving in OOP series we learnt 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 **params** keyword and its use in polymorphism.

To sum up lets list down all the point to remembers once more,



1. C# recognizes the method by its parameters and not 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 same. So this is not polymorphism.
3. Modifiers such as static are not considered as part of 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 can not have the same signature and also non-members cannot have the same name as members.
5. Parameter names should be unique. And also we can not have a parameter name and a declared variable name in the same function as same.
6. In case of pass by value, the value of the variable is passed and in the case of ref and out, the address of the reference is passed.
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. Parameter array must be a single dimensional array.

**Diving in OOP (Part 2) : Polymorphism and Inheritance (Inheritance)**

1. **Example of Inheritance**

class ClassA: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

ClassA is now said to have been derived from ClassB. What that means is all the code we wrote in ClassBcan now be accessed and used in ClassA.

1. **Let’s take another scenario. Suppose we get into a situation where ClassA also has a method of same name as of inClassB. 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 *Program.cs* class:

class Program

{

static void Main(string[] args)

{

ClassA a = new ClassA();

a.Display1();

Console.ReadKey();

}

}

We get 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 to have a method with the same name already declared in its base class.

So, ClassA undoubtedly can contain 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.

1. **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 methods 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 base class.

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

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

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

The keyword base can be used in any of the derived class. It means call the method of the base class. Thus base.Display1 will call the methodDisplay1 fromClassB the base class

of ClassA as defined earlier.

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

4. **base keyword can only be used in derived classes?**

class ClassB

{

public int x = 100;

public void Display1()

{

Console.WriteLine("ClassB Display1");

}

}

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

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, i.e., ClassB is base class. Therefore, class ClassA can use all the members of class ClassB. Inheritance does not have backwards compatibility, whatever members ClassA contains do not permeate upwards to ClassB. When we tried to access Display2 method of classA from the instance of class ClassB, it cannot give 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, i.e., simply not visible in the derived class.

A class member could be of two types, i.e. either a static member that directly belongs to a class or an instance member that is accessed through instance of that class and belongs to that particular instance only. 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 have to make them static by using 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, 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.

1. **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 derive from a built in class in C#, let’s run the code.

We get so many compile time errors.

**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'

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, etc.

1. **One more case,**

public class ClassW

{

}

public class ClassX  
{  
}

public class ClassY : ClassW, ClassX  
{  
}

In the above mentioned case, we see three classes, ClassW, ClassX and ClassY. ClassY is derived from ClassWand 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 about interfaces in this article. We are not allowed to derive from more than one class, thus every class can have only one base class.

1. **Suppose we try to write code as below:**

public class ClassW:ClassY

{

}

public class ClassX:ClassW  
{  
}

public class ClassY : ClassX  
{  
}

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, our code should build successfully. Let’s compile the code. What do we get? Again a compile time error.

**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 which was derived from ClassY and ClassY was again derived from ClassX, which caused circular dependency in three classes, that is logically impossible.

1. **Equalizing the Instances/Objects**

Let’s directly start with a real case:

ClassB:

public class ClassB

{

public int b = 100;

}

public class ClassA  
{  
public int a = 100;  
}

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 two different classes. Let’s compile the code,

We get compile time error:

**Error**

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

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

9.

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 object, i.e., derived from Object class (i.e. 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 object class.

public class ClassB

{

public int b = 100;

}

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

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

Now compile the code, we get:

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

C# is very particular about objects equating. **Thus when we write classA = classB, classA**

**looks like ClassA, ClassB and object and as a looks like ClassB, there is a match at ClassB. Result is No error** . 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, because the **class ClassB is less/smaller than the class ClassA.** The class ClassA has ClassB and more. We cannot have a larger class on the right and a smaller 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 only and only equate the dissimilar objects if they are derived from each other. We can equate an object of a base class to a derived class but not vice versa.

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 be a classA, i.e., a ClassA instance. But it finds classB, i.e., a ClassB instance. So when we apply the cast, we actually try to convert instance of ClassB to instance of ClassA. This approach satisfies the rules of C# on only equating similar objects type. 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 the following code, and try to typecast classA toClassB object.

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.

10.

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 char, but char can be converted to int.

**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 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, System.Array, etc.
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 which was derived from ClassY and ClassY was again derived from ClassX, which caused circular dependency in three classes, that is logically impossible.
9. We can only and only equate the dissimilar objects if they are derived from each other. We can equate an object of a base class to a derived class but not vice versa.
10. We cannot implicitly convert an int to char, but char can be converted to int.

**Diving in OOP (Day 3): Polymorphism and Inheritance (Dynamic Binding/Run Time Polymorphism)**

**Run time polymorphism also called late binding or method overriding**

1. What are New and Override Keywords in C#?

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

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/774578/Diving-in-OOP-Day-Polymorphism-and-Inheritance-Dyn)

/// *<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();  
}  
}

Output:

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

Warnings:

'InheritanceAndPolymorphism.ClassA.AAA()' hides inherited member

'InheritanceAndPolymorphism.ClassB.AAA()'. Use the new keyword if hiding was intended.

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

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

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

Class ClassB is the super class of class ClassA. That means ClassA is the derived class and ClassB is the base class. The class ClassA comprises ClassB and something more. So we can conclude that object of ClassA is bigger than object of ClassB. Since ClassA is inherited from ClassB, it contains its own methods and properties. Moreover, it will also contain methods/properties that are inherited from ClassB too.

Let’s take the case of object y. It looks like ClassB and initialized by creating an object that also looks like ClassB, well and good. Now, when we call the methods AAA and BBB and CCC through the object y, we know that it will call them from ClassB.

Object x looks like that of ClassA, i.e., the derived class. It is initialized to an object that looks like ClassA. When we call AAA, BBB and CCC method through x, it calls AAA, BBB and CCC from ClassA.

Now there is 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 which does not give an error as explained earlier. But there is no change at all in the output we get and the behavior is identical to that of object y.**Therefore initializing it to an object that looks like ClassB or ClassA does not seem to matter**.

2.

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

Our code is as follows:

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”);  
}  
}

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

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/774578/Diving-in-OOP-Day-Polymorphism-and-Inheritance-Dyn)

/// <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:

Error: 'InheritanceAndPolymorphism.ClassA.AAA()': cannot override inherited member

'InheritanceAndPolymorphism.ClassB.AAA()' because it is not marked virtual, abstract, or override

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

**Marked all the methods of base class as virtual.**

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”);  
}  
}

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

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/774578/Diving-in-OOP-Day-Polymorphism-and-Inheritance-Dyn)

ClassB AAA

ClassB BBB

ClassB CCC

ClassA AAA

ClassA BBB

ClassA CCC

ClassA AAA

ClassB BBB

ClassB CCC

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

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

We wanted to override the AAA of the base class ClassB. We are infact telling C# that this AAA is similar to the AAAof the one in base class.

New keyword acts in the exact opposite to override keyword. The method BBB as we see has the new modifier.z.BBB() calls BBB from ClassB and not ClassA. New means that the method BBB is a new method and it has absolutely nothing to do with the BBB in the base class. It may have the same name i.e. BBB as in the base class, but that is only a coincidence. As z looks like ClassB, the BBB of ClassB gets called even though there is a BBB inClassA. 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 avirtual method. Virtual means that the base class is granting us permission to invoke the method from the derived class and not the base class. But, we have to add the modifier override if our derived class method has to be called.

1. 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:

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/774578/Diving-in-OOP-Day-Polymorphism-and-Inheritance-Dyn)

/// *<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

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/774578/Diving-in-OOP-Day-Polymorphism-and-Inheritance-Dyn)

ClassB AAA

ClassB BBB

ClassA CCC

ClassB AAA

ClassB BBB

ClassA CCC

ClassC AAA

ClassA BBB

ClassA CCC

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

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

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

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

The object y that looks like of ClassB but is initialized here to the derived class, i.e., ClassA.

y.AAA() first looks into the class ClassB. Here, it verifies whether the method AAA is marked virtual. The answer is an emphatic no and hence everything comes to halt and the method AAA gets called from class ClassB.

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



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

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

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

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/774578/Diving-in-OOP-Day-Polymorphism-and-Inheritance-Dyn)

public override void CCC()

{

Console.WriteLine("ClassC CCC");

}

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

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

1. **One more example:**

One more example:

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/774578/Diving-in-OOP-Day-Polymorphism-and-Inheritance-Dyn)

internal class A

{

public virtual void X()

{

}

}

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

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

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

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/774578/Diving-in-OOP-Day-Polymorphism-and-Inheritance-Dyn)

Error: 'InheritanceAndPolymorphism.C.X()': cannot override inherited member

'InheritanceAndPolymorphism.B.X()' because it is not marked virtual, abstract, or override

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

1. 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

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/774578/Diving-in-OOP-Day-Polymorphism-and-Inheritance-Dyn)

Class: A ; Method X

Class: C ; Method X

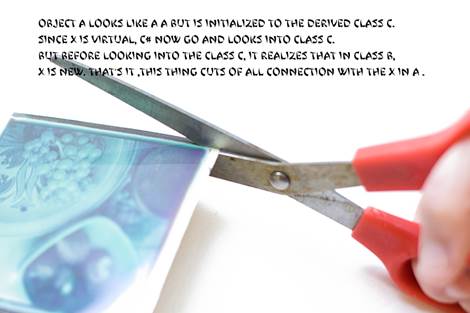
If in the above code, we remove the modifier override from X() in class C, we get:

Output

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/774578/Diving-in-OOP-Day-Polymorphism-and-Inheritance-Dyn)

Class: A ; Method X

Class: B ; Method X



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

In the second case, object b looks like class B now but in turn is initialized to an object of class C. C# first looks at classB. Here X is new and virtual both, which makes it a unique method X. Sadly, the X in C has the override modifier which sees to it that the X of C hides the X of B. This calls the X of C instead. If we remove the override modifier from X in class C, the default will be new, that cuts off the relation from the X of B. Thus, as it is, a new method, the Xof B gets invoked.

A virtual method cannot be marked by the modifiers static, abstract or override. A non virtual method is said to be invariant. This means that the same method gets called always, irrespective of whether one exists in the base class or derived class. In a virtual method, the run-time type of the object decides on which method to be called and not the compile-time type as is in the case of non virtual methods. For a virtual method, there exists a most derived implementation which always gets called.

1. Runtime Polymorphism with Four Classes

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

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/774578/Diving-in-OOP-Day-Polymorphism-and-Inheritance-Dyn)

ClassB XXX

ClassB XXX

ClassD XXX

ClassD XXX

Explanation

One last explanation of virtual and override will be a bit complex.

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

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

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/774578/Diving-in-OOP-Day-Polymorphism-and-Inheritance-Dyn)

/// *<summary>*

/// *Class C*

/// *</summary>*

public class ClassC : ClassB

{

public override void XXX()

{

Console.WriteLine("ClassC XXX");

}

}

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

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

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

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

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

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/774578/Diving-in-OOP-Day-Polymorphism-and-Inheritance-Dyn)

ClassA XXX

ClassB XXX

ClassA XXX

ClassB XXX

ClassC XXX

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

1. The Infinite Loop

public class ClassA

{

public virtual void XXX()

{

Console.WriteLine("ClassA XXX");

}

}

public class ClassB:ClassA  
{  
public override void XXX()  
{  
((ClassA)this).XXX();  
Console.WriteLine(“ClassB XXX”);  
}  
}

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 we got in the big article.

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

**Diving in OOP (Day 4): Polymorphism and Inheritance (All About Abstract Classes in C#)**

1. Abstract Classes

*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.*

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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/781765/Diving-in-OOP-Day-Polymorphism-and-Inheritance-All)

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 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 new keyword.

1. Non Abstract Method Definition in Abstract Class

Let’s add some code to our abstract class:

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/781765/Diving-in-OOP-Day-Polymorphism-and-Inheritance-All)

/// *<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 that we cannot use new if we have already used an abstract modifier.

1. Abstract Class Acting as a Base Class

Let’s add one more class now:

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/781765/Diving-in-OOP-Day-Polymorphism-and-Inheritance-All)

/// *<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 abstract class. Creating an object of ClassB does not gives us any 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.

1. Non Abstract Method Declaration in Abstract Class

Another scenario:

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/781765/Diving-in-OOP-Day-Polymorphism-and-Inheritance-All)

/// *<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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/781765/Diving-in-OOP-Day-Polymorphism-and-Inheritance-All)

Compile time error: 'InheritanceAndPolymorphism.ClassA.YYY()'

must declare a body because it is not marked abstract, extern, or partial

In the above code, 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 abstractclass must also be declared abstract as per the rules of C#.

1. Abstract Method Declaration in Abstract Class

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

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/781765/Diving-in-OOP-Day-Polymorphism-and-Inheritance-All)

/// *<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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/781765/Diving-in-OOP-Day-Polymorphism-and-Inheritance-All)

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

1. Abstract Method Implementation in Derived Class

Now provide a body of method YYY() in ClassB. Let’s see what happens:

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/781765/Diving-in-OOP-Day-Polymorphism-and-Inheritance-All)

/// *<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();  
}  
}

Everything seems fine now, but no? Compile the code, what we get:

**Output**

Two compile time errors this time:

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/781765/Diving-in-OOP-Day-Polymorphism-and-Inheritance-All)

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 no success till now. The compiler error indicates clearly that both of our base and derived class contains the same method named YYY().

If both our derived class and base class contain the method with the same name, always an error occurs. The only way to overcome this error is derived class explicitly add the modifier override to its method signature. We have already discussed such scenarios in our previous parts of the articles of Diving in OOP series.

Let’s add the override keyword before derived class method YYY().

/// *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?

Abstract Method Implementation in Derived Class with Different Return Type

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

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/781765/Diving-in-OOP-Day-Polymorphism-and-Inheritance-All)

/// *<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 return type of method YYY from void to int in derived class. Compile the code.

**Output**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/781765/Diving-in-OOP-Day-Polymorphism-and-Inheritance-All)

Compile time error: 'InheritanceAndPolymorphism.ClassB.YYY()': return type must be 'void'

to match overridden member 'InheritanceAndPolymorphism.ClassA.YYY()'

Therefore one more constraint.

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

Let’s see the implementation of the second line mentioned in “point to remember”,

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/781765/Diving-in-OOP-Day-Polymorphism-and-Inheritance-All)

/// *<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 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();  
}  
}

Compiler error:

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/781765/Diving-in-OOP-Day-Polymorphism-and-Inheritance-All)

'InheritanceAndPolymorphism.ClassB' does not implement

inherited abstract member 'InheritanceAndPolymorphism.ClassA.YYY3()'

‘InheritanceAndPolymorphism.ClassB’ does not implement inherited  
abstract member ‘InheritanceAndPolymorphism.ClassA.YYY2()’

‘InheritanceAndPolymorphism.ClassB’ does not implement inherited  
abstract member ‘InheritanceAndPolymorphism.ClassA.YYY1()’

If we implement these three methods in derived class, we’ll get no error.

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

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 which is what the compiler kept warning us about. We can initialize int variable a of ClassA to any value we wish. The variables in abstractclass act similar to that in any other normal class.

Whenever a class remains incomplete, i.e., we do not have the code for some methods, we mark those methodsabstract 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 have to implement the abstract, i.e., our incomplete methods from 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.

Abstract Method in Non Abstract Class

Let’s take another code block:

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/781765/Diving-in-OOP-Day-Polymorphism-and-Inheritance-All)

/// *<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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/781765/Diving-in-OOP-Day-Polymorphism-and-Inheritance-All)

Compiler error: 'InheritanceAndPolymorphism.ClassA.YYY()' is abstract

but it is contained in non-abstract class 'InheritanceAndPolymorphism.ClassA'

We just removed 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 have to be abstract as well.

**Point to remember**: If a class has even a single abstract method, then the class has to 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 abstract class has to give implementation to its abstract method. By default, the modifier new gets added to the derived class method, that makes it a new/different method.

Abstract Base Method

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/781765/Diving-in-OOP-Day-Polymorphism-and-Inheritance-All)

/// *<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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/781765/Diving-in-OOP-Day-Polymorphism-and-Inheritance-All)

Compile time error : Cannot call an abstract base member:

'InheritanceAndPolymorphism.ClassA.YYY()'

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

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:

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/781765/Diving-in-OOP-Day-Polymorphism-and-Inheritance-All)

/// *<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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/781765/Diving-in-OOP-Day-Polymorphism-and-Inheritance-All)

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 so it has to 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 base class. But we’ll do something different and tricky,

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

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

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

In case of classA.XXX() will definitely first look into the class ClassA. Here, it finds the method XXX marked as virtual. These kind of scenarios we have already taken n number of times in our earlier articles where we discussed about run time polymorphism . C# will then crawl over to class ClassB. Here it gets shocked that the method XXX()is abstract, i.e., 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 gets executed.

In the case of b.XXX()(), since the method is new, the links to the base class gets 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 abstract class ClassB.

Let’s replace the override modifier in ClassC with “new” like:

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/781765/Diving-in-OOP-Day-Polymorphism-and-Inheritance-All)

public class ClassC:ClassB

{

public new void XXX()

{

System.Console.WriteLine("ClassC XXX");

}

}

**Output**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/781765/Diving-in-OOP-Day-Polymorphism-and-Inheritance-All)

Compile time error: 'InheritanceAndPolymorphism.ClassC' does not implement

inherited abstract member 'InheritanceAndPolymorphism.ClassB.XXX()'

The error indicates that as 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.

Also there is one more point to remember.

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

Can Abstract Class be Sealed?

Let’s take this final question into our consideration. Let’s test this too with an example.

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/781765/Diving-in-OOP-Day-Polymorphism-and-Inheritance-All)

/// *<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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/781765/Diving-in-OOP-Day-Polymorphism-and-Inheritance-All)

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.

1. We cannot create an object of abstract class using new keyword.
2. A class can be derived from an abstract class.
3. 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 passed to it or the return type irrespective of the number of methods declared as abstract in 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 has to 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. Abstract class cannot be sealed class.
11. Abstract class cannot be a static class.

**Diving into OOP (Day 5): All About C# Access Modifiers (Public/Private/Protected/Internal/Sealed/Constants/Static and Readonly Fields)**

“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.”

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.

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 shown in the above code snippet. We added a class Modifiers and two staticmethods AAA and BBB. Method BBB is marked as public. We call the method BBB from 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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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 which 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:

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

class Program

{

static void Main(string[] args)

{

Modifiers.AAA();

Console.ReadKey();

}

}

**Output**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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

So , since methodAAA is private, therefore no one else can have access to it except Modifiers class.

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

Modifiers

Now mark the AAA method as protected, our class looks like:

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

class Modifiers

{

protected static void AAA()

{

Console.WriteLine("Modifiers AAA");

}

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

**Program**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

class Program

{

static void Main(string[] args)

{

Modifiers.AAA();

Console.ReadKey();

}

}

**Output**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

'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 namedprotected. But BBB can access 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 this:

Modifiers Base Class

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

class ModifiersDerived:ModifiersBase

{

public static void XXX()

{

AAA();

BBB();

CCC();

}

}

**Program Class**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

class Program

{

static void Main(string[] args)

{

ModifiersDerived.XXX();

Console.ReadKey();

}

}

**Output**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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

Now in this case, we are dealing with 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 method XXX, we can call CCC because it is marked protected, but it cannot be called from anywhere else including 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 another 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:

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

AccessModifiersLibrary.ClassA:

namespace AccessModifiersLibrary  
{  
internal class ClassA  
{  
}  
}

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

Now in your console application, “AccessModifiers” i.e. created earlier. Add the reference ofAccessModifiersLibrary library by adding its compiled DLL as a reference to AccessModifiers.

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

**AccessModifiers.Program**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

using AccessModifiersLibrary;

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

And compile the code.

**Output**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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. Internal modifier means that access is limited to 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? i.e.,

**AccessModifiersLibrary.ClassA**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

namespace AccessModifiersLibrary

{

class ClassA

{

}

}

**AccessModifiers.Program**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

using AccessModifiersLibrary;

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

**Output**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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 results remain the same.

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

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

Namespaces with Modifiers

Let’s for fun, mark a namespace of AccessModifiers class library as public in Program class:

**Program**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

public namespace AccessModifiers

{

class Program

{

static void Main(string[] args)

{

}  
}  
}

Compile the application.

**Output**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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 defaultpublic 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:

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

namespace AccessModifiers

{

private class Program

{

static void Main(string[] args)

{

}  
}  
}

Compile the code.

**Output**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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

**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 above explained access modifiers, but default modifier is private.

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

we can’t mark a member with more than one access modifier often. But there are such scenarios too, we’ll cover them in 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:

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

namespace AccessModifiersLibrary

{

internal class ClassA

{

public void MethodClassA(){}

}

}

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

**Program**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

using AccessModifiersLibrary;

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

**Output**

Compile time errors:

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

'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 Program class due to protection level of ClassA, i.e. 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:

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

AccessModifiersLibrary.ClassA:

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

**Program**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

using AccessModifiersLibrary;

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

**Output on compilation**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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

Public Class and Internal Method

Make ClassA as public and MethodClassA as internal:

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

AccessModifiersLibrary.ClassA:

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

**Program**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

using AccessModifiersLibrary;

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

**Output on compilation**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

'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:

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

using AccessModifiersLibrary;

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

**Compiler output**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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, i.e., 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 console application, place the following code:

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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 is containing a protected member, i.e., a. But to the same class, no modifiers make sense. However as a isprotected, in the derived class method MethodBBB, we cannot access it through AAA as 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 inMethodBBB (). This means that we cannot access the protected members from an object of the base class, but from the objects of derived class only. This is in spite of the fact that a is a member of AAA i.e. the base class. Even so, we still cannot access it. Also we cannot access a from the method Main.

Accessibility Priority in Inheritance

**Program**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

namespace AccessModifiers

{

class AAA

{

}  
public class BBB:AAA  
{

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

**Compiler Output**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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**: In 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 as the derived class BBB has to have an access modifier which 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 i.e. ClassA marked as public and ClassB internal or default, we get rid of the error.

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

Another scenario:

**Program**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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

Here the accessibility of AAA is internal which is more restrictive than public. The accessibility of method MethodBis public which 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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

namespace AccessModifiers

{

class AAA

{

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

**Compiler Output**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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

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

Change the code to:

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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.

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 internalcombination.

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, which are not nested in other types, can only have internal or public accessibility. The default accessibility for these types is internal.

Nested types, which are members of other types, can have declared accessibilities as indicated in the following table.”

**Reference**: <http://msdn.microsoft.com/en-us/library/ba0a1yw2.aspx>

| **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 thatsealed class. In other words, a class marked as sealed can’t act as a base class to any other 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, properties like a normal class do.

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

**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.

Lab2

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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 anotherconst variable z, whose value is set to 300. Thus C# first evaluates z to 300 then y becomes 290 i.e. z -1 and finally x takes on the value of y i.e. 290 + 100 resulting in 390.

Lab3

**Program**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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 into error. The value of const x depends upon y, and yin turn depends upon value of z, but we see value z depends upon x as x is assigned directly to z, it results in a circular dependency.

**Point to remember**: Like classes, const variables cannot be circular, i.e., they cannot depend on each other.

Lab4

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

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

using System;

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

public class ClassA  
{

}  
}

**Output**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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 Program class:

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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

**Point to remember**: One can only initialize a const variable to a compile time value, i.e., a value available to the compiler while it is executing.

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

Lab5

**ClassA**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

public class ClassA

{

public const int aaa = 10;

}

**Program**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

public class Program

{

public static void Main(string[] args)

{

ClassA classA=new ClassA();

Console.WriteLine(classA.aaa);

Console.ReadKey();

}

}

**Output**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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, i.e., a name to reference a const. A const has to be static as no one will be allowed to make any changes to a const variable.

Just mark the const as static.

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

Compile time error: The constant 'AccessModifiers.ClassA.aaa' cannot be marked static

C# tells us frankly that a field i.e. already static by default cannot be marked as static.

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

Lab6

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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 constvariable 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.

Lab1

**Program**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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.

Lab2

**Program**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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, i.e. instance variables and then initialize each of them to their default values as well.

Lab3

**Program**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

10

15

Output is self explanatory. C# always initializes static variables to their initial value after creating them. Variables xand 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 as the variable y has a value of 0, so x will get a value of 10. Then at the next line, y is the value of x + 5. The variable xhas a value of 10 and so y now becomes 15. As C# does not see both lines at the same time, it does not notice the circularity of the above definition.

Lab4

**Program**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

using System;

namespace AccessModifiers  
{  
public class Program  
{  
int x = y + 10;  
int y = x + 5;  
public static void Main(string[] args)  
{

}  
}  
}

**Output**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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 Lab3 does not work for instance variables as the rules of an instance variable are quite different than that of static variables. The initializer of an instance variable has to 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

Lab1

**Program**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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.

Lab2

**Program**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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).

Lab3

**Program**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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 have to be initialized at the time of creation.

Lab4

**Program**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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 have seen in the above mentioned example.

Lab5

**Program**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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 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 consts and readonly fields to the actual values.

Lab6

**Program**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

using System;

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

**Output**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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, which comes first. Now here the readonly modifier precedes the data type int, we already discussed at the very start of the article. This is just a rule that must always be remembered.

Lab7

**Program**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

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**

https://i2.wp.com/www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/792326/Diving-into-OOP-Day-All-About-Csharp-Access-Modifi)

Compile time error:

A readonly field cannot be passed 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 which 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 have to remember:

1. The default access modifier is private for class members.
2. A class marked as internal can 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 isinternal 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, i.e., 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, i.e., 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, i.e., a name to reference a const. Aconst has to be static as 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).

**Diving in OOP (Day 6): Understanding Enums in C# (A Practical Approach)**

*“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.”*

Now 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 enum is called as static variables, so an enum can be considered here as static objects. Thereforeother enums in the above example can be declared in the same way as Yellow, like Blue can be declared asColor.Blue. The output in the above two examples we see is 0 when we typecast 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 can we infer that an enumrepresents a constant number, therefore an enum type is a distinct type having named constants.

**Point to remember**: An enum represents for 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 defaultdatatype 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 andsbyte.

**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 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 of mentioned in the error.

**Point to remember**: 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 enum, call it class Derived, so our code.

**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’

Image credit: <https://www.flickr.com/photos/lwr/931211869/>

**Point to remember**: By default, enum is a sealed class and therefore sticks to all the rules that a sealed class follows, so no class can derive from enum, i.e., a sealed type.

Can System.Enum be a base class toenum?

**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,enum is also derived from 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 large number of enums defined and we want to compare the values of enum to each other to check if they are smaller, larger or equal value to one another.

Since all enums implicitly derive from Enum class that implements the interface IComparable, they all have a methodCompareTo(), that we just used in the above example. The method being nonstatic has to be used through a member. Yellow has value 0, Blue as 1 and Green as 2. In the first statement, when Color.Yellow compared toColor.Blue, value of Yellow is smaller than Blue hence -1 returned, same applied for the second statement when Color.Blue compared to Color.Green. Green has larger value, i.e., 2 than that of Color.Blue having value 1 only. In the third statement, i.e., vice versa of first statement, we get the result of camparisonas 1,becauseBlue is larger than Yellow. In the last statement where Color.Green compares to itself, we undoubtedly get the value 0.

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

Another comparison example is shown below:

**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 Enum type and resulted in a compile time error. Sinceenum acts as an individual data type so it cannot be directly compared to an int, however, we can typecast the enumtype to int to perform comparison, like in the below 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 IFormatter interface. It’s a static method so can be used directly with theenum class defined as Color. It’sfirst parameter is the type of the enum class, second is the member that has to be formatted and third is the format, i.e., 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, i.e., instance of type as a parameter and in return gives an array of strings. Like in the above example, we had array of 3 members in our enum, therefore their names are displayed one by one.

Another example is as follows:

**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 staring type and got an output too, so, numerous predefined conversion methods can be used to convert enum from one data type to another.

**Point to remember**: Numerous predefined conversion methods can be used to convert 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 above 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, the first member takes the value 0 and increments it by one to succeeding members.

Let’s take 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

Surprised! We can always specify the default constant value to any enum member, here we see, we specified value 2to yellow, so as per 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 toBrown, 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 ourenum from byte, we know we can do that? We then changed the value of yellow from 2 to300, and we resulted in a compile time error. Since here our underlying data type was byte, so it is as simple as that, that we cannot specify the value to enum members which exceeds the range of underlying data types. The value 300is beyond the range of byte. It is similar to assigning the beyond range value to a byte data type variable.

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 any error, so we see, more than one enum members can be initialized a same constant value.

**Point to remember**: More than one enum members can be initialized a 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 above example, we tried to initialize the enum member out of the scope of defined enum, i.e., in another class, and got a compile time error. We must not forget that an enum acts as a constant, which 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 have 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 checkingcolor on the basis of passed parameter to the method, i.e., Enum Color. In Main method, we try to access thatCheckColor method, passing various parameters. We see that the switch statement in CheckColor method can take any of the datatype passed and in return case statements use name of that type and not the plain intnumber to compare the result. We see that this made our program more readable. So enum plays an important role in making the program more readable and structured, 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 valueBlue 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 error, C# is smart enough to catch these kind of tricks.

Diving Deep

Let’s take some complex scenarios:

**Lab1**

**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 above 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, as C# knows that “-” is also a unary operator, that while using above code may create a confusion for compiler that we are using subtraction or typecasting a negative value. So always use parenthesis while typecasting negative values.

**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 reserved member for the enumerator. C# compiler like this keyword has large number of reserved inbuilt keywords.



Image credit: [www.vector.rs](http://www.vector.rs/)

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 have 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, enum is a sealed class and therefore sticks to all the rules that a sealed class follows, so no class can derive from enum, i.e., a sealed type.
5. The enum type is implicitly derived from System.Enum and so we cannot explicitly derive it fromSystem.Enum.
6. enum is also derived from three interfaces IComparable, IFormattable and IConvertible.
7. Numerous predefined conversion methods can be used to convert enum from one data type to another.
8. More than one enum members 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.

**Diving into OOP (Day 7): Properties in C# (A Practical Approach)**

*“*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.*“*

If we analyze properties internals, they are very different from normal variables; properties are internally methods that do not have their own memory like variables have. We can leverage property to write our custom code whenever we access a property. We can access the code when we call/execute properties or at the time 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.

A property must have an accessor, i.e. 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.

Get accessor

**Properties.cs**

Hide   Copy Code

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**

Hide   Copy Code

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**



It returns the correct age subjective to date of birth provided. Did you notince something here? Our property contains some code and logic to calculate age, but the caller i.e. *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

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

**Properties.cs**

Hide   Copy Code

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**

Hide   Copy Code

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**



In the above given example, I made few minor changes in get accessor, i.e. just printing that control is in “Get accessor” and introduced a “Set” in Age property too. Everything else remains same. Now when I call Name property, it works as was working earlier.Since we used “Set” so now we are allowed to set the value of a property. When I do properties.Age = 40; that means I am setting value 40 to that property. We can say a property can also be used to assign some value.In this case 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 gets called which returns value with some custom logic. We have a drawback here, as we see, whenever we call get we get the same vale and not the value that we assigned to that property i.e. because get has its custom fixed logic. Let’s try to overcome this situation.

**Properties.cs**

Hide   Shrink https://i0.wp.com/www.codeproject.com/images/arrow-up-16.png   Copy Code

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**

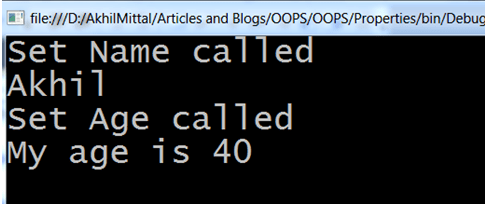
Hide   Copy Code

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**



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

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

**[Point to remember](https://www.blogger.com/null)**

The variable used for property should be of same data type as the data type of the property.

In our case we used variables, name and age, they share same datatype as their respective properties do. We don’t use variables as there might be scenarios in which we do not have control over those variables, end user can change them at any point of code without maintaining the change stack. Moreover one major use of properties is user can 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 variable.

When using Automatic Properties, they do this internally, i.e. we don’t have to define an extra variable to do so,like shown below,

Lab3

**Properties.cs**

Hide   Copy Code

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**

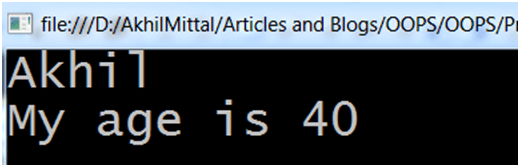
Hide   Copy Code

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**



Here

Hide   Copy Code

public string Name { get; set; }

public int Age { get; set; }

are automatic properties.

I hope now you know 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 class.

**Properties.cs**

Hide   Copy Code

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**

Hide   Copy Code

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

Hide   Copy Code

properties.Name = "Akhil";

properties.Age = 40;

But since they were marked read-only i.e. only with get accessor, we encountered a compile time error.

Write-Only

A property can also be made write-only i.e. vice versa to 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 get accessor in it.

**Properties.cs**

Hide   Copy Code

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**

Hide   Copy Code

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 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 above mentioned example, our property is marked only with set accessor, but we tried to access those properties in our main program with,

Hide   Copy Code

Console.WriteLine(properties.Name);

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

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

Insight of Properties in C#

Lab1

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

**Properties.cs**

Hide   Copy Code

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 compile time error,

Error The type ‘Properties.Properties’ already contains a definition for ‘Name’

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

Lab2

Can we define properties same as an already defined variable? The answer is NO.

**Properties.cs**

Hide   Copy Code

using System;

namespace Properties

{

public class Properties

{

private string name;

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

Build the project; we get 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 same common name with the same case because at the time of accessing them, compiler may get confused that whether you are trying to access a property or a variable.

Properties vs Variables

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

To summarize, [MSDN](https://msdn.microsoft.com/en-us/library/sk5e8eth(VS.80).aspx) explains this theory better than me,

| **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 moduleProperty’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**

Hide   Copy Code

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;

}

}

}

}

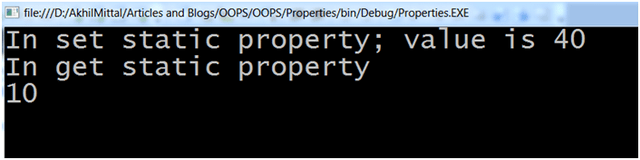
**Program.cs**

Hide   Copy Code

using System;

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

**Output**



In above 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 it is variable or a method. They’ll be accessed via class name only.

Properties return type

Lab1

**Properties.cs**

Hide   Copy Code

using System;

namespace Properties  
{  
public class Properties  
{  
public void AbsProperty  
{  
get  
{  
Console.WriteLine(“Get called”);  
}  
}  
}  
}

Compile the program.

Output is a compile time error,

Error ‘AbsProperty’: property or indexer cannot have void type

**[Point to remember](https://www.blogger.com/null)**

**A property cannot have a void return type.**

Lab2

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

**Properties.cs**

Hide   Copy Code

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 compiler understands “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 return statement empty, and remove 5, we do not get any error and code compiles,

using System;

namespace Properties  
{  
public class Properties  
{  
public int Age  
{  
set { return ; }  
}  
}  
}

Value Keyword

We have a reserved keyword named value.

Hide   Copy Code

using System;

namespace Properties  
{  
public class Properties  
{  
public string Name  
{  
set { string value; }  
}  
}  
}

Just compile the above given 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’, which 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 “set” accessor as it may give different meaning to already reserved keyword value.

Abstract Properties

Lab1

Yes, we can also have abstract properties; let’s see how it works,

**Properties.cs**

Hide   Copy Code

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); }  
}  
}  
}

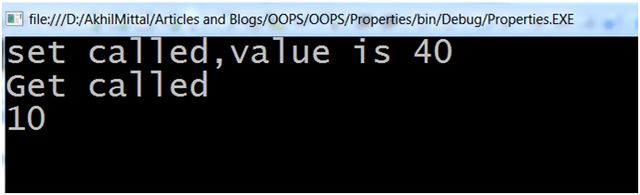
**Program.cs**

Hide   Copy Code

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**



In above example, I just created a base class named “BaseClass” and defined an abstract property namedAbsproperty. 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 have to override it in my derived class to add functionality to it. So I used override keyword in my derived class.

In base class, abstract property has no body at all, neither for “get” and nor for “set”, so we have to implement both the accessor in our derived class, like shown in “Properties” class.

**[Point to remember](https://www.blogger.com/null)**

**If one do not mark property defined in derived class as override, it will by default be considered as new.**

For more understanding follow <http://www.codeproject.com/Articles/774578/Diving-in-OOP-Day-Polymorphism-and-Inheritance-Dyn> article on new and override.

[Lab2](https://www.blogger.com/null)

**Properties.cs**

Hide   Copy Code

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**

Hide   Copy Code

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

**[Point to remember](https://www.blogger.com/null)**

**You cannot override an accessor that is not defined in a base class abstract property.**

Properties in Inheritance

Just follow the given code,

**Properties.cs**

Hide   Copy Code

using System;

namespace Properties  
{  
public class PropertiesBaseClass  
{  
public int Age  
{  
set {}  
}  
}

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

**Program.cs**

Hide   Copy Code

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 above given code, in *Properties.cs* file I created two classes one is Base i.e.PropertiesBaseClass and second in Derived i.e. PropertiesDerivedClass. I purposely declared set accessor in Base class and get in Derived class for the same property name i.e. Age. Now this case may give you the feeling that when compiled, our code of property Age will become one, i.e. it will take set from Base class and get from derived class and combine it into a single entity of Age property.But this is practically not the case. The compiler treats both these properties differently, and does not consider them to be same. In this case the property in derived class actually hides the property in base class , they are not the same but independent properties.The same concept of method hiding applies here too. You can read about hiding in<http://www.codeproject.com/Articles/774578/Diving-in-OOP-Day-Polymorphism-and-Inheritance-Dyn>.

To use the property of base class from a derived class object, you need to cast it to base class and then use it.

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

**Error Property or indexer ‘Properties.PropertiesDerivedClass.Age’ cannot be assigned to — it is read only**

i.e. we can do ((PropertiesBaseClass) pDerivedClass).Age = 15;

but we cannot do pDerivedClass.Age = 10; because derived class property has no “set” accessor.

Summary

Let’s recall all the points that we have to remember,



* The variable used for property should be of same data type as the data type of the property.
* A property cannot have a void return type.
* If one do not mark property defined in 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.
* Get accessor is only used to read a property value. A property having only get cannot be set with any value from the caller.

**Diving into OOP (Day 8): Indexers in C# (A Practical Approach)**

Indexers in C# (The definition)

*“Indexers allow instances of a class or struct to be indexed just like arrays. Indexers resemble*[*properties*](https://msdn.microsoft.com/en-us/library/x9fsa0sw.aspx) *except that their accessors take parameters.”* indexers allow us to leverage the capability of accessing the class objects as an array.

Lab 1

Hide   Copy Code

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 Indexer class and tried to use that object as an array. Since actually it was not an array, it resulted as a compile time error.

Lab 2

**Indexer.cs**

Hide   Copy Code

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**

Hide   Copy Code

namespace Indexers

{

class Program

{

static void Main(string[] args)

{

Indexer indexer=new Indexer();

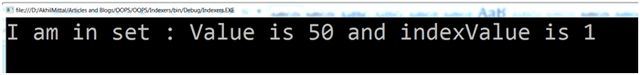
indexer[1] = 50;

}

}

}

**Output**



Here we just made a use of indexer to index my object of the class Indexer. Now my object can be used as an array to access different 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 automatically gets called. And the keyword “value” in “set” accessor was used to hold or keep track of the assigned value to our property. In above example, indexer[1] = 50;

calls the “set” accessor of “this” property i.e. an indexer therefore 50 becomes value and 1 becomes index of that value.

Lab 3

**Indexer.cs**

Hide   Copy Code

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 20;  
}  
}  
}  
}

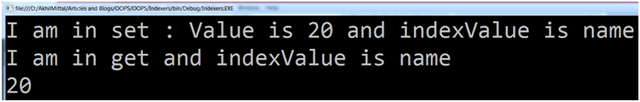
**Program.cs**

Hide   Copy Code

using System;

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

**Output**



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

Data-Types in Indexers

Lab 1

**Indexer.cs**

Hide   Copy Code

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;  
}  
}  
}  
}

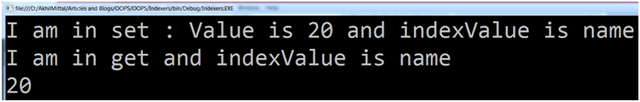
**Program.cs**

Hide   Copy Code

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**



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

The string parameter “indexValue” has a value “name”, like we passed in Main method of *Program.cs*. So one can have more than one indexers in a class deciding what should be the data type of the parameter value of array. An indexer, like properties follow 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 having following code,

Hide   Copy 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 through IIndexers interface,

**Indexer.cs**

Hide   Copy Code

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 hold names. Now we can implement 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**

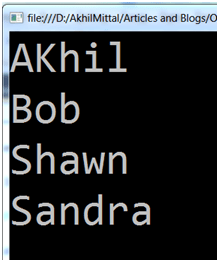
Hide   Copy Code

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,



In main method, we took Interface reference to create an object of IndexerClass, and we accessed that object array through indexer values like an array. It gives the names one by one.

Now if I want to access “set” accessor as well, I can easily do that. To check this, just add two more lines where you set the value in indexer,

Hide   Copy Code

iIndexer[2] = "Akhil Mittal";

Console.WriteLine(iIndexer[2]);

I set the value of 2nd element as a new name, let’s see the output,



Indexers in Abstract class

Like we used indexers in Interfaces, we can also use indexers in abstract class. 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**

Hide   Copy Code

namespace Indexers

{

public abstract class AbstractBaseClass

{

public abstract string this[int indexerValue] { get; set; }

}

}

Define derived class, inheriting from abstract class,

**IndexerClass**

We here use override in indexer to override the abstract indexer declared in abstract class.

Hide   Copy Code

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 reference of abstract class to create an object of Indexer class.

Hide   Copy Code

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:



All of the above code is self-explanatory. You can explore more scenarios by yourself for more detailed understanding.

Indexer Overloading

**Indexer.cs**

Hide   Shrink https://i0.wp.com/www.codeproject.com/images/arrow-up-16.png   Copy Code

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);  
}  
}  
}  
}

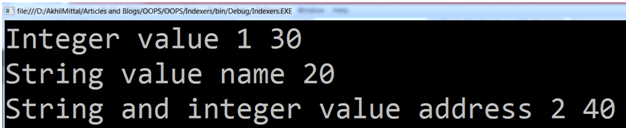
**Program.cs**

Hide   Copy Code

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**



In above example, we see that an indexer’s signature in actually count of 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 like we do in method overloading. You can read more about method over loading in<http://www.codeproject.com/Articles/771455/Diving-in-OOP-Day-Polymorphism-and-Inheritance-Ear>. Here now we have overloaded indexers that takes integer, string integer and string combined as actual parameters. Like methods cannot be overloaded on the base of return types, so indexers follow the same methodology of overload like methods do.

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 last section, just add a static keyword to the indexer signature,

Hide   Copy Code

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 hane a property can be static too.

Point to remember

Properties can be static but indexers cannot be.

Inheritance/Polymorphism in Indexers

**Indexer.cs**

Hide   Shrink https://i0.wp.com/www.codeproject.com/images/arrow-up-16.png   Copy Code

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 + “ dValue from base class indexer: “ + dValue);  
return 500;  
}  
set  
{  
Console.WriteLine(“Set of IndexerDerivedClass; indexer value: “ + indexerValue + “ set value “ + value);  
base[indexerValue] = value;  
}

}  
}  
}

**Program.cs**

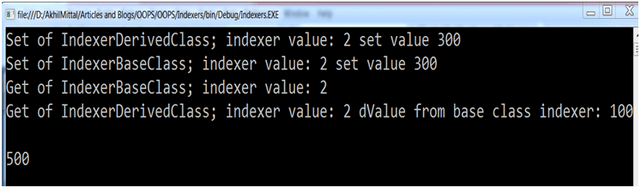
Hide   Copy Code

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**



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 like we discussed in prior examples. There after a derived class is created named IndexerDerivedClass, this derives from IndexerBaseClass and overrides “this” indexer from base class, note that base class indexer is marked virtual, so we can override it in derived class by marking it “override” in derived class.The example makes call to indexer of base class. Sometimes when we need to override code in derived class in the derived class, we may require the base class indexer should be called first. This is just a situation. The same rule of run time polymorphism applies here , we declare base class indexer and virtual and derived class one as override. In “set” accessor of derived class, we can call 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 “value” keyword too. So, indexDerived[2] in Main() method of *Program.cs* gets replaced to base[2] in “set” accessor. Whereas In “get” accessor it is vice versa, we require to putbase[indexerValue] to right hand side of equal sign. The “get” accessor in base class returns a value, i.e. 100, which we get in dValue variable.

.NET Framework and Indexers

Indexers play a crucial role in .NET framework. Indexers are widely used in .NET Framework inbuilt classes, libraries such as collections and enumerable. Indexers are used in collections that are searchable like Dictionary, Hashtable, List, Arraylist etc.

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.

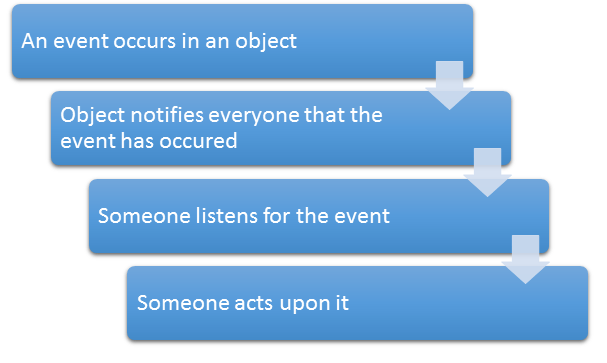
Properties vs Indexers

I have already explained a lot about properties and indexers, to summarize, let me point to an [MSDN](https://msdn.microsoft.com/en-us/library/4bsztef7.aspx) link for better understanding,

| **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](https://msdn.microsoft.com/en-us/library/ms228503.aspx) accessor of a property has no parameters. | A get accessor of an indexer has the same formal parameter list as the indexer. |
| A [set](https://msdn.microsoft.com/en-us/library/ms228368.aspx) 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](https://msdn.microsoft.com/en-us/library/a1khb4f8.aspx) parameter. |
| Supports shortened syntax with [Auto-Implemented Properties (C# Programming Guide)](https://msdn.microsoft.com/en-us/library/bb384054.aspx). | Does not support shortened syntax. |

**Understanding Events in C# (An Insight)**

*Events are declared using delegates. If you have not yet studied the Delegates Tutorial, you should do so before continuing. Recall that a delegate object encapsulates a method so that it can be called anonymously. An event is a way for a class to allow clients to give it delegates to methods that should be called when the event occurs. When the event occurs, the delegate(s) given to it by its clients are invoked.”*



In C#, delegates are used with events to implement event handling. The .NET Framework event model uses delegates to bind notifications with methods known as event handlers. When an event is generated, the delegate calls the associated event handler.

Delegates play a very important role in C#, it is one of the entities that can be directly put into a namespace of a class. This quality of delegate makes it accessible to all other classes as well. Delegates work on Object oriented pattern and try to follow encapsulation by enclosing method and object together. Delegate in C# define a class and uses namespace System.Delegate. Delegates are ideal for anonymous method invocation. I’ll discuss events and delegates in more detail in my upcoming articles.

The following figure shows the mechanism used by the publisher and subscriber objects.

