Object Oriented Programming Using C# .NET

C# is an object oriented programming language. OOP includes classes, objects, overloading, encapsulation, data hiding, and inheritance. In this article, we will learn how to write code for Classes and Objects, Constructor and Destructor, Function Overloading, Encapsulation, Inheritance, Interface, Polymorphism in C# and .NET.

Object-oriented programming (OOP) is the core ingredient of the .NET framework. OOP is so important that, before embarking on the road to .NET, you must understand its basic principles and terminology to write even a simple program. The fundamental idea behind OOP is to combine into a single unit both data and the methods that operate on that data; such units are called an object. All OOP languages provide mechanisms that help you implement the object-oriented model. They are encapsulation, inheritance, polymorphism and reusability. Let's now take a brief look at these concepts.

This chapter covers the following:

1. OOP's overview
2. Classes and Objects
3. Constructor and Destructor
4. Function Overloading
5. Encapsulation
6. Inheritance
7. Interface
8. Polymorphism

**Encapsulation**

Encapsulation binds together code and the data it manipulates and keeps them both safe from outside interference and misuse. Encapsulation is a protective container that prevents code and data from being accessed by other code defined outside the container.

**Inheritance**

Inheritance is the process by which one object acquires the properties of another object. A type derives from a base type, taking all the base type members fields and functions. Inheritance is most useful when you need to add functionality to an existing type. For example all .NET classes inherit from the System.Object class, so a class can include new functionality as well as use the existing object's class functions and properties as well.

**Polymorphism**

Polymorphism is a feature that allows one interface to be used for a general class of action. This concept is often expressed as "one interface, multiple actions". The specific action is determined by the exact nature of circumstances.

**Reusability**

Once a class has been written, created and debugged, it can be distributed to other programmers for use in their own program. This is called reusability, or in .NET terminology this concept is called a component or a DLL. In OOP, however, inheritance provides an important extension to the idea of reusability. A programmer can use an existing class and without modifying it, add additional features to it.

**Compiling the Program**

You can compile a C# program into either an assembly or a module. If the program has one class that contains a Main () method then it can be compiled directly into an assembly. This file has an ".exe" extension. A program with no Main() method can be compiled into a module as in the following:

*csc /target:module "program name"*

You can then compile this program by F9 or by simply running the C# command line compiler (csc.exe) against the source file as the following:

*csc oops.cs*

Classes and Objects

Classes are special kinds of templates from which you can create objects. Each object contains data and methods to manipulate and access that data. The class defines the data and the functionality that each object of that class can contain.

A class declaration consists of a class header and body. The class header includes attributes, modifiers, and the class keyword. The class body encapsulates the members of the class, that are the data members and member functions. The syntax of a class declaration is as follows:

Attributes accessibility modifiers class identifier:*baselist { body }*

Attributes provide additional context to a class, like adjectives; for example the Serializable attribute. Accessibility is the visibility of the class. The default accessibility of a class is internal. Private is the default accessibility of class members. The following table lists the accessibility keywords;

|  |  |
| --- | --- |
| Keyword | Description |
| public | Public class is visible in the current and referencing assembly. |
| private | Visible inside current class. |
| protected | Visible inside current and derived class. |
| Internal | Visible inside containing assembly. |
| Internal protected | Visible inside containing assembly and descendent of thecurrent class. |

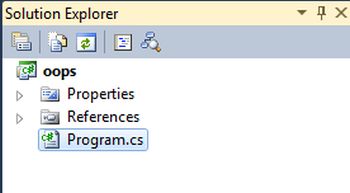
Modifiers refine the declaration of a class. The list of all modifiers defined in the table are as follows;

|  |  |
| --- | --- |
| Modifier | Description |
| sealed | Class can't be inherited by a derived class. |
| static | Class contains only static members. |
| unsafe | The class that has some unsafe construct likes pointers. |
| Abstract | The instance of the class is not created if the Class is abstract. |

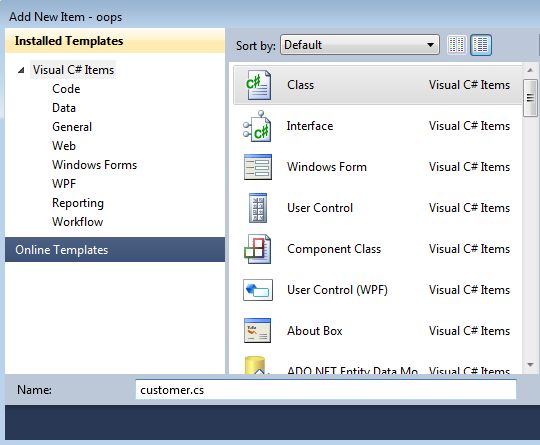
The baselist is the inherited class. By default, classes inherit from the System.Object type. A class can inherit and implement multiple interfaces but doesn't support multiple inheritances.

**Step-by-step Tutorial for Creating a Class**

1. Open Visual Studio 2010 from start menu.
2. Go to "File" > "New" > "Project...", select "Console Application" in the right pane and provide the name "oops" for the project.
3. Then in the Solution Explorer, you will notice some files that are automatically created as,



1. You can also write your own code in the default program.cs file that is created but it is a good programming practice to create a new class.
2. For adding a new class, right-click over the project name (oops) in the Solution Explorer, then click "Add" > "Class". Give the name to the class "customer" as in the following;



When you open the customer.cs class. you will find some default-generated code as in the following,

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
6. **namespace** oops
7. {
8. **class** customer
9. {
10. }
11. }

**Note**

The C# console application project must require a single entry point Main () function that is already generated in the program class. For example if you add a new customer class and want to define one or more Main () entry points here then .NET will throw an error of multiple entry points. So it is advisable to delete or exclude the program.cs file from the solution.

So here in this example the customer class defines fields such as CustID, Name and Address to hold information about a particular customer. It might also define some functionality that acts upon the data stored in these fields.

1. **using** System;
3. **namespace** oops
4. {
5. **class** customer
6. {
7. // Member Variables
8. **public** **int** CustID;
9. **public** **string** Name;
10. **public** **string** Address;
12. //constuctor for initializing fields
13. customer()
14. {
15. CustID=1101;
16. Name="Tom";
17. Address="USA";
18. }
20. //method for displaying customer records (functionality)
21. **public** **void** displayData()
22. {
23. Console.WriteLine("Customer="+CustID);
24. Console.WriteLine("Name="+Name);
25. Console.WriteLine("Address="+Address);
26. }
27. // Code for entry point
28. }
29. }

At line 9, we are defining a constructor of the customer class for initializing the class member fields. The constructor is a special function that is automatically called when the customer class object is created (instantiated). And at line 11 we are printing these fields to the console by creating a user defined method displayData().

You can then instantiate an object of this class to represent one specific customer, set the field value for that instance and use its functionality, as in:

1. **class** customer
2. {
3. // class members code
5. //Entry point
6. **static** **void** Main(**string**[] args)
7. {
8. // object instantiation
9. customer obj = **new** customer();
11. //Method calling
12. obj.displayData();
14. //fields calling
15. Console.WriteLine(obj.CustID);
16. Console.WriteLine(obj.Name);
17. Console.WriteLine(obj.Address);
18. }
19. }

**Multiple Class Declaration**

Sometimes circumstances require multiple classes to be declared in a single namespace. So in that case it is not mandatory to add a separate class to the solution, instead you can attach the new class into the existing program.cs or another one as in the following;

1. **using** System;
3. **namespace** oops
4. {
5. **class** Program
6. {
8. **public** **void** MainFunction()
9. {
10. Console.WriteLine("Main class");
11. }
12. **static** **void** Main(**string**[] args)
13. {
14. //main class instance
15. Program obj = **new** Program();
16. obj.MainFunction();
18. //other class instace
19. demo dObj=**new** demo();
20. dObj.addition();
21. }
22. }
24. **class** demo
25. {
26. **int** x = 10;
27. **int** y = 20;
28. **int** z;
30. **public** **void** addition()
31. {
32. z = x + y;
33. Console.WriteLine("other class in Namespace");
34. Console.WriteLine(z);
35. }
36. }
37. }

Here in this example, we are creating an extra class "demo" in the program.cs file at line 12 and finally we are instantiating the demo class with the program class inside the Main() entry in lines 6 to 11. So it doesn't matter how many classes we are defining in a single assembly.

**Partial classes**

Typically, a class will reside entirely in a single file. However, in situations where multiple developers need access to the same class, then having the class in multiple files can be beneficial. The partial keywords allow a class to span multiple source files. When compiled, the elements of the partial types are combined into a single assembly.

There are some rules for defining a partial class as in the following;

* A partial type must have the same accessibility.
* Each partial type is preceded with the "partial" keyword.
* If the partial type is sealed or abstract then the entire class will be sealed and abstract.

In the following example we are adding two files, partialPart1.cs and partialPart2.cs, and declare a partial class, partialclassDemo, in both classes.

**partialPart1.cs**

1. **using** System;
3. **namespace** oops
4. {
5. **public** partial **class** partialclassDemo
6. {
7. **public** **void** method1()
8. {
9. Console.WriteLine("method from part1 class");
10. }
11. }
12. }

**partialPart2.cs**

1. **using** System;
3. **namespace** oops
4. {
5. **public** partial **class** partialclassDemo
6. {
7. **public** **void** method2()
8. {
9. Console.WriteLine("method from part2 class");
10. }
11. }
12. }

And finally we are creating an instance of the partialclassDemo in the program.cs file as the following:

**Program.cs**

1. **using** System;
3. **namespace** oops
4. {
5. **class** Program
6. {
7. **static** **void** Main(**string**[] args)
8. {
9. //partial class instance
10. partialclassDemo obj = **new** partialclassDemo();
11. obj.method1();
12. obj.method2();
13. }
14. }
15. }

**Static classes**

A static class is declared using the "static" keyword. If the class is declared as static then the compiler never creates an instance of the class. All the member fields, properties and functions must be declared as static and they are accessed by the class name directly not by a class instance object.

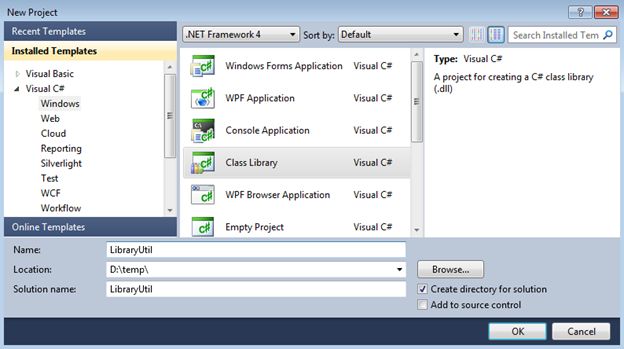
1. **using** System;
3. **namespace** oops
4. {
5. **static** **class** staticDemo
6. {
7. //static fields
8. **static** **int** x = 10, y;
10. //static method
11. **static** **void** calcute()
12. {
13. y = x \* x;
14. Console.WriteLine(y);
15. }
16. **static** **void** Main(**string**[] args)
17. {
18. //function calling directly
19. staticDemo.calcute();
20. }
21. }
22. }

**Creating and accessing Class Component Library**

.NET provides the capability of creating libraries (components) of a base application rather than an executable (".exe"). Instead the library project's final build version will be ".DLL" that can be referenced from other outside applications to expose its entire functionality.

**Step-by-step tutorial**

First create a class library based application as,



Then we are implementing a math class library that is responsible of calculating square root and the addition of two numbers as:

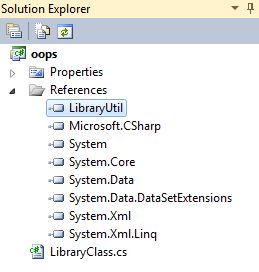
1. **using** System;
3. **namespace** LibraryUtil
4. {
5. **public** **class** MathLib
6. {
7. **public** MathLib() { }
9. **public** **void** calculareSum(**int** x, **int** y)
10. {
11. **int** z = x + y;
12. Console.WriteLine(z);
13. }
15. **public** **void** calculareSqrt(**double** x)
16. {
17. **double** z = Math.Sqrt(x);
18. Console.WriteLine(z);
19. }
20. }
21. }

Build this code and you will notice that a DLL file was created, not an executable, in the root directory of the application*(path = D:\temp\LibraryUtil\LibraryUtil\bin\Debug\ LibraryUtil.dll).*

Now create another console based application where you utilize all the class library's functionality.

Then you have to add the class library dll file reference to access the declared class in the library dll. (Right-click on the Reference then "Add reference" then select the path of the dll file.)

When you add the class library reference then you will notice in the Solution Explorer that a new LibraryUtil is added as in the following;



Now add the namespace of the class library file in the console application and create the instance of the class declared in the library as in the following;

1. **using** System;
2. **using** LibraryUtil; // add library namespace
3. **namespace** oops
4. {
5. **public** **class** LibraryClass
6. {
7. **static** **void** Main()
8. {
9. //library class instance
10. MathLib obj = **new** MathLib();
12. //method populate
13. obj.calculareSum(2, 5);
14. obj.calculareSqrt(25);
15. }
16. }
17. }

Finally run the application.

Constructor and Destructor

A constructor is a specialized function that is used to initialize fields. A constructor has the same name as the class. Instance constructors are invoked with the new operator and can't be called in the same manner as other member functions. There are some important rules pertaining to constructors as in the following;

* Classes with no constructor have an implicit constructor called the default constructor, that is parameterless. The default constructor assigns default values to fields.
* A public constructor allows an object to be created in the current assembly or referencing assembly.
* Only the extern modifier is permitted on the constructor.
* A constructor returns void but does not have an explicitly declared return type.
* A constructor can have zero or more parameters.
* Classes can have multiple constructors in the form of default, parameter or both.

The following example shows one constructor for a customer class.

1. **using** System;
2. **namespace** oops
3. {
4. **class** customer
5. {
6. // Member Variables
7. **public** **string** Name;
9. //constuctor for initializing fields
10. **public** customer(**string** fname, **string** lname)
11. {
12. Name= fname +" "+ lname;
13. }
14. //method for displaying customer records
15. **public** **void** AppendData()
16. {
17. Console.WriteLine(Name);
18. }
19. //Entry point
20. **static** **void** Main(**string**[] args)
21. {
22. // object instantiation
23. customer obj = **new** customer("Barack", "Obama");
25. //Method calling
26. obj.AppendData();
27. }
28. }
29. }

**Note**

The moment a new statement is executed, the default constructor is called.

**Static Constructor**

A constructor can be static. You create a static constructor to initialize static fields. Static constructors are not called explicitly with the new statement. They are called when the class is first referenced. There are some limitations of the static constructor as in the following;

* Static constructors are parameterless.
* Static constructors can't be overloaded.
* There is no accessibility specified for Static constructors.

In the following example the customer class has a static constructor that initializes the static field and this constructor is called when the class is referenced in the Main () at line 26 as in the following:

1. **using** System;
2. **namespace** oops
3. {
4. **class** customer
5. {
6. // Member Variables
7. **static** **private** **int** x;
9. //constuctor for static initializing fields
10. **static** customer()
11. {
12. x = 10;
13. }
14. //method for get  static field
15. **static** **public** **void** getData()
16. {
17. Console.WriteLine(x);
18. }
19. //Entry point
20. **static** **void** Main(**string**[] args)
21. {
22. //static Method calling
23. customer.getData();
24. }
25. }
26. }

**Destructors**

The purpose of the destructor method is to remove unused objects and resources. Destructors are not called directly in the source code but during garbage collection. Garbage collection is nondeterministic. A destructor is invoked at an undetermined moment. More precisely a programmer can't control its execution; rather it is called by the Finalize () method. Like a constructor, the destructor has the same name as the class except a destructor is prefixed with a tilde (~). There are some limitations of destructors as in the following;

* Destructors are parameterless.
* A Destructor can't be overloaded.
* Destructors are not inherited.
* Destructors can cause performance and efficiency implications.

The following implements a destructor and dispose method. First of all we are initializing the fields via constructor, doing some calculations on that data and displaying it to the console. But at line 9 we are implementing the destructor that is calling a Dispose() method to release all the resources.

1. **using** System;
2. **namespace** oops
3. {
4. **class** customer
5. {
6. // Member Variables
7. **public** **int** x, y;
8. //constuctor for  initializing fields
9. customer()
10. {
11. Console.WriteLine("Fields inititalized");
12. x = 10;
13. }
14. //method for get field
15. **public** **void** getData()
16. {
17. y = x \* x;
18. Console.WriteLine(y);
19. }
20. //method to release resource explicitly
21. **public** **void** Dispose()
22. {
23. Console.WriteLine("Fields cleaned");
24. x = 0;
25. y = 0;
26. }
27. //destructor
28. ~customer()
29. {
30. Dispose();
31. }
32. //Entry point
33. **static** **void** Main(**string**[] args)
34. {
35. //instance created
36. customer obj = **new** customer();
38. obj.getData();
40. }
41. }
42. }

At line 12 when the instance is created, fields are initialized but it is not necessary that at the same time the destructor is also called. Its calling is dependent on garbage collection. If you want to see the destructor being called into action then put a breakpoint (by F9) at line 10 and compile the application. The CLR indicates its execution at the end of the program by highlighting line 10 using the yellow color.

Function Overloading

Function overloading allows multiple implementations of the same function in a class. Overloaded methods share the same name but have a unique signature. The number of parameters, types of parameters or both must be different. A function can't be overloaded on the basis of a different return type alone.

1. **using** System;
2. **namespace** oops
3. {
4. **class** funOverload
5. {
6. **public** **string** name;
8. //overloaded functions
9. **public** **void** setName(**string** last)
10. {
11. name = last;
12. }
14. **public** **void** setName(**string** first, **string** last)
15. {
16. name = first + "" + last;
17. }
19. **public** **void** setName(**string** first, **string** middle, **string** last)
20. {
21. name = first + "" + middle + "" + last;
22. }
24. //Entry point
25. **static** **void** Main(**string**[] args)
26. {
27. funOverload obj = **new** funOverload();
29. obj.setName("barack");
30. obj.setName("barack "," obama ");
31. obj.setName("barack ","hussian","obama");
33. }
34. }
35. }

At lines 3, 4 and 5 we are defining three methods with the same name but with different parameters. In the Main (), the moment you create an instance of the class and call the functions setName() via obj at lines 7, 8 and 9 then intellisense will show three signatures automatically.

Encapsulation

Encapsulation is the mechanism that binds together the code and the data it manipulates, and keeps both safe from outside interference and misuse. In OOP, code and data may be combined in such a way that a self-contained box is created. When code and data are linked together in this way, an object is created and encapsulation exists.

Within an object, code, data or both may be private or public to that object. Private code is known to and accessible only by another part of the object, that is private code or data may not be accessible by a piece of the program that exists outside the object. When the code and data is public, other portions of your program may access it even though it is defined within an object.

1. **using** System;
2. **namespace** oops
3. {
4. **class** Encapsulation
5. {
6. /// <summary>
7. /// Every member Variable and Function of the class are bind
8. /// with the Encapsulation class object only and safe with
9. /// the outside inference
10. /// </summary>
12. // Encapsulation Begin
13. **int** x;
15. //class constructor
16. **public** Encapsulation(**int** iX)
17. {
18. **this**.x = iX;
19. }
21. //calculating the square
22. **public** **void** MySquare()
23. {
24. **int** Calc = x \* x;
25. Console.WriteLine(Calc);
26. }
28. // End of Encapsulation
30. //Entry point
31. **static** **void** Main(**string**[] args)
32. {
33. //instance created
34. customer obj = **new** customer(20);
36. obj. MySquare();
38. }
40. }
41. }

Inheritance

Inheritance is the process by which one object can acquire the properties of another object. Inheritance is a "is a kind of" relationship and it supports the concept of classification in which an object needs only define those qualities that make it unique within the class. Inheritance involves a base class and a derived class. The derived class inherits from the base class and also can override inherited members as well as add new members to extend the base class.

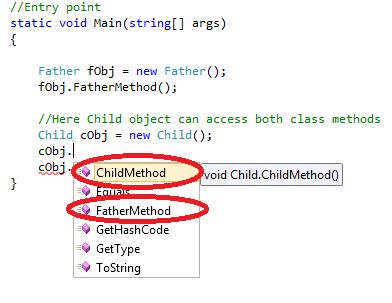
A base type represents the generalization, whereas a derived type represents a specification of an instance. Such as Employees that can have diverse types, such as hourly, salaried and temporary so in that case Employees is the general base class and hourly, salaried and temporary employee are specialized derived classes.

Classes can inherit from a single class and one or more interfaces. When inheriting from a class, the derived class inherits the members including the code of the base class. The important point to remember is that Constructors and Destructors are not inherited from the base class.

The syntax of inheritance is as in the following;

*Class derivedClass : baseClass, Iterface1, Interface2 { body }*

For example we are defining two classes, Father and Child. You notice at line 7, we are implementing inheritance by using a colon (:); at this moment all the properties belonging to the Father Class is accessible to the Child class automatically.



1. **using** System;
2. **namespace** oops
3. {
4. //Base Class
5. **public** **class** Father
6. {
7. **public** **void** FatherMethod()
8. {
9. Console.WriteLine("this property belong to Father");
10. }
11. }
13. //Derived class
14. **public** **class** Child : Father
15. {
16. **public** **void** ChildMethod()
17. {
18. Console.WriteLine("this property belong to Child");
19. }
20. }
21. **class** Inheritance
22. {
23. //Entry point
24. **static** **void** Main(**string**[] args)
25. {
27. Father fObj = **new** Father();
28. fObj.FatherMethod();
30. //Here Child object can access both class methods
31. Child cObj = **new** Child();
32. cObj.FatherMethod();
33. cObj.ChildMethod();
34. }
35. }
36. }

At line 11 , the Intellisense only shows the Father class functions but at line 15 to 16 the Child class object is able to access both class methods as in the following.

We can create a class in the VB.Net language or another .NET supported language and can inherit them in a C# .Net class and vice versa. But a class developed in C++ or other unmanaged environment can't be inherited in .NET.

**Note**

Cross-language and multiple inheritance is not supported by .NET.

**Accessibility**

Accessibility sets the visibility of the member to outside assemblies or derived types. The following table describes member accessibility;

|  |  |  |
| --- | --- | --- |
| Modifiers | Outside Assembly | Derived Class |
| private | No | No |
| public | Yes | Yes |
| protected | No | No |
| internal | Yes ( this assembly only) | Yes ( this assembly only) |
| internal protected | Yes ( this assembly only) | Yes |

**Constructor in Inheritance**

Constructors in a base class are not inherited in a derived class. A derived class has a base portion and derived portion. The base portion initializes the base portion, and the constructor of the derived class initializes the derived portion.

The following is the syntax of a constructor in inheritance;

Accessibility modifier classname(parameterlist1) : base(parameterlist2) { body }

So the base keyword refers to the base class constructor, while parameterlist2 determines which overloaded base class constructor is called.

In the following example, the Child class's constructor calls the single-argument constructor of the base Father class;

1. **using** System;
2. **namespace** oops
3. {
4. //Base Class
5. **public** **class** Father
6. {
8. //constructor
9. **public** Father()
10. {
11. Console.WriteLine("Father class constructor");
12. }
14. **public** **void** FatherMethod()
15. {
16. Console.WriteLine("this property belong to Father");
17. }
18. }
20. //Derived class
21. **public** **class** Child : Father
22. {
23. **public** Child()
24. : **base**()
25. {
26. Console.WriteLine("child class constructor");
27. }
28. **public** **void** ChildMethod()
29. {
30. Console.WriteLine("this property belong to Child");
31. }
32. }
33. **class** Inheritance
34. {
35. //Entry point
36. **static** **void** Main(**string**[] args)
37. {
38. //Here Child object can access both class methods
39. Child cObj = **new** Child();
40. cObj.FatherMethod();
41. cObj.ChildMethod();
42. Console.ReadKey();
43. }
44. }
45. }

At line 4, we are defining a base Father Class constructor and in the derived class Child, at line 8 we are initializing it explicitly via base keyword. If we pass any parameter in the base class constructor then we have to provide them in the base block of the child class constructor.

**Virtual Methods**

By declaring a base class function as virtual, you allow the function to be overridden in any derived class. The idea behind a virtual function is to redefine the implementation of the base class method in the derived class as required. If a method is virtual in the base class then we have to provide the override keyword in the derived class. Neither member fields nor static functions can be declared as virtual.

1. **using** System;
2. **namespace** oops
3. {
4. **class** myBase
5. {
6. //virtual function
7. **public** **virtual** **void** VirtualMethod()
8. {
9. Console.WriteLine("virtual method defined in the base class");
10. }
11. }
13. **class** myDerived : myBase
14. {
15. // redifing the implementation of base class method
16. **public** **override** **void** VirtualMethod()
17. {
18. Console.WriteLine("virtual method defined in the Derive class");
19. }
20. }
21. **class** virtualClass
22. {
23. **static** **void** Main(**string**[] args)
24. {
25. // class instance
26. **new** myDerived().VirtualMethod();
27. Console.ReadKey();
28. }
29. }
30. }

**Hiding Methods**

If a method with the same signature is declared in both base and derived classes, but the methods are not declared as virtual and overriden respectively, then the derived class version is said to hide the base class version. In most cases, you would want to override methods rather than hide them. Otherwise .NET automatically generates a warning.

In the following example, we are defining a VirutalMethod() in the myBase class but not overriding it in the derived class, so in that case the compiler will generate a warning. The compiler will assume that you are hiding the base class method. So to overcome that problem, if you prefix the new keyword in the derived class method then the compiler will prefer the most derived version method. You can still access the base class method in the derived class by using the base keyword.

1. **using** System;
2. **namespace** oops
3. {
4. **class** myBase
5. {
6. //virtual function
7. **public** **virtual** **void** VirtualMethod()
8. {
9. Console.WriteLine("virtual method defined in the base class");
10. }
11. }
13. **class** myDerived : myBase
14. {
15. // hiding the implementation of base class method
16. **public** **new** **void** VirtualMethod()
17. {
18. Console.WriteLine("virtual method defined in the Derive class");
20. //still access the base class method
21. **base**.VirtualMethod();
22. }
23. }
24. **class** virtualClass
25. {
26. **static** **void** Main(**string**[] args)
27. {
28. // class instance
29. **new** myDerived().VirtualMethod();
30. Console.ReadKey();
31. }
32. }
33. }

**Abstract Classes**

C# allows both classes and functions to be declared abstract using the abstract keyword. You can't create an instance of an abstract class. An abstract member has a signature but no function body and they must be overridden in any non-abstract derived class. Abstract classes exist primarily for inheritance. Member functions, properties and indexers can be abstract. A class with one or more abstract members must be abstract as well. Static members can't be abstract.

In this example, we are declaring an abstract class Employess with a method displayData() that does not have an implementation. Then we are implementing the displayData() body in the derived class. One point to be noted here is that we have to prefixe the abstract method with the override keyword in the derived class.

1. **using** System;
2. **namespace** oops
3. {
4. //abstract class
5. **public** **abstract** **class** Employess
6. {
7. //abstract method with no implementation
8. **public** **abstract** **void** displayData();
9. }
11. //derived class
12. **public** **class** test : Employess
13. {
14. //abstract class method implementation
15. **public** **override** **void** displayData()
16. {
17. Console.WriteLine("Abstract class method");
18. }
19. }
20. **class** abstractClass
21. {
22. **static** **void** Main(**string**[] args)
23. {
24. // class instance
25. **new** test().displayData();
26. }
27. }
28. }

**Sealed Classes**

Sealed classes are the reverse of abstract classes. While abstract classes are inherited and are refined in the derived class, sealed classes cannot be inherited. You can create an instance of a sealed class. A sealed class is used to prevent further refinement through inheritance.

Suppose you are a developer of a class library and some of the classes in the class library are extensible but other classes are not extensible so in that case those classes are marked as sealed.

1. **using** System;
2. **namespace** oops
3. {
4. **sealed** **class** SealedClass
5. {
6. **void** myfunv();
7. }
9. **public** **class** test : SealedClass //wrong. will give compilation error
10. {
11. }
12. }

Interface

An interface is a set of related functions that must be implemented in a derived class. Members of an interface are implicitly public and abstract. Interfaces are similar to abstract classes. First, both types must be inherited; second, you cannot create an instance of either. Although there are several differences as in the following;

* An Abstract class can contain some implementations but an interface can't.
* An Interface can only inherit other interfaces but abstract classes can inherit from other classes and interfaces.
* An Abstract class can contain constructors and destructors but an interface can't.
* An Abstract class contains fields but interfaces don't.

So the question is, which of these to choose? Select interfaces because with an interface, the derived type still can inherit from another type and interfaces are more straightforward than abstract classes.

1. **using** System;
2. **namespace** oops
3. {
4. // interface
5. **public** **interface** xyz
6. {
7. **void** methodA();
8. **void** methodB();
9. }
11. // interface method implementation
12. **class** test : xyz
13. {
14. **public** **void** methodA()
15. {
16. Console.WriteLine("methodA");
17. }
18. **public** **void** methodB()
19. {
20. Console.WriteLine("methodB");
21. }
22. }
23. **class** interfaceDemo
24. {
25. **static** **void** Main(**string**[] args)
26. {
27. test obj = **new** test();
28. obj.methodA();
29. obj.methodB();
30. }
31. }
32. }

An interface can be inherited from other interfaces as in the following:

1. **public** **interface** xyz
2. {
3. **void** methodA();
4. **void** methodB();
5. }
7. **public** **interface** abc : xyz
8. {
9. **void** methodC();
10. }

Polymorphism

Polymorphism is the ability to treat the various objects in the same manner. It is one of the significant benefits of inheritance. We can decide the correct call at runtime based on the derived type of the base reference. This is called late binding.

In the following example, instead of having a separate routine for the hrDepart, itDepart and financeDepart classes, we can write a generic algorithm that uses the base type functions. The method LeaderName() declared in the base abstract class is redefined as per our needs in 2 different classes.

1. **using** System;
2. **namespace** oops
3. {
4. **public** **abstract** **class** Employee
5. {
6. **public** **virtual** **void** LeaderName()
7. {
8. }
9. }
11. **public** **class** hrDepart : Employee
12. {
13. **public** **override** **void** LeaderName()
14. {
15. Console.WriteLine("Mr. jone");
16. }
17. }
18. **public** **class** itDepart : Employee
19. {
20. **public** **override** **void** LeaderName()
21. {
22. Console.WriteLine("Mr. Tom");
23. }
24. }
26. **public** **class** financeDepart : Employee
27. {
28. **public** **override** **void** LeaderName()
29. {
30. Console.WriteLine("Mr. Linus");
31. }
32. }
34. **class** PolymorphismDemo
35. {
36. **static** **void** Main(**string**[] args)
37. {
38. hrDepart obj1 = **new** hrDepart();
39. itDepart obj2 = **new** itDepart();
40. financeDepart obj3 = **new** financeDepart();
42. obj1.LeaderName();
43. obj2.LeaderName();
44. obj3.LeaderName();
46. Console.ReadKey();
47. }
48. }
49. }