ABSTRACT CLASSES AND INTERFACES

Abstract classes

The abstract modifier is used to indicate that a class is incomplete and intended only to be a base class of other classes. The abstract class only provides the signature or declaration of the abstract methods and leaves the implementation of these methods to derived or sub-classes. An abstract class differs from a non-abstract class in the following ways:

- An abstract class cannot be instantiated, and it is an error to use the new operator on an abstract class. While it is possible to have variables and values whose compile-time types are abstract, such variables and values will necessarily either be null or contain references to instances of non-abstract classes derived from the abstract types.
- An abstract class is permitted (but not required) to contain abstract methods and accessors.
- An abstract class cannot be sealed.
- An abstract function is automatically virtual (although you don't need to supply the virtual keyword).

When a non-abstract class is derived from an abstract class, the non-abstract class must include actual implementations of all inherited abstract methods and accessors, or it too must be declared as an abstract class. Such implementations are provided by overriding the abstract methods and accessors as shown below:

```
abstract class A {
     public abstract void F();
}
abstract class B: A {
     public  void  G()  {}
}
class C: B {
     public override void F() {
          //  actual  implementation  of  F
     }
}
```

A class inheriting an abstract class and implementing all its abstract methods is called the concrete class of the abstract class. We can declare a reference of the type of abstract class and it can point to the objects of the classes that have inherited the abstract class. If any class contains any abstract methods, then that class is also abstract and must be declared as such.

Example

```
using System;
abstract  class  Shape
{
     public abstract void Area();
}
class   Circle:Shape
{
     private double radius;
     public  Circle(double  radius)
     {
          this.radius=radius;
     }
     public  override  void  Area()
     {
          Console.WriteLine("Area of Circle " +
          Math.PI*Math.Pow(radius,2));
      }
}
```

```
class   Rectangle:Shape
{
      private double length, width;
      public  Rectangle(double  length,  double  width)
      {
            this.length=length;
            this.width=width;  }
      public  override  void  Area()
      {
            Console.WriteLine("Area  of  Rectangle  "  +
            length*width);
      }
}
class   Tester
{
      static  void  Main()
      {
            Shape  []  shapes  =  {new  Circle(7),  new  Rectangle(5,4)};
            Random random = new Random();
            for(int  i=0;  i<5;  i++)
            {
                  int  randNum  =  random.Next(0,  2);
                  shapes[randNum].Area();
            }
      }
}
```

Interfaces

An interface is a named collection of semantically related abstract methods and often helps in providing a standard structure that the deriving classes would follow. In general, an interface can only contain declarations of methods, properties, indexers, and events and whose members are all by default public and abstract. Notice that interfaces cannot contain constants, fields (private data members), constructors and destructors or any type of static member.

An interface is declared using the interface keyword. Interfaces, similar to abstract classes, cannot be instantiated but it can only declare the reference of the interface type and then point to any class implementing the interface.

The implementation of the methods in an interface is done in the class (or structure) that implements the interface. A class implementing the interface must provide the body for all the members of the interface. The compiler enforces this specification and does not compile any concrete class which inherits the interface, but does not implement all the members of the interface. A colon: is used to show that a class is implementing a particular interface.

Classes and structs may implement multiple interfaces, contrary to class-inheritance where you can inherit only one class. Interface itself can inherit other interfaces. It is a convention in C#, to prefix the name of interfaces with uppercase 'I' like IDisposable, ISerializable, IEnumerator, etc.

Contrasting interfaces to abstract classes

i). Interfaces are pure protocol. Interfaces never define state data and never provide an implementation of the methods

ii). Interface types are also quite helpful given that C# only support single inheritance

iii). Interfaces provide another way to inject polymorphic behavior into a system

Example

```
using System;
interface   IShape
{
     double   Dimension1
     {
          get; set;
     }
     double Dimension2
     {
          get; set;
     }
     void Area();
}
class   Rectangle:IShape
{
     private  double  dimension1,  dimension2;
     public double Dimension1
     {
          get { return dimension1; }
          set { dimension1 = value; }
     }
     public  double  Dimension2
     {
          get { return dimension2; }
          set { dimension2 = value; }
     }
     public  void  Area()
     {
          Console.WriteLine("Area  of  Rectangle  "  +
          dimension1*dimension2);
     }
}
class Triangle : IShape {
    private  double  dimension1,dimension2;
    public double Dimension1
     {
          get { return dimension1;}
          set { dimension1 = value;}
     }
     public  double  Dimension2
     {
          get { return dimension2;}
          set { dimension2 = value;}
     }
    public void Area() {
          Console.WriteLine("Area of Triangle " +
          1.0/2.0*dimension1*dimension2);
    }
}
class   Tester
{
     static void Main()
```

```
          {
               Rectangle rectangle=new Rectangle();
               Triangle triangle=new Triangle();
               IShape shape;
               shape= rectangle as IShape;
               shape.Dimension1=5;
               shape.Dimension2=4;
               shape.Area();
               shape= triangle as IShape;
               shape.Dimension1=7;
               shape.Dimension2=2;
               shape.Area();  }
     }
```

Note that while all the implementing members in Rectangle or Triangle are declared as public, they are not declared public in the IShape interface. If we don't mark these members as public, the compiler will flag an error since all the members of the interface are abstract and public by default and we cannot decrease the accessibility level of the original member during polymorphism. Also note that there is no override keyword when overriding the abstract methods of the interface. The reason for not applying the override keyword is that we do not actually override any default implementation, but provide our own specific implementation for the members.

Implementing more than One Interface

A class can implement more than one interface. In such a case, a class has to provide the implementation for all the members of each of the implementing interfaces.

Example: Suppose we have two interfaces IAdd and IMultiply as follows:

```
using System;
interface  IAdd
{
     int Add();
}
interface IMultiply
{
     int Multiply();
}
class  Computation:IAdd,IMultiply
{
     int number1,number2;
     public Computation(int number1,int number2)
     {
          this.number1=number1;
          this.number2=number2;
     }
     public int Add()
     {
          return(number1+number2);
     }
     public int Multiply()
     {
          return(number1*number2);
     }
}
```

```
class   InterfaceTest
{
     public  static  void  Main()
     {
          Computation computation=new Computation(10,20);
          IAdd add=(IAdd)computation; //casting
          Console.WriteLine("Sum= "+add.Add());

          IMultiply multiply=(IMultiply)computation;
          Console.WriteLine("Product=    "+multiply.Multiply());
     }
}
```

Interfaces and Inheritance

A base class of a derived class may implement an interface. When an object of the derived class is converted to the interface type, the inheritance hierarchy is searched for a class that directly implements the interface.

Example

```
using System;
interface   IDisplay
{
     void Print();
}
class   DisplayBase:IDisplay
{
     public  void  Print()
     {
          Console.WriteLine("Base Display");
     }
}
class   DisplayDerived:DisplayBase
{
     public  new  void  Print()
     {
          Console.WriteLine("Derived Display");
     }
}
class   TestInterface
{
     public  static  void  Main()
     {
          DisplayDerived  d=new  DisplayDerived();
          d.Print();
          IDisplay   dis=(IDisplay)d;
          dis.Print();
     }
}
```

Explicit implementation of methods

If a class is implementing more than one interface and at least two of them have methods with similar signatures, then we can provide explicit implementation of the particular method by prefixing its name with the name of the interface and the . operator. Consider the case defined below where we have two interfaces (IDisplay1 and IDisplay2) and both contain a Display method with identical signatures.

```
using System;
interface   IDisplay1
{
     void display();
}
interface IDisplay2
{
     void display();
}
class   DisplayClass:IDisplay1,IDisplay2
{
     void   IDisplay1.display()
     {
          Console.WriteLine("Interface 1 Display");
     }
     void IDisplay2.display()
     {
          Console.WriteLine("Interface 2 Display");
     }
}
class   InterfaceTester
{
     public  static  void  Main()
     {
          DisplayClass display=new DisplayClass();
          IDisplay1 test1=(IDisplay1)display;
          test1.display();
          IDisplay2   test2=(IDisplay2)display;
          test2.display();
     }
}
```

Abstract Class and Interfaces

Implemented interface methods can be declared virtual or abstract (i.e., an interface can be implemented by an abstract class) as shown below

Example:

```
interface   A
{
     void Method();
}
abstract class B:A
{
     ……….
     public abstract void Method();
}
```

In the above example, class B does not implement the interface method; it simply redeclares as a public abstract method. It is the duty of the class that derives from B to override and implement the method.

An Interface Inheriting One or More Interfaces

An interface can inherit from more than one interface.

Example

The following is a C# program that illustrates an interface (IFile) inheriting two other interfaces IReadable and IWritable

```
using System;
interface   IWritable
{
     void Write(string s);
}
interface IReadable
{
     string ReadLine();
}
interface  IFile  :  IWritable,  IReadable
{
     void  Open(string  filename);
     void Close();
}
class  MyFile  :  IFile
{
     private string filename;
     public  void  Open(string  filename)
     {
          this.filename = filename;
          Console.WriteLine("Opening file: {0}", filename);
     }
     public string ReadLine()
     {
          return "Reading a line from MyFile: " + filename;
     }
     public  void  Write(string  s)
     {
          Console.WriteLine("Writing  '{0}'  in  the  file:  {1}",  s,
          filename);
     }
     public  void  Close()
     {
          Console.WriteLine("Closing the file: {0}", filename);
     }
}
class   Test
{
     static  void  Main()
     {
          MyFile aFile = new MyFile();
          aFile.Open("c:\\csharp.txt");
          aFile.Write("My name is Frank");
          Console.WriteLine(aFile.ReadLine());
          aFile.Close();
     }
}
```

Here, we have created an instance of MyFile and named the reference aFile. We later call different methods on the object using the reference.