Abstract Classes, Interfaces Namespaces & Indexers

C# Programming

Introduction to Abstract Class

- Abstract classes are one of the essential behaviours provided by .NET.
- Commonly, you would like to make classes that only represent base classes, and don't want anyone to create objects of these class types.
- You can make use of abstract classes to implement such functionality in C# using the modifier 'abstract'.
- An abstract class means that, no object of this class can be instantiated, but can make derivations of this.
- An example of an abstract class declaration is:

```
abstract class absClass
{
}
```

Abstract Method

- An abstract class can contain either abstract methods or non abstract methods. Abstract members do not have any implementation in the abstract class, but the same has to be provided in its derived class as:

```
abstract class absClass
{
    public abstract void abstractMethod();
}
```

- Also, note that an abstract class does not mean that it should contain abstract members. Even we can have an abstract class only with non abstract members. For example:

```
abstract class absClass
{
    public void NonAbstractMethod()
    {
        Console.WriteLine("NonAbstract Method");
    }
}
```

Abstract Classes

- Abstract classes are classes that cannot be instantiated.
- They are used as base class containing common fields and functionality for the subclasses.
- An abstract class is created by using abstract keyword
- Abstract class can contain
  - concrete methods (methods with definition)
  - abstract methods (methods declaration only)
- It is not compulsory for an abstract class to have abstract methods but if a class has even one abstract method then the class has to be marked as abstract.

Classes inheriting from abstract class

- A class inheriting from abstract class must provide implementation for the abstract method by overriding it or mark itself as abstract class.
- An abstract method is implicitly a virtual method but it cannot have the virtual modifier.
- Abstract when an abstract class inherits a virtual method from a base class, the abstract class can override the virtual method with an abstract method.

Abstract Class Example

```
using System;
abstract class Figure
{
    public int Dimension;
    public abstract double Area();
    public abstract double Perimeter();
}
class Square : Figure
{
    public override double Area()
    {
        return Dimension * Dimension;
    }
    public override double Perimeter()
    {
        return 4 * Dimension;
    }
}
```

Presented by

Ranjan Bhatnagar

Abstract Class Example

```
class Circle : Figure
{
    public override double Area()
    {
        return Math.PI * Dimension * Dimension;
    }
    public override double Perimeter()
    {
        return 2 * Math.PI * Dimension;
    }
}
class Program
{
    static void Main()
    {
        Figure fig = new Square(); // or Circle();
        fig.Dimension = 10;
        Console.WriteLine(fig.Area());
        Console.WriteLine(fig.Perimeter());
    }
}
```

Important rules applied to abstract classes

- An abstract class cannot be a sealed class.
- Declaration of abstract methods are only allowed in abstract classes.
- An abstract method cannot be private.
- The access modifier of the abstract method should be same in both the abstract class and its derived class.
- An abstract method cannot have the modifier virtual. Because an abstract method is implicitly virtual.
- An abstract member cannot be static.

Abstract Properties

- The abstract modifier indicates that the thing being modified has a missing or incomplete implementation.
- An abstract property declaration does not provide an implementation of the property accessors.
- It declares that the class supports properties, but leaves the accessor implementation to derived classes.

Abstract Properties

```
abstract class dimensions
{
    protected int  height ;
    public abstract int  Height
    {
        get ;
        set ;
    }
}
```

Abstract Property - it is necessary to declare the class containing the property as **abstract**.
And the get and set keywords without any accessor body.
And the derived class must implement the property

```
public class window :
                        dimensions
{
    public override int  Height
    {
        get
        {
            return  height ;
        }
        set
        {
            height = value ;
        }
    }
}
```

Abstract Properties Example

```
using System;
public abstract class Shape
{
    private string name;
    public Shape(string s)
    {
        Id = s;
    }
    public string Id
    {
        get {
            return name;
        }

        set {
            name = value;
        }
    }
    public abstract double Area
    {
        get;
    }
    public override string ToString()
    {
        return Id + " Area = " + string.Format("{0:F2}", Area);
    }
}
```

```
public class Square : Shape
{
    private int side;

    public Square(int side, string id)
        : base(id)
    {
        this.side = side;
    }

    public override double Area
    {
        get
        {
            return side * side;
        }
    }
}
```

Example Cont.

```
public class Circle : Shape
{
    private int radius;
    public Circle(int radius, string id) : base(id)
    {
        this.radius = radius;
    }
    public override double Area
    {
        get {
            return radius * radius * System.Math.PI;
        }
    }
}
```

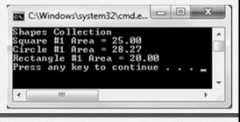
```
public class Rectangle : Shape {
    private int width;
    private int height;
    public Rectangle(int width, int height, string id): base(id)
    {
        this.width = width;
        this.height = height;
    }
    public override double Area
    {
        get {
            return width * height;
        }
    }
}
```

Presented by
Ranjan Bhatnagar

Example Cont.

```
class TestClass
{
    static void Main()
    {
        Shape[] shapes =
        {
            new Square(5, "Square #1"),
            new Circle(3, "Circle #1"),
            new Rectangle( 4, 5, "Rectangle #1")
        };

        System.Console.WriteLine("Shapes Collection");
        foreach (Shape s in shapes)
        {
            System.Console.WriteLine(s);
        }
    }
}
```



is keyword

- C# language provides the is keyword to determine whether two items are compatible.
- Unlike the as keyword, however, the is keyword returns false, rather than a null reference, if the types are incompatible.
- is keyword is used to check if the base class reference is of derived type specified

```
Figure one = new Square();
       if(one is Square)
              Console.WriteLine("It is a Square");
one = new Circle();
       if (one is Circle)
            Console.WriteLine("It is a Circle");
```

as Keyword (Casting)

- as keyword is used to obtain a reference to the more derived type.
- If the types are incompatible then the reference is set to null.

```
Circle c1 = new Circle();
        Figure f1 = c1 as Figure;
        if(f1 is Figure)
            Console.WriteLine("It is now of Figure Class");
        else
            Console.WriteLine("Not of Figure Class");
```

- casting will throw an exception if it fails.
- One important thing to bear in mind is that the methods that can be accessed using the instance are methods declared in its class.

What is the Difference?

- What is the difference between cast and as?

```
Circle c1 = new Circle();
Figure f1 = c1 as Figure;
            or
Figure f1 = (Figure) c1;
```

- For the above example both the statement work the same way. Let us take another example

```
Figure f = new Circle();
LargeCircle lc1 = f as LargeCircle;
Console.WriteLine(lc1==null?"Null":"Object");
```

- The statement above returns null whereas

```
Figure f = new Circle();
LargeCircle lc2 = (LargeCircle)f;
Console.WriteLine(lc1==null?"Null":"Object");
```

- Throws a InvalidCastException at runtime

What is an Interface ?

- An interface is a special kind of construct like class which contains just the declaration of methods (abstract methods).
- It defines a contract and any class (or struct) that implements this interface must provide implementation for all the methods declared inside the interface.
- Example of an interface that is .NET defined interfaces are IEnumerable, ICloneable etc.
- It can be a member of a namespace or a class .
- It can inherit from one or more base interfaces.
- It cannot be instantiated

Interfaces

- An interface is a point of contact with an object.
- All public members of a class can be said as an interface to it.
- An interface is a collection of related methods and properties without implementation.
- Interface = Pure Abstract Class (A class with all members declared as abstract)
- An interface represents a contract, in that a class that implements an interface must implement every aspect of that interface exactly as it is defined. The class and Interface are related with Implements relationship

Presented by

Ranjan Bhatnagar

Interface Syntax

```
modifier interface interface-name {
        members ;
}
```

- Modifiers allowed when the interface is declared outside a class are public and internal.
- Modifiers allowed are when the interface is declared inside a class new, internal, private, public, protected
- It is a compile-time error for interface member declarations to include any modifiers.

More on Interface

- An interface cannot be instantiated.
- For some reason if the implementing class cannot provide implementation to any of the interface member, the member in must still be declared in the class but it can be declared as abstract (also making the class as abstract).
- Every member of an interface is by default public and abstract. Interface cannot have field members. i.e. it can have only procedures (property / function / sub)
- A variable of type interface can refer to an object of any class implementing that interface.

Abstract v/s Interface

```
abstract class shapes
{
    abstract public void draw( ) ;
}
```

```
interface shapes
{
    void draw( ) ;
}
```

```
abstract class shapes
{
    public void draw( )
    {
    }
}
```

```
interface class shapes
{
    public void draw( )
    {
    }
}
```

```
abstract class shapes
{
    abstract public void draw( ) ;
    public void erase( )
    {
    }
}
```

```
interface class shapes
{
    interface public void draw( ) ;
    public void erase( )
    {
    }
}
```

Example: interface

```
public interface IShape
{
    void printSides(string s); // method
    int sides { get; set; } // properties
    int this[int index] { get; set; }
//indexer
}
IShape s= new IShape ();→ ERROR!
```

```
class Polygon : IShape
{
    private int n;
    private int[] Sides = new int[1];
    public void printSides(string s)
    {
        Console.WriteLine("{0} with number of sides {1}", s, n);
        for (int i = 0; i < Sides.Length; i++)
        {
            Console.WriteLine("Side {0} length {1} ", i, Sides[i]);
        }
    }
```

Example: Implementing Interface

```
public int sides
    {
        get { return n; }
        set
        {
            Sides = new int[value];
            n = value;
        }
    }
    public int this[int index]
    {
        get { return Sides[index]; }
        set
        { Sides[index] = value; }
    }
    static void Main()
    {
        Polygon square = new Polygon();
        square.sides = 4;
        square[0] = square[1] = square[2] = square[3] = 10;
        square.printSides("Square");
    }
}
```

```
C:\Windows\system32\cmd.exe
Square with number of sides 4
Side 0 length 10
Side 1 length 10
Side 2 length 10
Side 3 length 10
Press any key to continue . . .
```

Interface Members

- Interfaces can contain methods, properties, events, and indexers.
- All interface methods are implicitly public and abstract.
- An interface cannot contain constants, fields, operators, instance constructors, destructors, or types, nor can an interface contain static members of any kind.
- When class (struct) implements interfaces it is similar to inheriting from class.
- A class or struct can implement more than one interface unlike class inheritance.

Presented by

Ranjan Bhatnagar

Interface Example

```
using System;
interface IFigure
{
    int Dimension { get; set; }
    double Area();
    double Perimeter();
}
```

```
class Circle : IFigure
{
    private int _Radius;
    public int Dimension
    {
        get { return _Radius; }
        set { _Radius = value; }
    }
    public double Area()
    {
        return Math.PI * _Radius * _Radius;
    }
    public double Perimeter()
    {
        return 2 * Math.PI * _Radius;
    }
}
```

Interface Example Cont.

```
class Square : IFigure
{
    private int _Side;
    public int Dimension
    {
        get { return _Side; }
        set { _Side = value; }
    }
    public double Area()
    {
       return Math.PI * _Side * _Side;
    }
    public double Perimeter()
    {
       return 2 * Math.PI * _Side;
    }
}
```

Interface Example Cont.

```
class Program
{
    public static void Main(string[] args)
    {
        IFigure fig = null;
        Console.WriteLine("Enter 'C' for Circle or 'S' for Square");
        string ch = Console.ReadLine();
        if (ch == "S")
            fig = new Square();
        else if (ch == "C")
            fig = new Circle();
        fig.Dimension = 10;
        Console.WriteLine(fig.Area());
        Console.WriteLine(fig.Perimeter());
    }
}
```

Differences

- Interface members are defined like class members except for a few important differences:
  - No access modifiers (public, private, protected, or internal) are allowed—all interface members are implicitly public.
  - Interface members can't contain code bodies.
  - Interfaces can't define field members.
  - Interface members can't be defined using the keywords static, virtual, abstract, or sealed.
  - Type definition members are forbidden.

Explicit Interface Implementation

- Explicit interface member implementation allows access to the interface declared method only through interface reference.
- This helps in overcoming the method name clashes if a class inherits from
  - two (or more) interfaces
  - or an interface and a class

Explicit Interface Implementation

```
interface IControl
{
    void Paint();
}
interface ISurface
{
    void Paint();
}
class SampleClass : IControl, ISurface
{
    // Both ISurface.Paint and IControl.Paint call this method.
    public void Paint()
    {
        Console.WriteLine("Paint method in SampleClass");
    }
}
```

Presented by

Ranjan Bhatnagar

Explicit Interface Implementation

```
using System;
class Test
{
    static void Main()
    {
        SampleClass sc = new SampleClass();
        IControl ctrl = (IControl)sc;
        ISurface srfc = (ISurface)sc;

        // The following lines all call the same method.
        sc.Paint();
        ctrl.Paint();
        srfc.Paint();
    }
}
```

Solution

```
public class SampleClass : IControl, ISurface
{
    void IControl.Paint()
    {
        System.Console.WriteLine("IControl.Paint");
    }
    void ISurface.Paint()
    {
        System.Console.WriteLine("ISurface.Paint");
    }
}
```

```
static void Main()
{
       // Call the Paint methods from Main.
       SampleClass obj = new SampleClass();
       //obj.Paint();  // Compiler error.
       IControl c = (IControl)obj;
       c.Paint();  // Calls IControl.Paint on SampleClass.
       ISurface s = (ISurface)obj;
       s.Paint(); // Calls ISurface.Paint on SampleClass.
}
```

Abstract class v/s. Interface

- An abstract class can have abstract members as well non abstract members. But in an interface all the members are implicitly abstract and all the members of the interface must override to its derived class.
- Defining an abstract class with abstract members has the same effect to defining an interface.
- The members of the interface are public with no implementation. Abstract classes can have protected parts, static methods, etc.
- A class can inherit one or more interfaces, but only one abstract class.

Abstract class v/s Interface

- Abstract classes can add more functionality without destroying the child classes that were using the old version. In an interface, creation of additional functions will have an effect on its child classes, due to the necessary implementation of interface methods to classes.
- The selection of interface or abstract class depends on the need and design of your project. You can make an abstract class, interface or combination of both depending on your needs.

Interface and new keyword

- You can, however, define members using the new keyword if you want to hide members inherited from base interfaces:

```
interface IMyBaseInterface
{
    void DoSomething();
}
interface IMyDerivedInterface : IMyBaseInterface
{
    new void DoSomething();
}
```

- This works exactly the same way as hiding inherited class members.

Casting

- Any reference can be converted to interface type through casting.
- If casting fails, a runtime error is generated.
- Note that the two unrelated references of class type is a compilation error while casting of interface type to class type (or vice versa) is not a compilation error even if class and interface type are unrelated.
- But without a cast unrelated conversions are always incorrect.

```
Ishape isquare= new Polygon(); // automatic conversion
..
Polygon p= (Polygon) isquare; // casting
```

Presented by

Ranjan Bhatnagar

is and as keywords

- Usage of is and as for interface inheritance is similar to their usage for class inheritance.

```
Console.WriteLine(square is IShape);
IShape line = new Polygon();// True
Console.Write(line is IShape); //True
Console.Write(line is Polygon); //True
Console.Write(line is IComparable); //False
```

- Polygon square= new Ploygon();

```
IShape sq = square as IShape;
```

Cloning

- The MemberWiseClone() method of the System.Object class does a shallow copy of the current object.
- This version works ok if the object does not contain a reference within itself.
- If the object contains references then assignment of reference fields does not result in a copy!
- That is the reason why MemberWiseClone() is declared as protected.

ICloneable

- A class can avail or override the **MemberWiseClone**() method of the **Object** class.
- Since this is a **protected** method it cannot be exposed to the other classes.
- The **ICloneable** interface has method that is used to expose **MemberWiseClone**() .
- This lets other classes know that cloning is possible for this class.
- Method in **ICloneable**
  - Object Clone()

Example- ICloneable

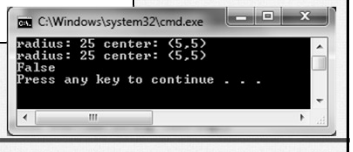
```
using System;
class Point : ICloneable
{
    private int x, y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public override string ToString()
    {
        return "(" + x + "," + y + ")";
    }
    public object Clone()
    {
        return this.MemberwiseClone();
    }
}
```

Example- ICloneable cont.

```
class Circle : ICloneable
{
    uint radius;
    Point center;
    public Circle() { }
    public Circle(uint r, Point p)
    {
        radius = r;
        center = (Point)p.Clone();
    }
    public object Clone()
    {
        Circle c = (Circle)this.MemberwiseClone();
        c.center = (Point)center.Clone();
        return c;
    }
    public override string ToString()
    {
        return "radius: " + radius + " center: " + center;
    }
```

Example- ICloneable cont.

```
public static void Main()
    {
        Point p = new Point(5, 5);
        Circle c1 = new Circle(25, p);
        Console.WriteLine(c1);
        Circle c2 = new Circle();
        if (c2 is ICloneable)
        {
            c2 = (Circle)c1.Clone();
            Console.WriteLine(c2);
            Console.WriteLine(c2 == c1);
        }
    }
}
```



Presented by

Ranjan Bhatnagar

Namespace

- Namespaces have the following properties:
  - They organize large code projects.
  - They are delimited by using the . operator.
  - The using directive obviates the requirement to specify the name of the namespace for every class.
  - The global namespace is the "root" namespace: global::System will always refer to the .NET Framework namespace System.

Declaring Namespace

- Namespaces are heavily used within C# programs in two ways. Firstly, the .NET Framework classes use namespaces to organize its many classes.
- Secondly, declaring your own namespaces can help control the scope of class and method names in larger programming projects.
- Use the **namespace** keyword to declare a namespace as:

```
namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            System.Console.WriteLine(
              "SampleMethod inside SampleNamespace");
        }
    }
}
```

What can we have?

- Within a namespace, you can declare one or more of the following types:
  - another namespace
  - class
  - interface
  - struct
  - enum
  - delegate
- Whether or not you explicitly declare a namespace in a C# source file, the compiler adds a default namespace.
- This unnamed namespace, sometimes referred to as the global namespace, is present in every file.

Namespaces

```
namespace talent
{
    class math
    {
    }
}
namespace Simple
{
    class Class1
    {
        static void Main ( string[ ] args )
        {
            talent.math  m1 = new talent.math( ) ;
        }
    }
}
```

```
using  talent ;
namespace Simple
{
    class Class1
    {
        static void Main ( string[ ] args )
        {
            math m1 = new math( ) ;
        }
    }
}
```

Nested Namespaces

```
namespace College.Library
{
    class Book
    {
    }
    class Magazine
    {
    }
}
```

Nested namespaces

```
namespace College
{
    namespace Library
    {
        class Book
        {
        }
        class Magazine
        {
        }
    }
}
```

```
College.Library.Book  b1 = new College.Library.Book( ) ;
```

```
using College.Library ;
Book b1 = new Book( ) ;
```

Indexers

- Like [ ] can be used for an array, [ ] can be used for any user defined class (or struct) as well, acting like "virtual arrays".
- Instances of that class can be accessed using the [ ] array access operator.
- For classes that encapsulate array- or collection-like functionality, using an indexer allows the users of that class to use the array syntax to access the class.

```
Syntax:
public return-type this[int index-position]{
get { return some-value}
set { some-value= value}
}
```

- An indexer value is not classified as a variable; therefore, you cannot pass an indexer value as a ref or an out parameter.

Presented by Ranjan Bhatnagar

About Indexer

- Indexers enable objects to be indexed in a similar manner to arrays.
- A get accessor returns a value.
- A set accessor assigns a value.
- The this keyword is used to define the indexers.
- The value keyword is used to define the value being assigned by the set indexer.
- Indexers do not have to be indexed by an integer value; it is up to you how to define the specific look-up mechanism.
- Indexers can be overloaded.
- Indexers can have more than one formal parameter, for example, when accessing a two-dimensional array.

Indexer Example

```
using System;
namespace sample
{
    public class array
    {
        float[] arr = new float[] { 12.5f,
                34.3f, 5.2f, 6.1f, 7.5f,
                        88.8f, 22.9f };
        public float this[int index]
        {
            get
            {
                return arr[index];
            }
            set
            {
                arr[index] = value;
            }
        }
    }
```

```
    class Class1
    {
        static void Main
              (string[] args)
        {
            array a;
            a = new array();
            a[3] = 43.2f;
            Console.WriteLine
                      (a[3]);
        }
    }
}
```

2-D Indexer

```
using System;
namespace sample
{
    public class array
    {
        int[,] arr2 = new int[,]
               { { 12, 34, 5, 6 },
               { 7, 88, 22, 2 } };
        public int this [int
               index1, int index2]
        {
            get
            {
                return arr2[
               index1, index2];
            }
            set
            {
                arr2[index1,
                index2] = value;
            }
        }
    }
```

```
class Class1
    {
        static void Main
              (string[] args)
        {
            array a;
            a = new array();
            Console.WriteLine
        (a[1, 0] + "\t" + a[0, 2]);
            a[1, 0] = 43;
            a[0, 2] = 89;
            Console.WriteLine
        (a[1, 0] + "\t" + a[0, 2]);
        }
    }
}
```

String Indexer

```
using System;
public class Capital
{
    string[] states = { "rj", "up",  "mp", "ap", "kl"  };
    string[] capitals = { "Jaipur", "Lucknow", "Bhopal",
                          "Hyderabad", "Trivendram" };
    public string this[string state]
    {
        get
        {
            int i = 0;
            while ((i < states.Length) &&
               (state.ToLower() != states[i]))
                ++i;
            return (i==capitals.Length)?"No Such State":capitals [i];
        }
```

String Indexer

```
        set
        {
            int i = 0;
            while ((i < states.Length) &&
               (state.ToLower() != states[i]))
                ++i;

            if (i != states.Length)
                capitals[i] = value;
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        Capital c1 = new Capital();
        //Pass state code and get capital
        Console.WriteLine(c1["mp"]);
    }
}
```

Properties v/s Indexers

Property	Indexer
Allows methods to be called as if they were public data members.	Allows elements of an internal collection of an object to be accessed by using array notation on the object itself.
Accessed through a simple name.	Accessed through an index.
Can be a static or an instance member.	Must be an instance member.
A get accessor of a property has no parameters.	A get accessor of an indexer has the same formal parameter list as the indexer.
A set accessor of a property contains the implicit value parameter.	A set accessor of an indexer has the same formal parameter list as the indexer, and also to thevalue parameter.
Supports shortened syntax with Auto-Implemented Properties	Does not support shortened syntax.

Presented by

Ranjan Bhatnagar

Thanks