C# Programming With .NET
(06CS/IS761)

Chapter wise questions appeared in previous years:

	UNIT VI: INTERFACES AND COLLECTIONS	Markes & Year Appeared

1	**What is an interface? Explain with an example implementation of Interfaces in C#.**
	• An *interface is* a named set of semantically related *abstract members.*
	• An interface expresses a *behavior that a given class or structure may choose to implement.*
	• *At a syntactic level, an interface is defined using the C# "Interface" keyword.*
	• Interfaces are specifications defining the type of behavior a class must implement.
	• Interfaces are the contracts; a class uses to allow other classes to interact with it in a well defined and anticipated manner.
	• Interfaces never specify a base class (not even System.Object).
	//A given class may implement as many interfaces as necessary, but may have exactly 1(one) base class
	public class Hexagon: shape, Ipointy {
	public Hexagon() { }
	public Hexagon(string name) ; base(name){}
	public override void Draw() {
	Console.WriteLine("Drawing {0} the Hexagon",PetName);
	}
	//Ipointy Implementation
	public byte GetNumberOfPoints()
	{ return 6; }
	}
	public class Triangle: Shape, Ipointy {
	public Triangle() { }
	public Triangle(string name):base(name){}
	public void override Draw() {
	Console.WriteLine("Drawing {0} the Triangle", PetName);
	}
	//Ipointy Implementation
	public byte GetNumberOfPoints()
	{ return 3; }
	}
		6M

2	**Three different ways of obtaining interface references.**
	Explicit casting, is a relation, as a relation.
	Interfaces as explicit casting:
	• **Consider the defination of IDraw3D, you are forced to name the method Draw3D() in order to avoid clashing with the abstract Draw() method defined in shapes base class:**
	//3D drawing behavior.
	public interface IDraw3D
	{
	void Draw3D();
	}
	• While there is nothing horribly wrong with this interface definition, a more natural method name would simply be Draw():
	//3D Drawing behavior
	public interface IDraw3D
		9M

```
    {
        void Draw();
    }
```

- If you were to create a new class that derives from Shape and implements IDraw3D.
- Assume you had defined the following new class Line: // Problems…….

```
public class Line: Shape, IDraw3D
{
        public override void Draw()
        {
                Console.WriteLine("Drawing a Line");
        }
}
```

Obtaining Interface : The 'as' Keyword:

- The second way you can determine whether a given type supports an interface is to make use of the 'as' keyword.
- If the object can be treated as the specified interface, you are returned a reference to the interface in question. If not, you receive a null reference.
- Therefore, be sure to check against a null value before proceeding:
- Note: Notice that when you make use of the as keyword, you have no need to make use of try/catch logic, given that if the reference is not null, you know you are calling on a valid interface reference.

```
static void Main(string[] args)
{
...
// Can we treat hex2 as IPointy?
Hexagon hex2 = new Hexagon("Peter");
IPointy itfPt2 = hex2 as IPointy;
if(itfPt2 != null)
Console.WriteLine("Points: {0}", itfPt2.Points);
else
Console.WriteLine("OOPS! Not pointy...");
Console.ReadLine();
}
```

Obtaining Interface References: The 'is' Keyword:

- You may also check for an implemented interface using the 'is' keyword.
- If the object in question is not compatible with the specified interface, you are returned the value false.
- On the other hand, if the type is compatible with the interface in question, you can safely call the members without needing to make use of try/catch logic.
- To illustrate, assume we have an array of Shape types containing some members that implement IPointy.
- Notice how we are able to determine which item in the array supports this interface using the is keyword, as shown in this retrofitted Main() method:

```
static void Main(string[] args)
{
...
// Catch a possible InvalidCastException.
Circle c = new Circle("Lisa");
IPointy itfPt = null;
try
{
itfPt = (IPointy)c;
Console.WriteLine(itfPt.Points);
```

```
}
catch (InvalidCastException e)
{ Console.WriteLine(e.Message); }
Console.ReadLine();
}
```

- While you could make use of try/catch logic and hope for the best, it would be ideal to determine which interfaces are supported before invoking the interface members in the first place.

3	**Explain explicit interfacing in detail.**	7M
4	**Interfaces as Polymorphic Agents.**	5M
5	**Explain in detail the IConvertable Interfaces along with its different supporting method types.**	8M

- Interfaces are used extensively throught the .NET base class libraries.
- IConvertable type allows you to dynamically concert between data types using interface-based programming technique.
- Using this interface method you cancast between types on a fly using language- agnostic terms.

```
Public interface IConvertable
{
    TypeCode Get TypeCode();
     bool ToBoolean(IFormatProvider provider);
     byte ToByte(IFormatProvider provider);
     char ToChar(IFormatProvider provider);
     DateTime ToDateTime(IFormatProvider provider);
     Decimal ToDecimal(IFormatProvider provider);
     double ToDouble(IFormatProvider provider);
     short ToInt16(IFormatProvider provider);
     int ToInt32(IFormatProvider provider);
     long ToInt64(IFormatProvider provider) ;
     .
     .
     .
     .
     UInt64 ToUInt64(IFormatProvider provider);
     }
```

6	**Building IEnumerable/ IEnumerator type with example.**	5M

Consider an example below to explain the use of IEnumerable and IEnumerator:

```
//Car is a Đontainer of Đar oďjeĐts.....
Public class Car
{
  Private Car[] CarArray;
//Create some car objects upon startup……..
  Public Cars()
   {
      CarArray = new[4];
       CarArray[0] = new Car(“FeeFee”, 200, 0);
       CarArray[1] = new Car(“Clunker”, 300, 0);
       CarArray[2] = new Car(“Zippy”, 30,0);
  }
}
```

Below method is defined by the IEnumerable interface type:

```
Public interface IEnumerable
{
```

```
    IEnumerable GetEnumerator();
}
//IEnumerable defines a single method
Public IEnumerable GetEnumerator ()
{
          ///Ok, now what……?
}
```

Now, Given that IEnumerable.GetEnumerator() returens an IEnumerator interface, you may update the cars type as shown below:

```
//Getting closer….
Public class cars: IEnumerable, IEnumerator
{
  ……
  //Implementing an IEnumerable……
    Public IEnumerator GetEnumerator()
      {
          Return(IEnumerator) this;
      }
}
```

7 **Building ICloneable interface type.** 6M

Building Cloneable objects (ICloneable):

- The system.Object defines a member named MemberWiseClone().
- This is used to make a shallow copy of an object instance.
- A given object instance can call this method itself during cloning as it is protected type.

Consider an example:

```
//The classic point example:
public class Point
{
       //public for easy access.
       public int x, y;
       public Point();
       public point(int x, int y)
         {
            this.x = x; this.y = y;
         }
//Override Object.ToString()……..
public override string ToString()
     { return "x: " + x + y : " + y; }
}
//To reference to same object……
Console.WriteLine("Assigning points.");
Point p1 = new Point(50, 50);
Point p2 = p1;
P2.x = 0;
```

If you wish to equip your custom types to support the ability to return an identical copy of itself to the caller, It can be implemented by standard ICloneable interface.

```
public interface ICloneable
{
        object Clone();
}
//The point classs supports deep copy of semantics ICloneable……
public class Point : ICloneable
{
    public int x, y;
```

```
        public Point() {}
        public Point(int x, int y) { this.x; this.y}
    public object Clone()
    {
        return new Point(this.x, this.y);
    }
    public override string ToString()
    {
        return "X: " + x + " Y: " +y;
    }
}
```

8 **Explain IComparable interface object types.** 6M

IComparable interface specifies a behavior that allows an object to be sorted based on some internal key.

```
//This interface allows an object to specify its relationship between other like objecs……
interface IComparable
{
    int CompareTo(object o);
}
```

Let's assume you have updated the Car class to maintain an internal ID number. Object users might create an array of Car types as follows:

```
//Make an array of Car types…….
Car[] myAutos = new Car[5];
myAutos[0] = new Car(123, "Rusty");
myAutos[1] = new Car(6, "Mary");
myAutos[2] = new Car(83."Chucky");
//Sort my Cars?
Array.Sort(myAutos); //Nope, not yet sorry!
```

If we try to test this, we will find an ArgumentException exception by the RunTime, with the following message: "One object must implement IComparable".

```
//The iteration of the Car can be ordered based on the CAR ID.
public class Car: IComparable
{
    …….
    //Icomparable implementation
      int IComparable.CompareTo(object o)
     {
       Car temp = (Car)o;
       if(this.CarId > temp.CarId)
            return 1; if(this.CarID
       < temp.CarID)
            return -1;
       else
           return 0;
   }
}
```

9 **List and explain in detail the different System.Colletctions Interface type.** 7M

System.Collections Interface	Meaning in Life
ICollection	Defines general characteristics (e.g., size, enumeration, thread safety) for all nongeneric collection types.
IComparer	Allows two objects to be compared.

IDictionary	Allows a nongeneric collection object to represent its contents using name/value pairs.
IDictionaryEnumerator	Enumerates the contents of a type supporting IDictionary.
IEnumerable	Returns the IEnumerator interface for a given object.
IEnumerator	Enables foreach style iteration of subtypes.
IHashCodeProvider	Returns the hash code for the implementing type using a customized hash algorithm.
IList	Provides behavior to add, remove, and index items in a list of objects. Also, this interface defines members to determine whether the implementing collection type is read-only and/or a fixed-size container.

10 **The class types of System.Collections:** 6M

System.Collections Class	**Meaning in Life Key**	**Implemented Interfaces**
ArrayList	Represents a dynamically sized array of objects.	IEnumerable, and ICloneable
Hashtable	Represents a collection of objects identified by a numerical key. Custom types stored in a Hashtable should always override System.	IDictionary, ICollection, Enumerable, and Hashtable ICloneable, Object.GetHashCode().
Queue	Represents a standard first in, first-out (FIFO) queue.	ICollection, ICloneable, IEnumerable
SortedList	Like a dictionary; however the elements can also be accessed by ordinal position (e.g., index).	IDictionary, ICollection, IEnumerable, ICloneable
Stack	A last-in, first-out (LIFO) queue providing push and pop (and peek) functionality.	ICollection, ICloneable, IEnumerable