Session: 8

Abstract Classes and Interfaces

Objectives

- Define and describe abstract classes
- Explain interfaces
- Compare abstract classes and interfaces

Abstract Classes

- Is a class specifically to be used as a incomplete base class.
- cannot be instantiated, but be implemented or derived.
- may contain one or more of the following:
  - normal data member(s) / method(s)
  - abstract method(s)
- Declared by keyword abstract
- Syntax:

```
public abstract class <ClassName>
{
  <accModifier> abstract <returnType> <MethodName>(pars);
}
```

Abstract Implementation

- The following code declares and implements an abstract class:

```
abstract class Animal {
   public void Eat()  {
       Console.WriteLine("eats food in order to survive");
   }
   public abstract void AnimalSound();
}

class Lion : Animal {
   public override void AnimalSound()  {
       Console.WriteLine("Lion roars");
   }
}
```

- contain only abstract members
- cannot be instantiated but can be inherited by classes or other interfaces.
- declared by the keyword **`interface.`**
- In C#, by default, all members declared in an interface have **`public`** access modifier.
- The following figure displays an example of an interface:

Animal Abstract Class

```
Eat()
{
  "Every animal eats food";
}
Habitat();
AnimalSound();
```

IAnimalInterface

```
Eat(); //No Body
Habitat();
AnimalSound();
```

Implementing an Interface

```
interface Ianimal {  void Habitat(); }

class Dog : Ianimal {
   public void Habitat() {
      Console.WriteLine("Can be housed with human
      beings");
   }

   static void Main(string[] args) {
      Dog objDog = new Dog();
      Console.WriteLine(objDog.GetType().Name);
      objDog.Habitat();
   }
}
```

Output

```
Dog
Can be housed with human beings
```

```
interface ITerrestrialAnimal { void Eat(); }
interface IMarineAnimal{ void Swim(); }
class Crocodile : ITerrestrialAnimal, IMarineAnimal{
  public void Eat() {
      Console.WriteLine("The Crocodile eats flesh");
  }
  public void Swim() {
      Console.WriteLine("The Crocodile can swim 4 times faster
                         than an Olympic swimmer");
  }
  static void Main(string[] args) {
      Crocodile o = new Crocodile();
      o.Eat();
      o.Swim();
  }
}
```

Output

```
The Crocodile eats flesh
The Crocodile can swim four times faster than an
Olympic swimmer
```

Interface Inheritance

- The following syntax is used to inherit an interface:

Syntax

```
interface <InterfaceName> : <Inherited_InterfaceName>
{
        // method declaration;
}
```

- where,
  - **InterfaceName:** name of the interface that inherits another interface.
  - **Inherited_InterfaceName:** name of the inherited interface.

Interface Inheritance

Snippet

```
interface Ianimal { void Drink();  }
interface Icarnivorous {  void Eat(); }
interface IReptile:IAnimal, Icarnivorous { void Habitat(); }
class Crocodile : Ireptile  {
   public void Drink()  {
    Console.WriteLine("Drinks fresh water");
   }
   public void Habitat() {
     Console.WriteLine("Can stay in Water and Land");
   }
   public void Eat() {
      Console.WriteLine("Eats Flesh");
   }
   static void Main(string[] args) {
      Crocodile o = new Crocodile();
      Console.WriteLine(o.GetType().Name);
      o.Habitat();
      o.Eat();
      o.Drink();
   }
}
```

Output

```
Crocodile
Can stay in Water and Land
Eats Flesh
Drinks fresh water
```

The `is` and `as` Operators in Interfaces 1-2

- The **`is`** and **`as`** operators when used with interfaces, verify whether the specified interface is implemented or not.

is Operator

Checks the compatibility between two types

returns a boolean value based on the check operation performed.

as Operator

performs conversion between compatible types

returns null if the two types are not compatible with each other.

The is and as Operators in Interfaces 2-2

```
interface ICalculate { double Area();  }

class Rectangle : ICalculate{
   float _length, float _breadth;
   public Rectangle(float valOne, float valTwo) {
      _length = valOne;
      _breadth = valTwo;
   }
   public double Area() { return _length * _breadth; }
   static void Main(string[] args) {
      Rectangle objRectangle = new Rectangle(10.2F, 20.3F);
      if (objRectangle is ICalculate) {
            Console.WriteLine("Area : {0:F2}" ,
            objRectangle.Area());
      } else {
            Console.WriteLine("Interface method not implemented");
      }
    }
}
```

Differences Between an Abstract Class and an Interface

- Abstract classes and interfaces are similar because both contain abstract methods that are implemented by the inheriting class.
- However, there are certain differences between them as shown in the following table:

Abstract Classes	Interfaces
can inherit a class and multiple interfaces.	can inherit multiple interfaces but cannot inherit a class.
can have methods with a body.	cannot have methods with a body.
is implemented using the override keyword.	is implemented without using the override keyword.
is a better option when you need to implement common methods and declare common abstract methods.	is a better option when you need to declare only abstract methods.
can declare constructors and destructors.	cannot declare constructors or destructors.

Recommendations for Using Abstract Classes and Interfaces

Abstract class

- create reusable programs and maintain multiple versions of these programs
- helps to maintain the version of the programs in a simple manner.
- must exist a relationship between the abstract class and the classes that inherit the abstract class.

Interface

- create different methods that are useful for different types of objects
- are suitable for implementing similar functionalities in dissimilar classes.
- cannot be changed once they are created.
- A new interface needs to be created to create a new version of the existing interface

Summary

- An abstract class can be referred to as an incomplete base class and can implement methods that are similar for all the subclasses.
- IntelliSense provides access to member variables, functions, and methods of an object or a class.
- When implementing an interface in a class, you need to implement all the abstract methods declared in the interface.
- A class implementing multiple interfaces has to implement all abstract methods declared in the interfaces.
- A class has to explicitly implement multiple interfaces if these interfaces have methods with identical names.
- Re-implementation occurs when the method declared in the interface is implemented in a class using the virtual keyword and this virtual method is then overridden in the derived class.
- The is operator is used to check the compatibility between two types or classes and as operator returns null if the two types or classes are not compatible with each other.