عنوان مضمون

Visual Programming-I

توسط : صفری

خزان1397

C# Object oriented features

C# - Interfaces

C# - Interfaces

An interface is defined as a **syntactical contract** that all the classes inheriting the interface should follow.

The interface defines the **'what'** part of the syntactical contract and the deriving classes define the **'how'** part of the syntactical contract.

C# - Interfaces

- Interfaces define properties, methods, and events, which are the members of the interface.
- Interfaces contain only the declaration of the members. It is the responsibility of the deriving class to define the members.
- It often helps **in providing a standard structure that the deriving classes would follow.**
- Abstract classes to some extent serve the same purpose, however, they are mostly used when only few methods are to be declared by the base class and the deriving class implements the functionalities.

Declaring Interfaces

Interfaces are declared using the **interface keyword**.

It is similar to class declaration.

Interface statements are **public by default**.

Following is an example of an interface declaration:

```
public interface Itransactions
 {
        void showTransaction();
        double getAmount();
}
```

```
public interface ITransactions {
    void showTransaction();
    double getAmount();
 }
 public class Transaction : ITransactions {
    private string date;
    private double amount;
    public Transaction() {
      date = " ";
      amount = 0.0;
    }
    public Transaction( string d, double a) {
      date = d;
      amount = a;
    }
    public double getAmount() {
      return amount;
    }
    public void showTransaction() {
      Console.WriteLine("Date: {0}", date);
      Console.WriteLine("Amount: {0}",    getAmount());
    }
 }
```

```
static void Main(string[] args) {
      Transaction t1 = new Transaction("8/10/2012", 78900.00);
      Transaction t2 = new Transaction("9/10/2012", 45900.00);

      t1.showTransaction();
      t2.showTransaction();
      Console.ReadKey();
    }
```

C# - Encapsulation

C# - Encapsulation

- Encapsulation, in object oriented programming methodology, prevents access to implementation details.
- Abstraction and encapsulation are related features in object oriented programming. Abstraction allows making relevant information visible and encapsulation enables a programmer to *implement the desired level of abstraction*.
- Encapsulation is implemented by using **access specifiers.**

C# - Encapsulation

- An **access specifier** defines the scope and visibility of a class member. C# supports the following access specifiers −
- Public
- Private
- Protected
- Internal
- Protected internal

Public Access Specifier

Public access specifier allows a class to expose its member variables and member functions to other functions and objects.

Any public member can be accessed from outside the class.

```
class Rectangle
{   public double length;
    public double width;
    public void Display()
     {  Console.WriteLine("Length: {0}", length);
       Console.WriteLine("Width: {0}", width);
     }
 }
 class ExecuteRectangle {
    static void Main(string[] args) {
       Rectangle r = new Rectangle();
       r.length = 4.5;
       r.width = 3.5;
       r.Display();
    }
 }
```

Private Access Specifier

- Private access specifier allows a class to hide its member variables and member functions from other functions and objects.
- Only functions of the same class can access its private members. Even an instance of a class cannot access its private members.

```
class Rectangle
{    private double length;
     private double width;
     public void Acceptdetails()
     {   Console.WriteLine("Enter Length: ");
          length = Convert.ToDouble(Console.ReadLine());
          Console.WriteLine("Enter Width: ");
          width = Convert.ToDouble(Console.ReadLine());
     }
      public void Display()
      {    Console.WriteLine("Length: {0}", length);
        Console.WriteLine("Width: {0}", width);
      }
 }
class ExecuteRectangle {
     static void Main(string[] args) {
        Rectangle r = new Rectangle();
        r.Acceptdetails();
        r.Display();
      }
  }
```

Protected Access Specifier

- Protected access specifier allows a child class to access the member variables and member functions of its base class.
- This way it helps in implementing inheritance. We will discuss this in more details in the inheritance chapter.

Internal Access Specifier

- Internal access specifier allows a class to expose its member variables and member functions to other functions and objects **in the current file or assembly** but not from another assembly.
- In other words, any member with internal access specifier can be accessed from any class or method defined **within the application** in which the member is defined.

```
class Rectangle
{     internal double length;
      internal double width;
      public void Display()
      {  Console.WriteLine("Length: {0}", length);
         Console.WriteLine("Width: {0}", width);
         Console.WriteLine("Area: {0}", GetArea());
      }
}
   class ExecuteRectangle {
      static void Main(string[] args)
     {    Rectangle r = new Rectangle();
           r.length = 4.5;
           r.width = 3.5;
           r.Display();
     }
   }
}
```

Suppose you have this file.dll (mytest.dll say):

```
public class calculator
{       internal int a;
        public int h;
}
```

Add this file to project Select Project ⇨Add Reference, or select the same option after rightclicking References in the Solution Explorer window.

```
class Program
    {   static void Main(string[] args)
        {   calculator ss = new calculator();
            ss.h = 4;       //accessible
            ss.a = 5;       //inaccessible
        }
    }
```

Protected Internal Access Specifier

- The protected internal access specifier allows a class to hide its member variables and member functions from other class objects and functions, except a child class within the same application.
- This is also used while implementing inheritance.