عنوان مضمون

Visual Programming-I

توسط : صفری

خزان1397

C# Object oriented features

WHAT IS OBJECT-ORIENTED PROGRAMMING?

- The type of programming you have seen so far is known as ***procedural programming*,**

  which often results in so-called monolithic applications, meaning all functionality is contained in a few modules of code (often just one).

With OOP techniques, you often use many more modules of code, with each offering specific functionality.

  Also, each module can be isolated or even completely independent of the others. This modular method of programming gives you much more versatility and provides more opportunity for code reuse.

WHAT IS OBJECT-ORIENTED PROGRAMMING?

- To illustrate this further, imagine that a high-performance application on your computer is a top of- the-range race car.
- **Written with traditional programming techniques**, this sports car is basically a single unit. If you want to improve this car, then you have to replace the whole unit by sending it back to the manufacturer and getting their expert mechanics to upgrade it, or by buying a new one.
- **If OOP techniques are used**, however, you can simply buy a new engine from the manufacturer and follow their instructions to replace it yourself, rather than taking a hacksaw to the bodywork.

What Is an Object?

- An ***object*** is a **building block** of an OOP application. This building block encapsulates part of the application, which can be a **process**, a **chunk of data**, or a more **abstract entity**.
- Objects in C# are created from **types**, just like the variables you've seen already.
- The type of an object is known by a special name in OOP, **its *class***.
- You can use class definitions to *instantiate* objects, which means creating a real, named *instance* of a class. The phrases *instance of a class* and *object* mean the same thing here; but *class* and *object* mean fundamentally different things.

Defining a Class

A class definition starts with the keyword class followed by the class name; and the class body enclosed by a pair of curly braces. Following is the general form of a class definition:

```
class  class_name
{
  <access specifier> <data type> variable1;
  <access specifier> <data type> variable2;
  ...
<access specifier> <return type> method1(parameter_list)
  {
    // method body
  }
}
```

```
using System;
namespace BoxApplication
{
   class Box
   {
      public double length;   // Length of a box
      public double breadth;  // Breadth of a box
      public double height;   // Height of a box


      public double getVolume()
       {
         double Volume=length * breadth * height;
         return Volume;
       }

    }
```

```
class program
   {
      static void Main(string[] args)
      {
         Box Box1 = new Box();

         Box1.height = 5.0;
         Box1.length = 6.0;
         Box1.breadth = 7.0;

         double volume = 0.0;
         volume = Box1.getVolume();
       Console.WriteLine("Volume of Box1 : {0}" ,volume);
       Console.ReadKey();
     }
   }
}
```

The Life Cycle of an Object

- Every object has a clearly defined life cycle. Apart from the normal state of “being in use,” this life cycle includes two important stages:

➤ **Construction**—When an object is first instantiated it needs to be initialized. This initialization is known as *construction* and is carried out by a constructor function, often referred to simply as a *constructor* for convenience.

➤ **Destruction**—When an object is destroyed, there are often some clean-up tasks to perform, such as freeing memory. This is the job of a destructor function, also known as a *destructor*.

Constructors

- All class definitions contain at least one constructor. These constructors can include a ***default constructor,***

    which is a parameter-less method with the same name as the class itself.

A class definition might also include several constructor methods with parameters, known **as *nondefault constructors*.**

- Code external to a class can't instantiate an object using a private constructor; it must use a public constructor.

C# Constructors

- A class **constructor** is a special member method of a class that is executed whenever we create new objects of that class.
- A constructor has exactly the same name as that of class and it does not have any return type.

```
class class-name
  {
    public class-name()
    {
      Console.WriteLine("Object is being created");
    }
  }
```

```
class Line
  {
     private double length;
     public Line()
     {
       Console.WriteLine("Object isbeingcreated");
     }

     public void setLength( double len )
     {
       length = len;
     }
 }
```

```
 class program
{
static void Main(string[] args)
     {
        Line line = new Line();
        line.setLength(6.0);
        Console.WriteLine("Length of line : {0}",
line.getLength());
        Console.ReadKey();
     }
}
```

- A **default constructor** does not have any parameter but if you need, a constructor can have parameters. Such constructors are called ***nondefault constructors***. This technique helps you to assign initial value to an object at the time of its creation

```
class Line
  {
     private double length;
     public Line(double len)  //Parameterized constructor
     {
        Console.WriteLine("Object is being created, length =
{0}", len);
        length = len;
     }
     public void setLength( double len )
     {
        length = len;
     }
     public double getLength()
     {
        return length;
     }
```

```
static void Main(string[] args)
      {
         Line line = new Line(10.0);
         Console.WriteLine("Length of line : {0}", line.getLength());

         // set line length
         line.setLength(6.0);
         Console.WriteLine("Length of line : {0}", line.getLength());
         Console.ReadKey();
      }
   }
```

Destructors

- A **destructor** is a special member method of a class that is executed whenever an object of its class goes out of scope. A **destructor** has exactly the same name as that of the class with a prefixed tilde (~) and it can neither return a value nor can it take any parameters.
- Destructor can be very useful for releasing memory resources before exiting the program. Destructors cannot be inherited or overloaded.

```
class Line
  {
    private double length;
    ~Line() //destructor
    {
        Console.WriteLine("Object is being  deleted");
    }
    public void setLength( double len )
    {
      length = len;
    }
    public double getLength()
     {
      return  length;
     }
```

```
  static void Main(string[] args)
     {
        Line line = new Line();

        // set line length
        line.setLength(6.0);
        Console.WriteLine("Length of line : {0}",
line.getLength());
     }
   }
```

Classes and Structures have the following basic differences:

- classes are reference types and structs are value types
- structures cannot have default constructor and Destructors
- structures do not support inheritance

classes are reference types and structs are value types

```
class Program
  {
     struct man
     {
        public int age;
     }
     static void Main(string[] args)
     {
        man man1 = new man();
        man1.age= 16;
        man man2 = new man();
        man2=man1;
        man2.age = 20;
        Console.WriteLine(man1.age);
        Console.WriteLine(man2.age);
        Console.ReadLine();
     }
  }
```

```
class man
  {
     public int age;
  }
  class Program
  {
     static void Main(string[] args)
     {
        man man1 = new man();
        man1.age= 16;
        man man2 = new man();
        man2=man1;
        man2.age = 20;
        Console.WriteLine(man1.age);
        Console.WriteLine(man2.age);
        Console.ReadLine();
     }
  }
```

structures cannot have default constructor and Destructors

```
class man
  {
      public int age;
      public man()
      {
         Console.WriteLine("the class created");
       }
      public man(int a)
      {
         age = a;
      }
     ~man()
     {
       Console.WriteLine("the class deleted");
     }
  }
```

```
struct man
      {
          public int age;
          public void man(int a)
          {
             age = a;
          }
      }
```

structures do not support inheritance

What is the **inheritance**?

One of the most important concepts in object-oriented programming is inheritance. **Inheritance allows us to define a class in terms of another class**, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and speeds up implementation time.

C# - Inheritance

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should **inherit** the members of an existing class.

This existing class is called the **base** class, and the new class is referred to as the **derived** class.

Base and Derived Classes

A class can be derived from more than one class , which means that it can inherit data and functions from multiple base classes.

The syntax used in C# for creating derived classes is as follows:

```
class <base_class>
{
   ...
}
class <derived_class> : <base_class>
{
   ...
}
```

```
class Shape
  {
    public int width;
    public  int height;
    public void setWidth(int w)
    {
      width = w;
    }
    public void setHeight(int h)
    {
      height = h;
    }
}

  // Derived class
  class Rectangle: Shape
  {
    public int getArea()
    {
      return (width * height);
    }
  }
```

```
static void Main(string[] args)
    {
      Rectangle Rect = new Rectangle();

      Rect.setWidth(5);
      Rect.setHeight(7);

      // Print the area of the object.
      Console.WriteLine("Total area: {0}",  Rect.getArea());
      Console.ReadKey();
    }
```

Calling Base Class Constructors

A derived class can call a constructor defined in its base class by using an expanded form of the derived class' constructor declaration and the **base** keyword. The general form of this expanded declaration is shown here:

derived-constructor(*parameter-list*) : base(*arg-list*) {

// body of constructor

}

```
class TwoDShape
 {
       double  width;
       double  height;
       public TwoDShape(double w, double h) {
           Width =w;
           Height= h;
       }
}
class Triangle : TwoDShape
{
  string Style;
  public Triangle(string s, double w, double h) : base(w, h) {
    Style =s;
  }
  public double Area() {
        return Width * Height / 2;
   }
}
```

```
static void Main()
 {
     Triangle t1 new Triangle("isosceles", 4.0, 4.0);
     Triangle t2 new Triangle("right", 8.0, 12.0);
      Console.WriteLine("Area is " + t1.Area());
}
```

Using Protected Access

Using Protected Access

a private member of a base class is not accessible to a derived class.

This would seem to imply that if you wanted a derived class to have access to some member in the base class, it would need to be public.

Of course, making the member public also makes it available to all other code, which may not be desirable.

Fortunately, this implication is untrue because C# allows you to create a *protected member.* A protected member is public within a class hierarchy, but private outside that hierarchy.

```
class B
{
     protected int i, j;
     public void Set(int a, int b)
     {
       i =a;
       j =b;
     }
     public void Show()
    {
        Console.WriteLine(i + " " + j);
    }
}
class D : B
{
    int k;
    public void Setk()
    {
      k= i * j;
    }
    public void Showk()
    {
        Console.WriteLine(k);
    }}
```

```
class ProtectedDemo
{
     static void Main()
      {
          D ob =new D();
          ob.Set(2, 3); // OK, known to D
          ob.Show(); // OK, known to D
          ob.Setk(); // OK, part of D
          ob.Showk(); // OK, part of D
      }
}
```

Static Members of a C# Class

Static Members of a C# Class

We can define class members as static using the **static** keyword. When we declare a member of a class as static, it means no matter how many objects of the class are created, there is only one copy of the static member.

The keyword **static** implies that only one instance of the member exists for a class. Static variables are used for defining constants because their values can be retrieved by invoking the class without creating an instance of it. Static variables can be initialized outside the member function or class definition. You can also initialize static variables inside the class definition.

- You can also declare a **member function** as **static**. Such functions can access only static variable . The static functions exist even before the object is created.

```
class car
  {
     public static int NumberOfWheels = 4;
     public string model;
     public double cost;
     public string color;
     public static int incNumberOfWheels()
     {
        NumberOfWheels = 2 + NumberOfWheels;
        return NumberOfWheels;
     }
     public void display()
     {
        Console.WriteLine(NumberOfWheels);
        Console.WriteLine(color);
        Console.WriteLine(cost);
        Console.WriteLine(model);
     }

  }
```

```
static void Main(string[] args)
      {

          car corola = new car();
          corola.color = "blue";
          corola.model = "2017";
          corola.cost = 2000.4;
          corola.display();
          car.incNumberOfWheels();
          car castar = new car();
          castar.cost = 10000;
          castar.color = "white";
          castar.model = "2008";
          castar.display();
          Console.ReadKey();


      }
```

Static and Instance Class Members

- As well as having members such as properties, methods, and fields that are specific to object instances, it is also possible to **have *static* members**, which can be methods, properties, or fields.
- **Static members are shared between instances of a class**, so they can be thought of as global for objects of a given class.
- Static properties and fields enable you to access data that is **independent of any object** instances, and static methods enable you to execute commands related to the class type but not specific to object instances. When using static members, in fact, you don’t even need to instantiate an object.

- For example, the Console.WriteLine() and Convert.ToString() methods you have been using are static.
- At no point do you need to instantiate the Console or Convert classes (indeed, if you try, you’ll find that you can’t, as the constructors of these classes aren’t publicly accessible, as discussed earlier). If you include the using static System.Console; declaration at the beginning of your program, Console. is not required and you can call WriteLine() directly.