المحاضرة الخامسة

Constructors in Derived & Base Classes

The constructors and destructors of the base class are not inherited. Rather than being inherited, constructors of the base class are called either implicitly or explicitly upon creation of an object from the derived class.

لا تُوْرَثَ دالة البناء ولكنها تُستدعى إمّا ضمنياً أَو بشكل صريح عند تكوين كائن مِنْ الصنفِ المُشتقّ

We know that constructors are used for initializing objects. One important thing to note is that, as long as no base class constructor takes any argument, the derived class need not have a constructor function. However, if any base class contains a constructor with one or more arguments, then it is mandatory (فرضي أو إجباري) for the derived class to have a constructor and pass the arguments to the base class constructors.

- Remember that when we use inheritance, we usually create objects using the derived class.
- When both the base and the derived classes contain constructors, the base constructor is executed first and then the derived class constructor.

Syntax

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

Derived constructor (parameter_list) : **base**(base parameter_list)

{

….. ..

}

Example:

```
class DerivedClass : BaseClass
{
        public DerivedClass (int x) : base(x)
        {  ... … }
 }
```

Example:

```
class B
   {
      protected int i, j;
      public B(int x, int y)    // base constructor
      {
         i = x;
         j = y;
      }
   }
class D : B
{
  int  k;
  public D (int x, int y, int z): base (y, z) // derived constructor calling base constructor
  {
        k = x;
  }
 static void Main(string[] args)
 {
   D d =new D (5, 7, 8);
 }
}
```

Example:

```
using System;
class E
  {
    public int x;
   public E()
    {
       Console.WriteLine("constructor  of E");
       x = 10;
    }
       public void E_Name()
         {
            Console.WriteLine("E,x={0}",x);
         }
  }
class S : E
{
    public int y;
    public S()
    {
```

```
        Console.WriteLine("constructor  of S");
         y = 9;
       }
        public void S_Name ()
          {
             Console.WriteLine("s, y= {0}",y);
             Console.WriteLine("E,x={0}", x);
           }

        new public void E_Name()
         {
            Console.WriteLine("sE");
         }
     }

class C : E
{
   public int r;
   public C( )
   { r = 5; }

   public void C_Name ()
        {
           Console.WriteLine("C, r= {0}",r);
        }
}
class B
{
          static void Main()
            {
                  S obj = new S( );
                  obj . S_Name ( ) ;
                  obj.x = 8;
                  obj.S_Name();
                  E ob2 = new E();
                  ob2.E_Name();
                  ob2 = obj;
                  ob2.E_Name();
                  ob2.x = 7;
                  ob2.E_Name();
              }
```

}

```
constructor  of E
constructor  of S
s, y= 9
E,x=10
s, y= 9
E,x=8
constructor  of E
E,x=10
E,x=8
E,x=7
Press any key to continue . .
```

Example:

```
class Shape
{
   public int x,y;
   public Shape()
   {
      Console.WriteLine("constructor  of  Shape");
      x = 10; y = 20;
   }
   public void display()
   {
      Console.WriteLine("shape, x = {0}, y = {1}", x, y);
   }
}

class Circle : Shape
{
   public double r;
   public Circle()
   {
      Console.WriteLine("constructor  of Circle");
      r = 9;
   }
    new public void display( )
   {
      Console.WriteLine("circle : shape");
      Console.WriteLine("circle, r = {0}", r);
      base.display();
   }
}
class Cube : Shape
```

```
{   public int h;
    public Cube()
    {   Console.WriteLine("constructor  of Cube");
        h = 5;
    }
    new public void display()
    {
        Console.WriteLine("cube : shape");
        Console.WriteLine("Cube, h = {0}", h);
        base.display();
    }
}
class B
{
 static void Main(string[] args)
 {
Circle obj = new Circle(); //call constructor Circle and Shape
        obj.display();
        obj.x = 8;
        obj.display();
Shape ob2 = new Shape(); //call constructor Shape only
        ob2.display();
        ob2 = obj; //ob2 refer to the same data members of obj
        ob2.display();
        ob2.x = 7; // x is changed for both objects
        ob2.display();
Cube ob3 = new Cube(); //call constructor Cube and Shape
        ob3.display();
 }
}
```

The program output:

```
constructor  of  Shape
constructor  of Circle
circle : shape
circle, r = 9
shape, x = 10, y = 20
circle : shape
circle, r = 9
shape, x = 8, y = 20
constructor  of  Shape
shape, x = 10, y = 20
shape, x = 8, y = 20
shape, x = 7, y = 20
constructor  of  Shape
constructor  of Cube
cube : shape
Cube, h = 5
shape, x = 10, y = 20
Press any key to continue . . .
```

Properties

Setting & getting the value of private attribute.

C# normally use methods known as set & get methods. These methods (knows as properties) are made public & private ways to access or modify private data.

Declaring Properties

Properties are declared using the property declaration statement, such as:

<modifier> <type> <property name>

{

- - - - - - - - - - - -

}

The property modifier includes **access modifiers**, such as public, private, protected, and internal. In addition, the property modifier includes the new, static, override, abstract, and sealed modifiers. You can use more than one modifier in a property declaration statement. Because a property is not a variable, you cannot pass a property as a ref or an out parameter.

The property declaration statement also includes the **type of the property**, followed by **its name of the property**. Inside the block of the property declaration statement, you include the **accessor** declaration statements. These statements are executable statements that define the actions to be performed while reading and writing values to an attribute.

Each property that you declare has accessors associated with it. These accessors define statements that enable you to read and write values to a property.

A **property** is an accessory that provides and encapsulates access to a data field

Accessors

The accessors used with properties are the get and set accessors. These accessors are followed by a block of statements to be executed when the accessor is invoked.

◆ get **accessor.** The get accessor is a method used to read values to a property. The get accessor does not take any parameter, but it returns a value of the data type specified in the property declaration statement. In the body of the get accessor, you need to include a return statement.

◆ set **accessor.** The set accessor is a method used to write values to a property. The set accessor takes a single implicit parameter named value but does not return any value..

To understand the get and set accessors, look at the following example:

```
public class Car
{
    string color;
        public string Color1
            {
               g e t
                  {
                     return color;
                  }
              s e t
                  {
                     color = value;
                  }
          }
}
```

This code declares a property Color1 for the class Car. The property declaration statement contains the get and set accessors. When you need to read data from the property, the get accessor is implicitly invoked. The get accessor returns a variable color of the same data type as that of the property. The variable color stores the value read by using the get accessor.

When you need to write a value to a property, the set accessor is invoked.T he set accessor takes a parameter named value that specifies the value to be written to the variable color. To write the value Green to the property, you simply need to write the following statement:

Car car1 = new Car();

car1.color = “Green”;

Based on the type of accessors defined in the property declaration statement, properties are classified as follows.

◆ read-only property. The property definition of a read-only property contains only the get accessor.

◆ write-only property. The property definition of a write-only property contains only the set accessor.

◆ read-write property. The property definition of a read-write property contains both the get and set accessors.

Ex:

```
class Point
 {
 int my_X; // my_X is private
int my_Y; // my_Y is private
 public int x
   {
          get
     {
        return my_X;
     }
       set
     {
       my_X = value;
     }
 }
 public int y
```

```
  {
      get
      {
          return my_Y;
      }
      set
      {
          my_Y = value;
      }
    }
}

class PropApp
    {
        public static void Main()
            {
                Point starting = new Point();
                Point ending = new Point();

             starting.x = 1;
             starting.y = 4;
            ending.x = 10;
             ending.y = 11;

 System.Console.WriteLine("Point 1: ({0},{1})", starting.x, starting.y);
 System.Console.WriteLine("Point 2: ({0},{1})", ending.x, ending.y);
    }
}
```

NOTE:

Unlike local variables, which are not automatically initialized, every field has a default initial values, a value provided by C# when you do not specify the initial values.