عنوان مضمون

Visual Programming-I

توسط : صفری

خزان1397

C# Object oriented features

C# - Polymorphism

C# - Polymorphism

The word **polymorphism** means having many forms. In object-oriented programming paradigm, polymorphism is often expressed as 'one interface, multiple functions'.

Polymorphism can be static or dynamic. In **static polymorphism**, the response to a function is determined at the compile time. In **dynamic polymorphism**, it is decided at run-time.

Static Polymorphism

- The mechanism of linking a function with an object during compile time is called early binding. It is also called static binding. C# provides two techniques to implement static polymorphism. They are:
- Function overloading
- Operator overloading

Static Polymorphism : Function Overloading

- You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments and/or order of arguments in the argument list.

- You **cannot** overload function declarations that differ only by **return type.**

```
class Printdata
 {
   void print(int i)
   {
     Console.WriteLine("Printing int: {0}", i );
   }

   void print(double f)
   {
     Console.WriteLine("Printing float: {0}" , f);
   }

   void print(string s)
   {
     Console.WriteLine("Printing string: {0}", s);
   }
```

```
static void Main(string[] args)
   {
     Printdata p = new Printdata();

     // Call print to print integer
     p.print(5);

     // Call print to print float
     p.print(500.263);

     // Call print to print string
     p.print("Hello C++");
     Console.ReadKey();
   }
```

Static Polymorphism : Operator overloading

You can redefine or overload most of the built-in operators available in C#. Thus a programmer can use operators with user-defined types as well. Overloaded operators are functions with special names the keyword **operator** followed by the symbol for the operator being defined. similar to any other function, an overloaded operator has a return type and a parameter list.

```
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()
   {
     return length * breadth * height;
   }

   // Overload + operator to add two Box objects.
   public static Box operator+ (Box b, Box c)
   {
     Box box = new Box();
     box.length = b.length + c.length;
     box.breadth = b.breadth + c.breadth;
     box.height = b.height + c.height;
     return box;
   }
 }
```

```
static void Main(string[] args)
    {
        Box Box1 = new Box();
        Box Box2 = new Box();
        Box Box3 = new Box();
        double volume = 0.0;
        // box 1 specification
        Box1.length = 6.0;
        Box1.breadth = 7.0;
        Box1.height = 5.0;
        // box 2 specification
        Box2.length = 12.0;
        Box2.breadth = 13.0;
        Box2.height = 10.0;
        // Add two object as follows:
        Box3 = Box1 + Box2;
        // volume of box 3
        volume = Box3.getVolume();
  Console.WriteLine("Volume of Box3 : {0}", volume);
        Console.ReadKey();
    }
  }
}
```

Dynamic Polymorphism

C# allows you to create abstract classes that are used to provide partial class implementation of an interface. Implementation is completed when a derived class inherits from it. **Abstract** classes contain abstract methods, which are implemented by the derived class. The derived classes have more specialized functionality.

Here are the rules about abstract classes:

- You cannot create an instance of an abstract class
- You cannot declare an abstract method outside an abstract class
- Abstract classes cannot be inherited.

```
abstract class Shape
 {
   public abstract int area();
 }
 class Rectangle:  Shape
 {
   private int length;
   private int width;
   public Rectangle( int a=0, int b=0)
   {
     length = a;
     width = b;
   }
   public override int area ()
   {
     Console.WriteLine("Rectangle class area :");
     return (width * length);
   }
 }
```

```
static void Main(string[] args)
    {
      Rectangle r = new Rectangle(10, 7);
      double a = r.area();
      Console.WriteLine("Area: {0}",a);
      Console.ReadKey();
    }
```

When you have a function defined in a class that you want to be implemented in an inherited class(es), you use **virtual** functions. The virtual functions could be implemented differently in different inherited class and the call to these functions will be decided at runtime.

Dynamic polymorphism is implemented by **abstract classes** and **virtual functions**.

```
class Shape
 { protected int width, height;
   public Shape( int a=0, int b=0)
   {  width = a;
      height = b;}
   public virtual int area()
   {  Console.WriteLine("Parent class area :");
      return 0;}
 }
 class Rectangle: Shape
 {  public Rectangle( int a=0, int b=0): base(a, b)
   {   }
   public override int area ()
   {  Console.WriteLine("Rectangle class area :");
      return (width * height);
   }
 }
 class Triangle: Shape
 { public Triangle(int a = 0, int b = 0): base(a, b)
   { }
   public override int area()
   {Console.WriteLine("Triangle class area :");
      return (width * height / 2); } }
```

```
static void Main(string[] args)
   {
      Rectangle r = new Rectangle(10, 7);
      Triangle t = new Triangle(10, 5);
       Console.WriteLine("Area: “+r.area() );
      Console.WriteLine("Area: “+t.area() );
      Console.ReadKey();
   }
```