Visual C# .Net using framework 4.5

Eng. Mahmoud Ouf

Lecture 05

Example

A Common Code developer wants to create a generic method for filtering an array of integers, but with the ability to specify the algorithm for filtration to the Application developer

Solution

The Common Code Developer:

```
public delegate bool IntFilter(int i);
public class Common
{
    public static int[] FilterArray(int[] ints, IntFilter filter)
    {
        ArrayList aList = new ArrayList();
        foreach(int i in ints)
        {
            if(filter(i))
            {aList.Add(i);}
        }
        return((int[])aList.ToArray(typeof(int)));
    }
}
```

Solution

The Application Code Developer (I)

```
Class MyApplication
{
        public static bool IsOdd(int i)
        {
                return ((i % 2)==1);
        }
        public static void Main()
        {
                int[] nums = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
                int[] fnum = Common.FilterArray(nums, MyApplication.IsOdd);
                foreach(int i in fnum)
                        Console.WriteLine(i);
        }
}
```

Solution

The Application Code Developer, Anonymous method “C#2.0” (II)

```
Class MyApplication
{
        public static void Main()
        {
                int[] nums = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
                int[] fnum = Common.FilterArray(nums, delegate(int i)
                                        {return((i%2)==1);});
                foreach(int i in fnum)
                        Console.WriteLine(i);
        }
}
```

Solution

The Application Code Developer, Lambda Expression “C#3.0” (III)

```
Class MyApplication
{
        public static void Main()
        {
                int[] nums = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
                int[] fnum = Common.FilterArray(nums, j=>((j % 2)==1));
                foreach(int i in fnum)
                        Console.WriteLine(i);
        }
}
```

Events

An event in C# is a way for a class (Sender) to provide notifications to clients (Listener) of that class when some interesting thing happens to an object. The most familiar use for events is in graphical user interfaces

The ideal case that we have 2 classes:

1)Sender, which must contains the events. The event itself didn't know what function will be executed, he knows what is the Signature of the function to be executed and its return type (delegate). In the event declaration, it must specify the delegate. Also, the Sender Class must contains the code that fire the event.

2)Listener, contains the relation of the events with the method that will be executed when the event fire. This is done through the delegate. It also contains the implementation of the methods to be executed when the event fire.

Example:

Write a program that read a number from the user, this number is between 1 and 100. If the number is Less than 1, it print a message on the screen "Less than 1" and if it is greater than 100, it prints "Out of Range"

Events

1) The Sender

```
public delegate void ValueHandler(string msg);
public class Sender
{
        public int I;
        //The Sender class can send these 2 events
        public event ValueHandler Low;
        public event ValueHandler High;
        public void ReadData()
        {
                I = int.Parse(Console.ReadLine());
                If(I < 1)
                {
                        if(Low != null)
                        {
                                Low("Out of Range, Number is too small");
                        }
                }
```

Events

1) The Sender

```
                else
                {
                        if(I > 100)
                        {
                                if(High !=null)
                                {
                                        High("Out of Range, Number is too large");
                                }
                        }
                }
        }//End Read Data
}//End Sender Class
```

Events

2) The Listening to the incoming Event

```
public class EventApp
{       public static int Main()
        {       Sender s = new Sender();
                //Hooks into events
                s.Low  += new ValueHandler(OnLow);
                s.High += new ValueHandler(OnHigh);
                //Call the Read Data
                s.ReadData();
        }
        public void OnLow(string str)
        {
                Console.WriteLine("{0}", str);
        }
        public void OnHigh(string str)
        {
                Console.WriteLine("{0}", str);
        }
}
```

Extension Method

Extension methods enable you to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type.

To create an extension method, declare it as a static method in a static class. The first parameter of an extension method must be the keyword this.

The following is an example of an extension method to convert the temperature from Fahrenheit to Celsius.

```
namespace MyNameSpace
{
    public static class MyClass
       {
         public static double ConvertToCelsius(this double fahrenheit)
          {
           return ((fahrenheit – 32) / 1.8); }
          }
       }
}
```

Extension Method

Now it is possible to invoke the extension method, ConvertToCelsius, as if it is an instance method:

```
double fahrenheit = 98.7;
double Celsius = fahrenheit.ConvertToCelsius();
```

So it adds a method called ConvertToCelisius to an existing type which is double here.

Anonymous Types

The C# compiler enables you to create a new type at runtime which is not available at the source code level.

It encapsulates a set of read-only properties into a single object without having to first explicitly define a type.

The type of the properties is inferred by the compiler which can create an anonymous type by using the properties in an object initializer.

Example:

```
var person = new { Name = "Mony Hamza", SSN = "12345678" };
```

Here, the compiler automatically creates an anonymous type and infers the types of the properties from the object initializer.

It also creates the private fields associated with these properties and the necessary set and get accessors.

When the object is instantiated, the properties are set to the values specified in the object initializer.

Anonymous Types

Here is an example for declaring an anonymous type and displaying its content:

```
class MyClass
{
   static void Main(string[] args)
   {
      // Declare an anonymous type:
      var obj1 = new { Name = "Ahmed", SSN ="12345678" };
      // Display the contents:
      Console.WriteLine("Name: {0}\nSSN: {1}", obj1.Name,obj1.SSN);
      Console.ReadLine();
   }
}
```

Output:

Name: “Ahmed”

SSN: 12345678

Anonymous Types

Note:

- Anonymous types are reference types that derive directly from object. From the perspective of the common language runtime, an anonymous type is no different from any other reference types.
- If two or more anonymous types have the same number and type of properties in the same order, the compiler treats them as the same type and they share the same compiler-generated type information.
- An anonymous type has method scope. To pass an anonymous type, or a collection that contains anonymous types, outside a method boundary, you must first cast the type to object. However, this defeats the strong typing of the anonymous type. If you must store your query results or pass them outside the method boundary, consider using an ordinary named struct or class instead of an anonymous type.

Generics

With the release of .NET 2.0, the C# programming language has been enhanced to support a new feature of the CTS termed generics. Simply, generics provide a way for programmers to define “placeholders” (formally termed type parameters) for method arguments and type definitions, which are specified at the time of invoking the generic method or creating the generic type

Generic methods: enable you to specify, with a single method declaration, a set of related methods.

Generic classes: enable you to specify, with a single class declaration, a set of related classes.

Generics

```
class Point<M>
{
        M x;
        M y;
        public Point(M a, M b)
        {
                x = a;
                y = b;
        }
        public override string ToString()
        {
                return ("X = " + x.ToString() + " , Y = " + y.ToString());
        }
}
```

Generics

Here, we have made a class Calculate contains a static method used to swap any 2 data from the same data type, it is also Generics

```
class Calculate<T>
{
        public static void Swap(ref T a, ref T b)
        {
                T temp;
                temp = a;
                a = b;
                b = temp;
        }
}
```

Generics

```
class Program
{
        static void Main(string[] args)
        {
                int x = 3;
                int y = 4;
                Calculate<int>.Swap(ref x, ref y);
                Console.WriteLine("X = {0}", x);
                Console.WriteLine("Y = {0}", y);
                Point<float> pt1 = new Point<float>(5.6f, 7.9f);
                Point<float> pt2 = new Point<float>(20.5f, 50.7f);
                Calculate<Point<float>>.Swap(ref pt1, ref pt2);
                Console.WriteLine("PT1 = {0}", pt1.ToString());
                Console.WriteLine("PT2 = {0}", pt2.ToString());
        }
}
```

Partial class

It is possible to split the definition of a class, struct or interface over two or more source files.

Each source file contains a section of the class definition, and all parts are combined when the application is compiled.

We use partial class for:

- When working on large projects, spreading a class over separate files allows multiple programmers to work on it simultaneously.
- When working automatic generated source, code can be added to the class without having to recreate the source file. Visual Studio uses this approach when creating Windows Forms

Use the **partial** keyword modifier to split class definition.

Partial class

```
public partial class Employee
{
        public void Test1()
        {
        }
}

public partial class Employee
{
        public void Test2()
        {
        }
}
```

Partial class

Notes:

- All class part must be defined within the same namesapace.
- All parts must use partial keyword.
- All parts must be available at compile time to form the final type.
- All parts must have the same accessibility (public, private, …).
- If any part is declared abstract, then the entire class is abstract.
- If any part is declared sealed, then the entire class is sealed.
- If any part declare a base type, then the entire type inherits that class.
- The partial keyword appears immediately before the keyword class, struct or interface.