Unit 6 Attributes, Delegates and Events

Structure:

6.1 Introduction

In the previous unit we discussed the object oriented programming concepts includes, objects and classes, constructors and destructors, method overloading and overriding inheritance, access modifiers, and polymorphism with the appropriate examples.

In this unit we are going to discuss the attributes in VB .NET including its role, to create the custom attributes. Also we are going to discuss the delegates and events. Delegate is nothing but the class has the signature to hold the reference of the methods. VB .Net events are handled by the delegates supports in call back events to process events.

Objectives:

After studying this unit, you will be able to:

- discuss about attributes and its creation
- explain the role of events in VB .NET
- write code to add events and even handlers
- brief techniques to add and remove handler
- explain delegates and how to do multicasting in delegates

6.2 Introduction to Attributes

Attributes provide a powerful method of associating metadata, or declarative information, with code (assemblies, types, methods, properties, and so forth). After an attribute is associated with a program entity, the attribute can be queried at run time by using a technique called reflection.

Attributes properties:

- Attributes add metadata to your program. Metadata is information about the types defined in a program. All .NET assemblies contain a specified set of metadata that describes the types and type members defined in the assembly. You can add custom attributes to specify any additional information that is required.
- You can apply one or more attributes to entire assemblies, modules, or smaller program elements such as classes and properties.
- Attributes can accept arguments in the same way as methods and properties.

- Your program can examine its own metadata or the metadata in other programs by using reflection.

Attributes can be placed on most of the declaration, though a specific attribute might restrict the types of declarations on which it is valid. In Visual Basic, an attribute is enclosed in angle brackets (< >). It must appear immediately before the element to which it is applied, on the same line. In this example, the Serializable Attribute attribute is used to apply a specific characteristic to a class:

```
<System.Serializable()> Public Class SampleClass Objects of this
   type can be serialized.
End Class
```

Method with the attribute DllImportAttribute is declared like

```
Imports System.Runtime.InteropServices
...
<System.Runtime.InteropServices.DllImport("user32.dll")> Sub
SampleMethod()
End Sub
```

More than one attribute can be declared like

```
Imports System.Runtime.InteropServices
Sub MethodA(<[In](), Out()> ByVal x As Double) End Sub
Sub MethodB(<Out(), [In]()> ByVal x As Double) End Sub
```

Attributes Parameter

Many attributes have parameters, which can be positional, unnamed, or named. Any positional parameters must be specified in a certain order and cannot be omitted; named parameters are optional and can be specified in any order. Positional parameters are specified first. For example, these three attributes are equivalent:

```
<DllImport("user32.dll")>
<DllImport("user32.dll", SetLastError:=False, ExactSpelling:=False)> <DllImport("user32.dll",
ExactSpelling:=False, SetLastError:=False)>
```

The first parameter, the DLL name, is positional and always comes first; the others are named. In this case, both named parameters default to false, so they can be omitted.

Attributes Targets

The target of an attribute is the entity to which the attribute applies. For example, an attribute may apply to a class, a particular method, or an entire assembly. By default, an attribute applies to the element that it precedes. But you can also explicitly identify, for example, whether an attribute is applied to a method, or to its parameter, or to its return value.

To explicitly identify an attribute target, use the following syntax:

```
<target : attribute-list>
```

Following figure 6.1 shows the possible attribute target

Attribute Target	Description
All	Attribute can be applied to any application element.
Assembly	Attribute can be applied to an assembly.
Class	Attribute can be applied to a class.
Constructor	Attribute can be applied to a constructor.
Delegate	Attribute can be applied to a delegate.
Enum	Attribute can be applied to an enumeration.
Event	Attribute can be applied to an event.
Field	Attribute can be applied to a field.
Interface	Attribute can be applied to an interface.
Method	Attribute can be applied to a method.
Module	Attribute can be applied to a module. **Note:** *Module* refers to a portable executable file (.dll or .exe) and not a Visual Basic standard module.
Parameter	Attribute can be applied to a parameter.
Property	Attribute can be applied to a property.
ReturnValue	Attribute can be applied to a return value.
Struct	Attribute can be applied to a structure; that is, a value type.

Fig. 6.1: Attribute Targets

Following are the common uses of attributes in code:

- Marking methods using the Web Method attribute in Web services to indicate that the method should be callable over the SOAP (Simple Object Access Protocol) protocol.
- Describing how to marshal method parameters when interoperating with native code. For more information, see Marshal As Attribute.
  - Describing the COM properties for classes, methods, and interfaces.
  - Calling unmanaged code using the Dll Import Attribute class.
  - Describing your assembly in terms of title, version, description, or trademark.
  - Describing which members of a class to serialize for persistence.
  - Describing how to map between class members and XML nodes for XML serialization.
  - Describing the security requirements for methods.
  - Specifying characteristics used to enforce security.
  - Controlling optimizations by the just-in-time (JIT) compiler so the code remains easy to debug.
  - Obtaining information about the caller to a method.

6.3 Creating Custom Attributes

Attributes are very convenient to the programmers because, with little effort, the runtime behavior of the program elements can be modified. For example, the attribute *Serializable* added to any class will indicate that the class can be serialized. As another example, consider the *Transaction* attribute of Serviced Component class. Just adding this attribute to the class determines its transactional behavior.

Custom attributes are nothing but normal classes that directly or indirectly derive from **System. Attribute**. Listed below are the steps required to create the custom attributes:

1. Applying the Attribute Usage attribute

The declaration of a custom attribute itself begins with the usage of an attribute, the **Attribute Usage** attribute. This has three members that qualify a custom attribute

Attribute Targets member

This member specifies to what program elements the custom attribute can be applied. For example, **Attribute Targets. Class** indicates that the attribute can be applied to a class, and **Attribute Targets. Method** indicates that the attribute can be applied to a method.

Inherited property

The Inherited property indicates whether the custom attribute can be inherited by classes that are derived from the classes to which your attribute is applied. This property takes either a true (the default) or false flag. For example, in the following code example, My Attribute has a default Inherited value of true.

<AttributeUsage(AttributeTargets.All, Inherited := True)> _Public Class MyAttribute Inherits Attribute .End Class

AllowMutliple property

Allow Mutliple property

The Allow Multiple property indicates whether multiple instances of the custom attribute can exist on an element. The value can be either true or false.

2. Initializing Attributes

We can use overloaded constructors to initialize attribute values. If properties are defined then, a combination of named and positional parameters can be used when initializing the attribute. Typically, all mandatory parameters are positional and all optional parameters are named. For example, consider an Attribute definition as shown below:

```
AttributeUsage(AttributeTargets.Class Or _AttributeTargets.Method)> Class DeveloperInfoAttribute Inherits Attribute

Private _Description As String Private _name As String Private lastChanged As String Public ReadOnly Property Notes() As String
Get Return _Description
End Get
 End Property
Public ReadOnly Property Author() As String
 Get Return _name
 End Get
 End Property
```

```
Public Property LastChanged() As String Get Return
_lastChanged
End Get
Set(ByVal Value As String)_lastChanged = Value End Set
End Property
Sub New(ByVal author As String, ByVal notes As String) _name = author _Description = notes
End Sub
End Class
```

Here, note that Author and Notes are required parameters for the attribute whereas Last Changed is an optional parameter.

3. Retrieving Attribute Information

As mentioned earlier, attributes are emitted into the assembly and hence we can use Reflection to obtain the attributes defined for a particular programming element. More specifically, we use the **Get Custom Attribute** shared method of the **Attribute** class to obtain an instance of the custom attribute. The code sample shown below obtains a reference to the custom attribute

```
DeveloperInfoAttribute defined for the class TestAttr Dim CustAttr As
DeveloperInfoAttribute
CustAttr = CType(Attribute.GetCustomAttribute(GetType(TestAttr), _
    GetType(DeveloperInfoAttribute)),DeveloperInfoAttribute)
```

Likewise, the code sample shown below gets an instance of the custom attribute defined for the method Dummy of the TestAttr class CustAttr = CType(Attribute.GetCustomAttribute(GetType(TestAttr). _

```
    GetMember("Dummy")(0), _
    GetType(DeveloperInfoAttribute)), _
    DeveloperInfoAttribute)
```

6.4 Events in VB .NET

In VB .NET the events and delegates are closely work together. An event is defined as a

message sent by an object announcing that something has happened. Events are implemented using delegates, a form of object-oriented function pointer that allows a function to be invoked indirectly by way of a reference to the function.

Delegate: A delegate can be defined as a type safe function pointer. It encapsulates the memory address of a function in your code. Whenever you create or use an event in code, you are using a delegate. When the event is thrown, the framework examines the delegate behind the event and then calls the function that the delegate points to.

6.4.1 Events and Event handler

As we discussed event informs the application that something important is happened. For example, when the user click on any of the control on the form, say Button. Button handler event procedure will handle the event. We can declare the event with the keyword Event with in classes. Modules or structures.

```
Event AnEvent (ByVal EventNumber As Integer)
```

When an object wants to communicate to an application about an event occurred will happen through message broadcasting called raising of an event. VB .NET supports this activity with the following syntax.

RaiseEvent AnEvent (EventNumer)

Here the event can be raised with in the block where it is declared.

Event Sender

The objects that are able to raise an event are called event sender or event source. Controls, Forms and user created objects are called event sender.

Event Handler

Event handlers are the pre-defined procedures that are called when the event occurs. All the valid procedures and subroutine are called event handler. VB.NET uses the standard structure to declare these handlers, event sender followed by underscore and the name of the event. Form1_Load

6.4.2 Adding Events to a class

Events can be added to the class by declaring them within the Event statement. Declaration of the statement includes the name of the event followed by the list arguments.

Public Event PercentDone(ByVal Percent As Single, - ByRef Cancel As
Boolean)

Events cannot have optional arguments, return values or param Array arguments. Adding an event to a class specifies that an object of this class can raise a specific event. To cause an event to actually happen, you must use the **Raise Event** statement. You can use **Handles** keyword or **Add Handler** statement to associate the event with an event handler procedure. Events must be raised within the scope where they are declared. For example, a derived class cannot raise events inherited from a base class.

6.4.3 Writing Event Handlers

The way you construct an event handler depends on how you want to associate it with events. The standard way to create an event handler is to use the Handles keyword with the WithEvents keyword. Visual Basic .NET provides a second way to handle events: the AddHandler statement. AddHandler and Remove Handler allow you to dynamically start and stop event handling for a specific event. You can use either approach, but you should not use both With Events and AddHandler with the same event.

Handling Events Using WithEvents

The **WithEvents** keyword allows you to create class or module-level object variables that can be used with the **Handles** clause in event handlers.

To handle events using With Events and Handles clause

1. In the Declarations section of the module that will handle the event, use the **WithEvents** keyword to declare an object variable for the source of your events, as in the following example:

   Public With Events Class Inst As Class1

2. In the Code editor, choose the **WithEvents** variable you just declared from the **Class Name** drop-down list on the left.

3. Choose the event you want to handle from the **Method Name** drop-down list on the right. The Code editor creates the empty event handler procedure with a **Handles** clause.

4. Add event-handling code to the event handler procedure using the supplied arguments. The following code provides an example:

   Public Sub ClassInst_AnEvent(ByVal EventNumber As System.Integer)

   _

   Handles ClassInst.AnEvent

   MessageBox.Show("Received event number: " & CStr(EventNumber)) End Sub

Event handling with AddHandler

AddHandler is statement available in VB .NET supports dynamic connection of events with event handler procedure.

Steps involved to handle event through AddHandler

Dim Cla1 As New Class1() // Declares an object variable of the class that is the source of the events you want to handle. Unlike a WithEvents variable, this can be a local variable in a procedure.

AddHnadler Cla1.AnEvent, AddressOf EHandler Use the AddHandler statement to specify the name of the event sender, and the AddressOf statement to provide the name of your event handler.

In event handling, any procedure can act as an event handler, provided it supports the exact parameter for the event. Below is the even handler sample code

Public Sub EHandler(ByVal EventNumber As Integer)
MessageBox.Show("Received event number " &
CStr(EventNumber))

End Sub

Stop Handling Events using RemoveHandler

The RemoteHandler can be used to disconnect the dynamically connected events.
RemoveHandler Cl.AnEvent, AddressOf EHandler Use the RemoveHandler statement to specify the name of the event sender, and the AddressOf statement to provide the name of your event handler. The syntax for RemoveHandler statements will always closely match the AddHandler statement used to start event handling.

WithEvents and the Handles clause

The WithEvents statement and the Handles clause provide a declarative way of specifying event handlers. Events raised by an object declared with the WithEvents keyword can be handled by any procedure with a Handles statement for that event, as shown in the following example:

Dim WithEvents EClass As New EventClass() ' Declare a WithEvents variable.

Sub TestEvents() EClass.RaiseEvents()
End Sub
' Declare an event handler that handles multiple events.
Sub EClass_EventHandler() Handles Eclass.XEvent, Eclass.YEvent MsgBox("Received Event.")

```
End Sub

Class EventClass
  Public Event XEvent()
  Public Event YEvent()
  Sub RaiseEvents() 'Raises two events handled by
EClass_EventHandler.
    RaiseEvent XEvent()
    RaiseEvent YEvent()
  End Sub
End Class
```

The **WithEvents** statement and the **Handles** clause are often the best choice for event handlers, because the declarative syntax they use makes event handling easier to code, read and debug. However, be aware of the following limitations on the use of **WithEvents** variables:

- You cannot use a **WithEvents** variable as a generic object variable. That is, you cannot declare it as **Object** – you must specify the class name when you declare the variable.
- You cannot use **WithEvents** to declaratively handle shared events, since they are not tied to an instance that can be assigned to a **WithEvents** variable.
- You cannot create arrays of **WithEvents** variables.
- **WithEvents** variables allow a single event handler to handle one or more kind of event, or one or more event handlers to handle the same kind of event.

6.4.4 Add Handler and Remove Handler

These handlers are like our Handles clause, supports handle the event effectively.

AddHandler: This statement is similar to the Handles clause in that both allow you to specify an event handler that will handle an event. However, AddHandler along with RemoveHandler provide greater flexibility than the Handles clause, allowing you to dynamically add, remove, and change the error handler associated with an event. And unlike Handles, AddHandler allows you to associate multiple event handlers with a single event.

AddHandler takes two arguments: the name of an event from an event sender such as a control, and an expression that evaluates to a delegate. You do not need to explicitly specify the delegate class when using AddHandler, since the AddressOf statement always returns a reference to the delegate. The following example associates an event handler with an event raised by an object:

```
AddHandler MyObject.Event1, AddressOf Me.MyEventHandler
```

RemoveHandler: As the name indicates this helps to deallocate an event from the event handler those are allotted or connected through Add Handler. The syntax also is similar to that of AddHandler except the syntax RemoveHandler.

```
RemoveHandler MyObject.Event1, AddressOf Me.MyEventHandler
```

Now we will see one example for event handler that defines the class that raises an event when you call the Cause Event method. This event is handled by an event handler procedure called Event Handler. To run this

example, add the following code to the form class of a Visual Basic .NET Windows Application project, and call the Test Events procedure with an integer argument.

```
Public Class Class1
  ' Declare an event for this class.
  Public Event Event1(ByVal EventNumber As Integer) ' Define a method
  that raises an event.
  Sub CauseEvent(ByVal EventNumber As Integer) RaiseEvent
    Event1(EventNumber)
  End Sub End Class

Protected Sub TestEvents(ByVal EventNumber As Integer) Dim MyObject As
  New Class1()
  AddHandler MyObject.Event1, AddressOf Me.EventHandler ' Cause the object
  to raise an event. MyObject.CauseEvent(EventNumber)
End Sub

Sub EventHandler(ByVal EventNumber As Integer)
  MessageBox.Show("Received event number " &
CStr(EventNumber))
End Sub
```

6.5 Delegates

A delegate can be defined as a **"**type safe function pointer**"**. It encapsulates the memory address of a function in your code. Whenever you create or use an event in code, you are using a delegate. When the event is thrown, the framework examines the delegate behind the event and then calls the
function that the delegate points to. Delegates can be combined to form groups of functions that can be called together.

Let us first take a quick look at how to define and invoke a delegate. First we declare our delegate in our form class:

Private Delegate Sub MyDelSub()

Then we use the delegate by simply declaring a variable of the delegate and assigning the sub or function to run when called. First the sub to be called:

Private Sub WriteToDebug() Debug.WriteLine("Delegate Wrote To Debug Window")End Sub

You will notice also that it matches our declaration of MyDelSub; it's a sub routine with no parameters. And then our test code:

Dim del As MyDelSubdel = New MyDelSub(AddressOf WriteToDebug) del.Invoke()

When we invoke the delegate, the WriteToDebug sub is run. Visual Basic hides most of the implementation of delegates when you use events, which are based off invoking a delegate. This is the equivalent of the above delegate invoke also.

Private Event My Event(), declare it in the class to use it, add a handler and raise the event. Add Handler My Event, Address Of Write To Debug Raise Event My Event()**.** If delegates stopped at this point, they would be useless since events are less work and do the same thing.

Delegates are also useful in situations where you need an intermediary between a calling procedure and the procedure being called. For example, you might want an object that raises events to be able to call different event handlers under different circumstances. Unfortunately,

the object raising events cannot know ahead of time which event handler is handling a specific event. Visual Basic .NET lets you dynamically associate event handlers with events by creating a delegate for you when you use the AddHandler statement. At run time, the delegate forwards calls to the appropriate event handler.

Although you can create your own delegates, in most cases Visual Basic .NET creates the delegate and takes care of the details for you. For example, an Event statement implicitly defines a delegate class named <EventName>EventHandler as a nested class of the class containing the Event statement, and with the same signature as the event. The AddressOf statement implicitly creates an instance of a delegate. For example, the following two lines of code are equivalent:

AddHandler Button1.Click, AddressOf Me.Button1_Click

AddHandler Button1.Click, New EventHandler(AddressOf Button1_Click)

Delegates that are created using shorthand can be identified by the compiler by its context.

Event declaration using existing delegate type

Following is the syntax used, If you want to declare an event with the existing delegate with its underlying delegate.

Event *AnEvent* As *DelegateType* also used to route multiple event to same handler.

6.5.1 Multicast delegates

Multicast delegates allow you to chain together several functions or subs that are all called together when the delegate is invoked. For the current iteration of the framework, you cannot designate the order that the functions are run, it runs one after another. Let us look at the code for the multicast delegate.

First we add a new sub for our second delegate.

```
Private Sub WriteToDebug2()
   Debug.WriteLine ("Delegate Wrote To Debug Window 2" ) End Sub
```

Our declaration of the MySubDelegate stays the same, and here is our new usage code.

```
Dim del As MyDelSub Dim del2 As
MyDelSub Dim delAll As [Delegate]
del = New MyDelSub(AddressOf WriteToDebug) del2 = New
MyDelSub(AddressOf WriteToDebug2) delAll =
MulticastDelegate.Combine(del, del2) delAll.DynamicInvoke( Nothing )
```

As we examine this code, we see three delegate variables; two for our normal delegates that call our subs and one form the combined of other two delegates. We set up our normal delegates as always, one points to WriteToDebug, the other to WriteToDebug2. When we combine the two delegates into our third, we utilize the static function Combine, of the Multicast Delegate class. It has two overloads, one that combines two delegates like we used, and one that takes an array of delegates. Next we invoke all the delegates with the combined delegates Dynamic Invoke property, passing in Nothing for its parameter. We could have also passed in an array of objects that would be used for parameters to the invoked subs.

If you check out the declaration of the last sample, you see another huge benefit of delegates. Notice that both del and del2 point to different functions, but are of the same type, MyDelSub. This opens up loads of programming potential. It allows you to point a MyDelSub variable to any sub that has the same signature as itself. In our case, it's a simple sub with no parameters. This behavior will let your program more generically. It works well on both sides of the equation,

either invoking a delegate from inside your class or receiving a delegate from outside your class to work upon. Just remember that when you register a function via handles or AddHandler, you are telling a delegate somewhere to make sure it calls your function when it is invoked. If you have multiple functions the have handles for the same event, you are just using a multicast delegate

6.6 Summary

- Attributes helps in adding metadata to your program. Metadata will have the collection of types that are defined in the program.
- Attributes parameter can be positional, named or unnamed.
- Generally the target of the attribute lies on the entity where the attribute belong to.
- Overloaded constructors are used to initialize values of the attributes.
- The role of event is to inform the application that something had happened.
- Event handlers are nothing but the predefined procedure will be called when the event occurs.
- Delegates are defined as “Type safe function pointer”. It comprise of memory addresses of all the functions defined in the program.
- Multicast delegates are useful where the delegate needs to invoked with the multiple delegates.
- Multicast delegates allow you to chain together several functions or subs that are all called together when the delegate is invoked

6.7 Questions and Exercises:

1. What is attributes? Explain its properties.
2. Discuss the targets of attributes.
3. How can we initialize the attributes explain with example.
4. Discuss on events and event handler.
5. Explain the role of delegates on VB .NET

6.8 Suggested Readings:

- http://msdn.microsoft.com/en-us/library/k2kt7a7y(v=vs.71).aspx
- http://www.developerfusion.com/article/5251/delegates-in-vbnet/2
- http://www.codeguru.com/vb/gen/vb_general/attributes/article.php/c6073 /Creating-and-Using-Custom-Attributes-with-VBNET.htm