Events

- You must follow naming conventions to avoid confusion
- You must document events and their actions thoroughly
- You must offer inherited classes the ability to raise your events
- You must not handle too many events within the same procedure

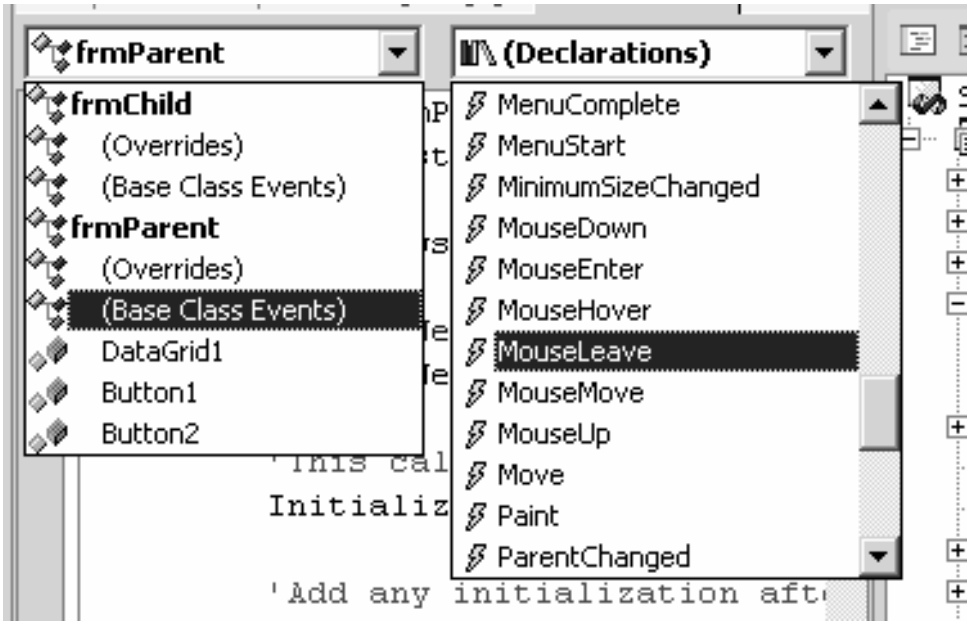
WithEvents and Events

WithEvents

The use of the `WithEvents` keyword allows us to handle any events raised by the object from within our code. You can of course only use the `WithEvents` keyword on objects that have events that you can read within your scope. This means that if the object you are attempting to handle events for has events, but they are all scoped **privately**, then you cannot declare the object `WithEvents`.

Unlike the use of Handles, discussed later, which links events to our methods, `WithEvents` makes all exposed events available to us, although without explicitly requesting the handling of the event we are not listening for it. This fact is true for all objects capable of creating events. Within VB.NET, the only way of using `WithEvents`, is with events that have been defined using the `Event` keyword and so preventing you from handling external events in this fasion.

An additional benefit gained within the Visual Studio IDE by the use of `WithEvents`, is the ability to use the combo boxes at the top of the IDE to select an event for a particular object from a list of events that the object exposes.




```
Private Sub Button1_Click( _
        ByVal sender As Object, _
        ByVal e As System.EventArgs) _
          Handles Button1.Click
  Debug.WriteLine("Mouse Click - Button1")
End Sub

Private Sub Button2_MouseHover( _
        ByVal sender As Object, _
        ByVal e As System.EventArgs) _
          Handles Button2.MouseHover
  Debug.WriteLine("Mouse Hover - Button2")
End Sub
```

Handling Events

Use of the `Handles` keyword allows us to force reception of particular events into our methods. We need to ensure that the event signature matches out method signature. You may have noticed this happening for you within the Visual Studio .NET IDE. If you drop a button onto a form and double-click it at design time, you will notice the IDE switching you to the code behind the form. The code section will be the event handler for the `Click` event of the button – such as:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
                          ByVal e As System.EventArgs) _
                          Handles Button1.Click
End Sub
```

The naming convention of the method follows the standard of `ObjectName_EventName`. The event can however be named anything you like, this is especially useful to note when you are using a single method to handle multiple events – this is covered later.

The use of the `Handles` keyword informs the compiler that this method will receive event notifications for the event associated with the described object; in this case, it is the `Click` event of the `Button1` object.

The `Handles` keyword can also be used to handle events in a sub-class from a base class..

Multiple Events

If we need one method that is required to handle multiple events, then the `Handles` keyword offers us this flexibility. Again, the only requirement is that the method signatures are the same. Adding an additional button to our form and allowing our event handler to receive both of the buttons click events demonstrates this:

```
Private Sub Button_Click(ByVal sender As System.Object, _
                         ByVal e As System.EventArgs) _
                         Handles Button1.Click, Button2.Click
End Sub
```

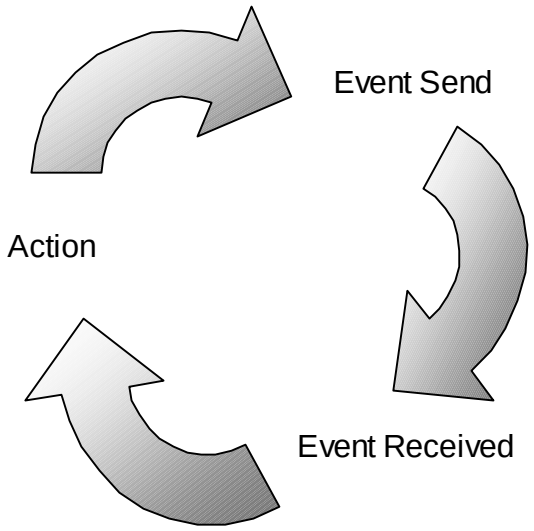
Now that this method is handling multiple events, then the standard naming convention causes the method to be a little confusing. As mentioned earlier, a method can be named anything that the developer requires as long as it conforms to the events signature. In this case dropping the button number from the name has served adequately.

You will have noticed that most events within .NET confirm to this standard signature. This is useful for a variety of reasons namely that you can handle different events within a single method, again as long as the event signatures are the same. A number of the standard .NET events not only conform to this standard but also have the basic signature of two parameters: an object and `System.EventsArgs`. If the second parameter is inherited from EventArgs then you will only see the arguments defined by `EventArgs`. If, however, you know that the object arriving is inherited from `EventArgs` you can cast the object in order to expose the additional arguments. If we felt that our `Button_Click` code needs to respond to the activation of the form, then by adding an additional argument to the end of our `Handles` section will accomplish this:

```
Private Sub Button_Click(ByVal sender As System.Object, _
                         ByVal e As System.EventArgs) _
                         Handles Button1.Click, Button2.Click, _
                         MyBase.Activated
End Sub
```

Events and Delegates

An event, using Windows terminology, is simply a message reported by an application or even your operating system to indicate the occurrence of something that may be of interest. . This action can be caused by an application, such as the indication that a process has completed. Events can also be instantiated by something that a user does, like clicking a button with their mouse. In order to simplify further context, you should be aware of some simple event terminology; the raiser of an event is referred to as the **sender**, while the application that responds to this event is referred to as the **receiver.** The use of events within a COM type application does however offer slightly different naming conventions of source as the raiser and sink as the receiver. An event driven application can be referred to as a closed-loop system; this means that the applications behavior is influenced by both past and current situations. This representation of the feedback into the application as a closed loop can be shown with the following schematic:.



A class that raises events does not always know if any other application is interested in the event, nor does it know what application will be making use of the message; there are of course exceptions to the rule, when you create an application where you or your organization design both the sender and receiver classes. A sender can send events using scope operatives to restrict the scoping of the event – the sender could then be entirely aware of whom the receiver would be.

In order to accomplish this, the .NET Framework offers us the use of the **delegate** type to provide us with the functionality of a pointer.

Delegates have a vast and important role within the .NET Framework; this section will concentrate solely on the use of delegates within event handling.

Naming Conventions

We have covered a couple of naming conventions so far, a couple descriptively and one or two within code. They have been summarized here to assist the reading of the rest of this chapter:

- Delegate Arguments – `ArgumentName`*Args*
- Delegate Naming – `DelegateName`*EventHandler*

Delegates

A delegate is a reference type that has a signature and holds references for only those methods that match its signature. **A delegate is equivalent to a type-safe function pointer or callback**. The following example shows the declaration of an event delegate:

```
Public Structure InvoiceItemArgs
  Public InvoiceNumber As String
  Public InvoiceDate As Date
  Public CreatedByUserId As Integer
End Structure

Public Delegate Sub InvoiceCreatedEventHandler ( _
                                          ByVal sender As Object, _
                                          ByVal e As InvoiceItemArgs)
```

As you can see, the syntax is similar to that of a method declaration. The `Delegate` keyword informs the compiler that this method is in fact a delegate type allowing the CLR to provide an implementation for this object..

Those of you familiar with using .NET may have noticed some similarities with the example above and the base events raised within .NET objects.

```
'Example of VB.NET internal Paint event declaration
Private Sub frmExample_Paint( _
    ByVal sender As Object, _
    ByVal e As System.Windows.Forms.PaintEventArgs) Handles MyBase.Paint
End Sub
```

Event delegates should follow declaration conventions in order to supply uniformity within your application. You should only supply two parameters for an event delegate; the first being the object that raises the event, the second being the data that the receiver requires.

An instance of the `CreateInvoiceEventHandler` event delegate can be bound to any method that has the same signature. In order to perform event delegation, we must first create a method with the same signature.

```
Public Class SalesInvoice
  Public Sub SalesInvoiceCreated(ByVal sender As Object, _
                                 ByVal e As InvoiceArgs)
```

```
  End Sub
End Class
```

In order to connect the `SalesInvoiceCreated()` method to an event we need to create an instance of the `InvoiceCreatedEventHandler` that takes a reference to the `InvoiceCreated()` method from the `Invoice` class.

```
Public Class Invoice
  Private clsSalesInvoice As New SalesInvoice()

  Private hanInvoiceCreated As InvoiceEventHandler = _
            AddressOf clsSalesInvoiceInvoice.InvoiceCreated
End Class
```

What we have achieved here is that when we call `Invoke()` on the `hanInvoiceCreated` object, we are in fact routing this call through to the `InvoiceCreated()` method of the instance of `clsSalesInvoice`. As this chapter is about the use of events, we can turn this example into an event delegation demonstration.

```
Public Structure InvoiceItemArgs
  Public InvoiceNumber As String
  Public InvoiceDate As Date
  Public CreatedByUserId As Integer
End Structure

Public Delegate Sub InvoiceCreatedEventHandler ( _
                                          ByVal sender As Object, _
                                          ByVal e As InvoiceItemArgs)

Public Class SalesInvoice
  Public Sub SalesInvoiceCreated(ByVal sender As Object, _
                                 ByVal e As InvoiceItemArgs)
    Console.WriteLine("Invoice Created.")
  End Sub
End Class

Public Class Invoice
  Public Event InvoiceCreated As InvoiceCreatedEventHandler

  Public Sub CreateNewInvoice()
    Console.WriteLine("Invoice Process Instantiated.")
    RaiseEvent InvoiceCreated(Me, New InvoiceItemArgs())
  End Sub
End Class

Public Class InvoiceProcess
  Private clsInvoice As New Invoice()
  Private clsSalesInvoice As New SalesInvoice()

  '1) The use of a delegate to raise the event
  Public Sub Main()
    AddHandler clsInvoice.InvoiceCreated, _
      AddressOf clsSalesInvoice.SalesInvoiceCreated
```

```
    Console.WriteLine("Delegate Created.")
    'start invoice process
    clsInvoice.CreateNewInvoice()
  End Sub

  '2) The use of WithEvents to perform the same operation
  '   using the familiar handles keyword
  Public Sub clsInvoice_InvoiceCreated( _
                           ByVal sender As Object, _
                           ByVal e As InvoiceItemArgs) Handles _
                           clsInvoice.InvoiceCreated
    'Call the SalesInvoiceCreated method of the clsSaleInvoice
    'class, passing in new parameters
    Console.WriteLine("WithEvents Created.")
    clsSalesInvoice.SalesInvoiceCreated(sender, e)
  End Sub
End Class
```

As is now shown, we have created an instance of both the `Invoice` and `SalesInvoice` classes. When the `InvoiceCreated` event is raised within the `Invoice` class, we are in fact routing this to the `SalesInvoiceCreated` method within our instance of the `SalesInvoice` class. The output from the above example is shown below. It should be noted that as both the `WithEvents` and `AddHandler` operations are in essence performing the same function, the priority of either function cannot be guaranteed.

```
Delegate Created.
Invoice Process Instantiated.
WithEvents Created.
Invoice Created.
Invoice Created.
```

If you find that your event delegate does not require any event data to be forwarded to the receiver then you can make use of the `System.EventHandler` class. This class gives us the most basic signature for event delegation. The signature for this class is two parameters, one being an object the second being a `System.EventArgs` argument. The standard System.EventHandler will be familiar to those of you that have had some exposure to VB.NET as most of the standard UI events have this signature.

Event Arguments

You may have noticed at the beginning of this section the inclusion of a structure named `InvoicePostArgs`. Again, typical naming conventions are applied here with the format of `ArgumentName`*`Args`*, this also allows for a common interface and follows those conventions applied within the .NET Framework itself. While the previous examples used a structure as the parameter for the arguments to our event delegate, the usual approach would be to provide a class that handles the arguments; the .NET framework takes this approach. A class allows us greater flexibility over the object being passed into the event delegate, and if we take this approach then we can prevent future upgrade pitfalls. A class is simply a special kind of structure that contains additional code. This means that by making use of a class instead of

structure is not only the approach that should be taken, but is in fact simply and extension of the previous approach. An example of the `InvoiceItemArgs` structure as a class is as follows.

```
Public Class InvoiceItemArgs
  Inherits EventArgs

  Private mInvoiceNumber As String
  Private mInvoiceDate As Date
  Private mCreatedByUserId As Integer

  Public Sub New()
    mInvoiceDate = Date.Now
  End Sub

  Public ReadOnly Property InvoiceNumber() As String
    Get
      Return mInvoiceNumber
    End Get
  End Property

  Public ReadOnly Property InvoiceDate() As Date
    Get
      Return mInvoiceDate
    End Get
  End Property

  Public ReadOnly Property CreatedByUserId() As Integer
    Get
      Return mCreatedByUserId
    End Get
  End Property
End Class
```

As is apparent, we need to generate more code to take this approach, but what is also apparent is the extra flexibility we have over the information passed into the argument. The example also demonstrates the inheritance from the `EventArgs` base class. This provides any additional methods within the `EventArgs` class, and so it should be encouraged as good practice. Currently, only the Empty field is offered above that which is included in standard classes. The Empty field is a read only instance of an empty `EventArgs` class; equivalent to creating a new `EventArgs` object.

```
Dim oEventArgs As New EventArgs()
```

One further advantage of the inheritance from the EventArgs base class is that we can pass our object into a method that expects a standard EventArgs parameter; like the `Paint()` event.

Events

An event is actually a kind of delegate implementation. Behind the scenes, the .NET environment writes delegates in order to handle events on your behalf. This concept is easier to understand if we consider the creation of "event type" code using delegates for a familiar UI event – `DoubleClick`. This event, implemented in the .NET Framework is fired each time the use clicks the mouse twice. While in reality Windows raises a `WM_LBUTTONDBLCLK` message, which is in turn handled by the .NET Framework, for the purposes of the demonstration we will have two classes: the sender (Windows), and the receiver (your application).

Firstly, we should create a delegate from which our sender will invoke the function:

```
Public Delegate Sub DoubleClickEventHandler( _
                      ByVal sender As Object, _
                      ByVal e As System.EventArgs)
```

As you can see, the delegate that we have created mimics the signature of the CLR's `DoubleClick` event. The delegate has been suffixed with `EventHandler`; this helps indicate what this declaration actually is.

Secondly, we will now create a class that performs the sending of the event, this would usually be Windows for an event such as the `DoubleClick`. The code will simply outline what may be happening within the Windows internal code.

```
Public Class Sender
  Public Event DoubleClick As DoubleClickEventHandler
    Protected Sub OnDoubleClick()
      RaiseEvent DoubleClick(Me, New System.EventArgs())
    End Sub
End Class
```

Using this declaration of an event allows us to indicate to the compiler that the event `DoubleClick` has the signature of `DoubleClickEventHandler`. Although events do not require parameters, this is considered usual practice within the .NET framework and as a result, this rule should generally be followed. The inclusion of the `OnDoubleClick()` method allows inherited classes to invoke our event, without which they would be unable to do so..

Finally, we need to create a class to receive the event. This class would commonly be a Windows Form in the case of a `DoubleClick` event:

```
Public Class Receiver
  Public Sub New()
    Dim SenderInstance As New Sender()
        AddHandler SenderInstance.DoubleClick, _
                 AddressOf Receiver_DoubleClick
    End Sub

    Private Sub Receiver_DoubleClick (ByVal sender As Object, _
```

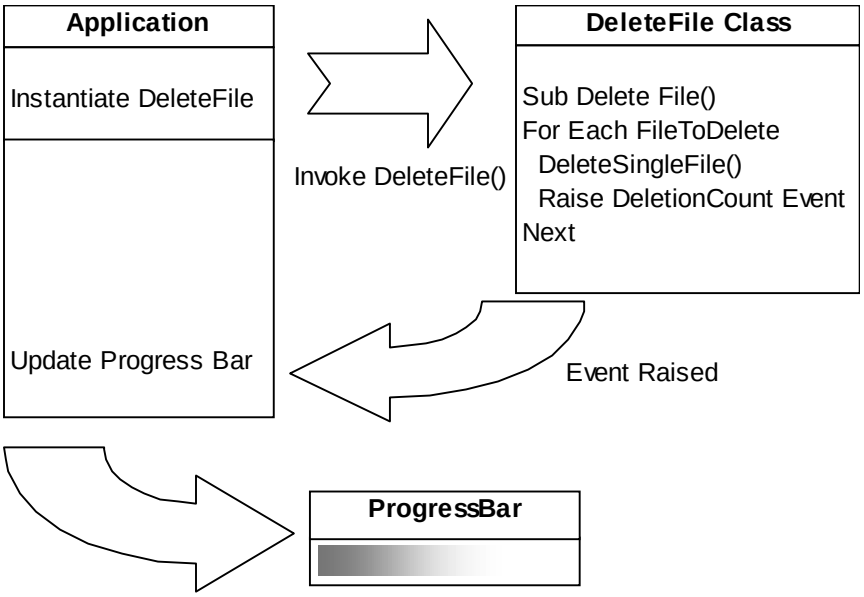
```
                                        ByVal e As System.EventArgs)
      Console.WriteLine("Event - DoubleClick")
    End Sub
End Class
```

This class demonstrates what actually happens when an object is declared using `WithEvents`. The method `Receiver_DoubleClick()` handles all the registered events raised from the `Sender` class. This rudimentary example demonstrates the code that VB.NET generates behind the scenes when we use some of the helper functions. Perhaps, as Microsoft include the form drawing code within the code module it would have been nice to include the event declaration code as well. It certainly would have clarified the code for developers that are not so familiar with the VB.NET environment.

The use of delegates within events allows us to encapsulate both an object instance and a method in order to allow the calling of the event to be anonymous. An event is a way in which a class handles the redirection of method calls when something useful has, is or will be useful occurs.

This leads nicely into what an event actually is. An event is an action that can be recognized in code of an important occurrence, such as the double-click of the mouse pointer or the opening of a form. Events allow different tasks to communicate; a piece of code that deletes files may raise an event indicating its progress in order to update a status indicator of some sort..

Figure 1



7086_06_01.fig

Figure 1 illustrates how a delete file class may inform the calling class of an indication of its progress. The application would firstly instantiate the `DeleteFile` class and then call the `DeleteFile()` method. This method would raise an event for each file deletion, and as our application has acknowledged intent to participate in that event, our calling application is informed of the event; and updates the progress indicator accordingly. This can also be demonstrated simply in code using a console output:

```
Module Application
  Private WithEvents oDeleteFile As DeleteFile
  Sub Main()
    oDeleteFile = New DeleteFile()
    oDeleteFile.DeleteFile(New String() _
                     {"c:\temp\file1.txt", _
                      "c:\temp\file2.txt"})
    Console.WriteLine("Process Completed")
  End Sub

  Public Sub oDeleteFile_FileDeleted( _
                     ByVal sender As Object, _
                     ByVal e As ConsoleApplication1.DeleteFileArgs) _
                     Handles oDeleteFile.FileDeleted
    Console.WriteLine("File: " & e.FileDeletionName & _
                     " - Count: " & e.FileDeletionCount.ToString & _
                     " – " & IIf(e.DeletionSuccess, "Deleted", "Failed"))
  End Sub
End Module

Public Class DeleteFile
  Event FileDeleted(ByVal sender As Object, ByVal e As DeleteFileArgs)

  Public Sub DeleteFile(ByVal strFileList() As String)
    If Not strFileList Is Nothing Then
      Dim strFile As String
      Dim intCount As Integer = 0
      Dim blnSuccess As Boolean
      'loop through, deleting files
      For Each strFile In strFileList
        Try
          System.IO.File.Delete(strFile)
          blnSuccess = True
        Catch DeletionException As Exception
          blnSuccess = False
        Finally
          'increment the counter
          intCount += 1
          'raise the event
          RaiseEvent FileDeleted(Me, New DeleteFileArgs(intCount, _
                                          strFile, blnSuccess))
        End Try
      Next strFile
    End If
```

```
  End Sub
End Class

Public Class DeleteFileArgs
  Inherits EventArgs

  Private mFileDeletionCount As Integer
  Private mFileDeletionName As String
  Private mDeletionSuccess As Boolean

  Public Sub New(ByVal FileDeletionCount As Integer, _
                 ByVal FileDeletionName As String, _
                 ByVal DeletionSuccess As Boolean)
    mFileDeletionCount = FileDeletionCount
    mFileDeletionName = FileDeletionName
    mDeletionSuccess = DeletionSuccess
  End Sub

  Public ReadOnly Property FileDeletionCount() As Integer
    Get
      Return mFileDeletionCount
    End Get
  End Property

  Public ReadOnly Property FileDeletionName() As String
    Get
      Return mFileDeletionName
    End Get
  End Property

  Public ReadOnly Property DeletionSuccess() As Boolean
    Get
      Return mDeletionSuccess
    End Get
  End Property
End Class
```

This example covers a number of the areas discussed in this chapter so far. These include:

- Inheriting from `EventArgs` in order to extend the standard parameter output.
- The declaration of the Deletion class using the keyword, `WithEvents` to allow use of the `Handles` keyword.
- The use of `Handles` to signify our intend to receive the `FileDeleted` event.

If the files exist on the machine then the output would look something like this:

```
File: c:\temp\file1.txt - Count: 1 - Deleted
File: c:\temp\file2.txt - Count: 2 - Deleted
Process Completed
```

Event Calling Restrictions

As we have already explored, an event is a mechanism for one action informing objects that are listening of the process that has occurred. This could be summarized by describing an event as a collection of addresses to inform.

Using this definition, it should be apparent that we are in control of a vast number of event handling declarations; in fact, we are restricted only by memory. This should be more than sufficient for most developers, although if you are required to extend the boundaries beyond that which is considered usual, you must understand the consequences of such a load bearing.

Each event handled will consume some of your resources, the more handled, the greater the resource requirement and the slower the machine handling them becomes. Even the creation of event handlers themselves will drastically slow down during extensive event handling code.

This should not cause the majority of you any discomfort, as I have already stated the number of events that can sensibly be handled even on a standard build PC without drastic depreciation of processor is certainly in its millions.

One caveat to this is that although we can create as many event receivers as we would like, the object from which we may be listening for events could determine a maximum number of listeners. This is trappable within a `try-catch` block and can be prevented by closing listeners down once you have finished with them. Another alternative is to create an Event Factory Handler; discussed later in this chapter.

Consuming Events

While the handling of events is commonly referred to as event consumption, this term can be a little confusing. The event is not consumed but rather borrowed when the parent class has finished with it. This is demonstrated in the following code example:

```
Public Class ParentForm
  Inherits System.Windows.Forms.Form

  Public Sub New()
    AddHandler MyBase.Load, AddressOf ParentForm_Load
  End Sub

  Private Sub ParentForm_Load(ByVal sender As Object, _
                              ByVal e As System.EventArgs)
    Debug.WriteLine("ParentForm Loaded")
  End Sub
End Class

Public Class ChildForm
  Inherits ParentForm
```

```
  Public Sub New()
    MyBase.New()

    AddHandler MyBase.Load, AddressOf ChildForm_Load
  End Sub

  Private Sub ChildForm_Load(ByVal sender As Object, _
                             ByVal e As System.EventArgs)
    Debug.WriteLine("ChildForm Loaded")
  End Sub
End Class
```

If `ChildForm` is instantiated then the debug window will display the following:

```
ParentForm Loaded
ChildForm Loaded
```

This demonstrates the point that the event is not consumed but rather passed on once the parent has finished with it. Although it is apparent from this example that the firing order of the events is filtered through from parent to child to grandchild and so on, this is not how the sequence of event firing works; events are not fired in any particular order. The only thing that we can be sure of is that all receivers of our event will receive the event.

The process of event delivery is a synchronous one; this means that each handler of the event will receive notification one at a time and normal processing in the event raising object will not continue until the last handler has finished. It should be also noted that we cannot stop or interrupt this process once started, and the event raised processing will continue until the last event handler has completed.

To summarize, the process of consuming events is simply the handling or the receiving of events.

Raising Events

In order to raise an event you must write code that achieves all of the following:

- Creates an object to hold event data: This object should preferably be a class that inherits from `System.EventArgs`. The MSDN documentation claims that you *must* use a class to hold your event data and that this class *must* be inherited from `System.EventArgs`; neither of these statements are true.. It is however good practice to follow this rule.
- An event delegate
- A class to raise the event from that facilitates the following inclusions:
  - A declaration for the event
  - A method to expose the event using the convention of `On`*EventName*

These three elements are the backbone for the raising of events. If you are using a the Visual Studio IDE then much of this code is written for you behind the scenes. As with the events raised by the .NET Framework, if you are raising events provided by a third party, then you would usually only need to define the class to raise the events.

Expanding on our previous examples, consider that we need to create a new invoice. This invoice will, as we have already acknowledged, pass the event through to our `SaleInvoiceCreated()` method. We need to create a method to allow the raising of the event; this will of course be named `OnInvoiceCreated()` to follow the correct naming conventions. This method has been declared with the Protected scope operator to ensure that only classes that inherit from the Invoice class can override this implementation. In addition to this, we need to create a method that allows us to perform a create invoice operation. This is demonstrated as follows:

```
Public Class Invoice
  Public Event InvoiceCreated As InvoiceCreatedEventHandler

  Protected Overridable Sub OnInvoiceCreated(ByVal e As InvoiceItemArgs)
    RaiseEvent InvoiceCreated(Me, e)
  End Sub

  Public Sub Create()
    Dim args As InvoiceItemArgs
    args.InvoiceNumber = "1234A"
    OnInvoiceCreated(args)
    args = Nothing
  End Sub
End Class

Public Class InvoiceProcess
  Private oInvoice As New Invoice()
  Private oSalesInvoice As New SalesInvoice()

  Public Sub New()
    AddHandler oInvoice.InvoiceCreated, _
      AddressOf oSalesInvoice.SalesInvoiceCreated

    InvoiceInstance.Create()
  End Sub
End Class
```

This demonstrates the use of the `RaiseEvent` keyword to raise an event. You must always supply the correct number of parameters and parameter types when raising events; this is the same as the way you would call methods.

The most important benefit from using `AddHandler` to redirect events to another method is that you have the ability to remove the handling of the events from your code.

Consider an application that responds to events from a class that is performing a hard drive operation of some form, perhaps defragmenting. In order to perform this process the drive must have no activity for a period of time, this is our event, and when fired we respond by starting the hard drive operation. Whilst this demonstrates basic event handling, suppose we only expect this application to operate outside of working hours. We could of course put a check in our code that verifies the time prior to performing the operation, but this would mean that we are responding to events that we don't need to. The preferred approach would be to turn off or on the event handling as is required. This would be accomplished with the use of the `RemoveHandler` keyword. The syntax for `RemoveHandler` is the same as that of `AddHandler`, so you remove the event handle the same way you add it.. This allows us to start and stop execution of the event programmatically within our source code at any time. The following example illustratesthe use of RemoveHandler:

```
AddHandler oInvoice.InvoiceCreated, _
    AddressOf oSalesInvoice.SalesInvoiceCreated
```

```
RemoveHandler oInvoice.InvoiceCreated, _
    AddressOf oSalesInvoice.SalesInvoiceCreated
```

The parameters required to remove an event handler must exactly match those used to create the event handler. The advantages of this kind of functionality are boundless, with careful consideration you now have the ability to chose when, where and what events you want to receive.

Errors during Event Firing

If an error is raised during the firing of an event and the method handling the event does not handle the error, control is returned to the code that raised the event and if that code doesn't handle the error, it is passed up the calling chain. This is the same manner applied to error handling within standard procedures.

This process is easy to demonstrate with the following code:

```
Public Class EventErrorTest
  'Events are declared parameter-less to ease code-read
  Event EventA()
  Event EventB()
  Event EventC()

  Public Sub New()
    AddHandler EventA, AddressOf SubA
    AddHandler EventB, AddressOf SubB
    AddHandler EventC, AddressOf SubC

    RaiseEvent EventA()

    RemoveHandler EventA, AddressOf SubA
    RemoveHandler EventB, AddressOf SubB
```

```
    RemoveHandler EventC, AddressOf SubC
  End Sub

  Private Sub SubA()
    Try
      Debug.WriteLine("SubA-Fired")
      RaiseEvent EventB()
      Debug.WriteLine("SubA-OK")
    Catch
      Debug.WriteLine("SubA-Error")
    End Try
  End Sub

  Private Sub SubB()
    Try
      Debug.WriteLine("SubB-Fired")
      RaiseEvent EventC()
      Debug.WriteLine("SubB-OK")
    Catch
      Debug.WriteLine("SubB-Error")
    End Try
  End Sub

  Private Sub SubC()
    Debug.WriteLine("SubC-Fired")
    Err.Raise(0)
  End Sub
End Class
```

This code will produce the following results when `New()` is invoked:

```
SubA-Fired
SubB-Fired
SubC-Fired
SubB-Error
SubA-OK
```

As is demonstrated, an error has occurred within `SubC()`. This error is handled within `SubB()` and results in `SubA()` continuing without any issues. If however the error wasn't handled in `SubB()` then `SubA()` would receive the error.

OnEventName

Most events in the .NET Framework and one of the events we created earlier in the Raising Events section, `OnInvoiceCreated()` within the `Invoice` class contain an `OnEventName()` method. `EventName` is the name of the event to which the method refers.

This method would usually raises the event for the given `EventName` and it can be replaced by overriding the method. This does mean however that only inherited classes will have access to

the `OnEventName` methods. If we inherited from the Form class we could for example raise the `DoubleClick` of the form from within a buttons `DoubleClick` event:

```
Private Sub Button1_Click( _
               ByVal sender As Object, _
               ByVal e As System.EventArgs) Handles Button1.DoubleClick
  Me.OnDoubleClick(e)
End Sub
```

Events and Inheritance

Within the Invoice class discussed earlier, we have the ability to raise events that were defined using the `Event` keyword within the base class:

```
Protected Overridable Sub OnInvoiceCreate(ByVal e As InvoiceItemArgs)
  RaiseEvent InvoiceCreated(Me, e)
End Sub
```

Fortunately, if we subclass from the Invoice class, the subclass inherits all of our events and benefits from them as if the new subclass was in fact the base class. Although there is one caveat, we cannot overload events like we can with other methods.

```
Public Class SupplierInvoice
  Inherits Invoice

  Private Sub SupplierInvoice_InvoiceCreated( _
                   ByVal sender As Object, _
                   ByVal e As InvoicePostArgs) _
                       Handles MyBase.InvoiceCreated
  End Sub
End Class
```

Without using `WithEvents`, we have the ability to handle all the events that were declared within the base class.

The only disadvantage is that whilst we can make use of the events created in the parent class, we cannot raise any of the events directly. This means that if we try to use the `RaiseEvent` keyword then the compiler would prevent compilation of the application. In fact, if you are developing within Visual Studio .NET, then the IntelliSense will not offer the events declared from the base class when using `RaiseEvent`.

Usefully, we can accomplish this by providing the developer with a suitably scoped method that raises the event on our behalf. The most appropriate scope would be protected as this ensures that only inherited classes can call the method (which subsequently raises the event). These methods are referred to as `OnEventName` methods and were discussed previously.

Event Factory Handler

If the code you are creating involves a large amount of event handling, hanlding different types of events, a better approach may be to create an object that handles the throughput of these events. The issue arises because each event registered with the compiler creates one field against each delegate instance. The storage cost of each of these fields is unacceptable on applications with a large number of events, and the solution is to make use of the .NET **event properties** construct. Event properties can be used with another data structure to store our event delegates.

An example event factory handler would provide the developer with methods for both the assignment and removal of event handlers. This technique is relatively easy if we make use of the `Component` class. The Component class enables object sharing between applications and so suits our ideally.

Unfortunately VB.NET doesn't currently support **accessor-declarations**, this makes the presentation of a VB.NET sample impossible. Whilst VB.NET does not currently support, this may become available in future releases of the language, after all the language should simply be semantics. For completeness, below is a simple example of a C# implementation; after all, one of the advantages of .NET is the ability to use the most appropriate language for a specific task. This example should be simple enough to understand regardless of your previous C# experience:

```
public event EnabledChangedEventHandler EnabledChanged
{
  add
  {
    Events.AddHandler (EventEnabledChanged, value);
  }
  remove
  {
    Events.RemoveHandler(EventEnabledChanged, value);
  }
}
```

This method only proves to be useful if you have a large number of events where most of time, most of them are not implemented.

Using Delegations within the Windows API

Some of the API calls within Windows require the use of callbacks. These are in fact the same as the delegations discussed at the beginning of this chapter. The requirement of the API is to understand the method that it needs to call when it is raising an event. This method must obey all the rules previously discussed, primarily that the signature of the callback method is the same as the event that it is raising.

A simple API that requires this functionality is the `EnumChildWindows` API. While it was possible to achieve this in earlier versions of VB, we didn't have the type safety provided by delegates. An example of both the delegate and the function call are:

```
Public Delegate Function EnumChildWindowsCallBack(
                  ByVal hWndParent As Int32, _
                  ByVal lParam As Int32) As Int32
  Public Declare Function EnumChildWindows Lib "user32" ( _
                   ByVal hWndParent As Int32, _
                   ByVal lpEnumFunc As EnumChildWindowsCallBack, _
                   ByVal lParam As Int32) As Int32
```

In order to make use of this function we must first create a method that can accept event calls from the API; this function must have the same signature as that which the API expects.

```
Private Function ChildWindowHook(ByVal hWnd As Int32, _
                                 ByVal lParam As Int32) As Int32
  Debug.WriteLine(hWnd.ToString)
End Function
```

Now that we have a function that acts as our event receiver, we can simply pass the address of this function to the API when we call it:

```
Private Sub LoopThoughChildren(ByVal hWndParent As Int32)
  EnumChildWindows(hWndParent, AddressOf ChildWindowHook, 0)
End Sub
```

As we can see, event delegation is very useful in a variety of circumstances, including within the API itself.

Summary

Event handling has moved on considerably since previous version of VB. The capabilities of event handling are boundless. We can assign and remove event handling within code at will. In addition to this, we can also handle more than one event within the same routine, even if the events are different events.

We now have a number of ways of receiving events from the use of WithEvents to the use of Handles.