Events (Visual Basic)

Visual Studio 2015

While you might visualize a Visual Studio project as a series of procedures that execute in a sequence, in reality, most programs are event driven—meaning the flow of execution is determined by external occurrences called *events*.

An event is a signal that informs an application that something important has occurred. For example, when a user clicks a control on a form, the form can raise a **Click** event and call a procedure that handles the event. Events also allow separate tasks to communicate. Say, for example, that your application performs a sort task separately from the main application. If a user cancels the sort, your application can send a cancel event instructing the sort process to stop.

Event Terms and Concepts

This section describes the terms and concepts used with events in Visual Basic.

Declaring Events

You declare events within classes, structures, modules, and interfaces using the **Event** keyword, as in the following example:

VB

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

Raising Events

An event is like a message announcing that something important has occurred. The act of broadcasting the message is called *raising* the event. In Visual Basic, you raise events with the **RaiseEvent** statement, as in the following example:

VB

```
RaiseEvent AnEvent(EventNumber)
```

Events must be raised within the scope of the class, module, or structure where they are declared. For example, a derived class cannot raise events inherited from a base class.

Event Senders

Any object capable of raising an event is an *event sender*, also known as an *event source*. Forms, controls, and user-defined objects are examples of event senders.

Event Handlers

Event handlers are procedures that are called when a corresponding event occurs. You can use any valid subroutine with a matching signature as an event handler. You cannot use a function as an event handler, however, because it cannot return a value to the event source.

Visual Basic uses a standard naming convention for event handlers that combines the name of the event sender, an underscore, and the name of the event. For example, the **Click** event of a button named `button1` would be named `Sub button1_Click`.

> **Note**
>
> We recommend that you use this naming convention when defining event handlers for your own events, but it is not required; you can use any valid subroutine name.

Associating Events with Event Handlers

Before an event handler becomes usable, you must first associate it with an event by using either the **Handles** or **AddHandler** statement.

WithEvents and the Handles Clause

The **WithEvents** statement and **Handles** clause provide a declarative way of specifying event handlers. An event 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:

VB

```
' Declare a WithEvents variable.
Dim WithEvents EClass As New EventClass

' Call the method that raises the object's events.
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()
    ' RaiseEvents raises both events.
    Sub RaiseEvents()
```

```
            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 an object variable. That is, you cannot declare it as **Object**—you must specify the class name when you declare the variable.
- Because shared events are not tied to class instances, you cannot use **WithEvents** to declaratively handle shared events. Similarly, you cannot use **WithEvents** or **Handles** to handle events from a **Structure**. In both cases, you can use the **AddHandler** statement to handle those events.
- 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.

Although the **Handles** clause is the standard way of associating an event with an event handler, it is limited to associating events with event handlers at compile time.

In some cases, such as with events associated with forms or controls, Visual Basic automatically stubs out an empty event handler and associates it with an event. For example, when you double-click a command button on a form in design mode, Visual Basic creates an empty event handler and a **WithEvents** variable for the command button, as in the following code:

VB

```
Friend WithEvents Button1 As System.Windows.Forms.Button
Protected Sub Button1_Click() Handles Button1.Click
End Sub
```

AddHandler and RemoveHandler

The **AddHandler** statement is similar to the **Handles** clause in that both allow you to specify an event handler. However, **AddHandler**, used with **RemoveHandler**, provides greater flexibility than the **Handles** clause, allowing you to dynamically add, remove, and change the event handler associated with an event. If you want to handle shared events or events from a structure, you must use **AddHandler**.

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:

VB

```
AddHandler Obj.XEvent, AddressOf Me.XEventHandler
```

RemoveHandler, which disconnects an event from an event handler, uses the same syntax as **AddHandler**. For example:

VB

```
RemoveHandler Obj.XEvent, AddressOf Me.XEventHandler
```

In the following example, an event handler is associated with an event, and the event is raised. The event handler catches the event and displays a message.

Then the first event handler is removed and a different event handler is associated with the event. When the event is raised again, a different message is displayed.

Finally, the second event handler is removed and the event is raised for a third time. Because there is no longer an event handler associated with the event, no action is taken.

VB

```
Module Module1

    Sub Main()
        Dim c1 As New Class1
        ' Associate an event handler with an event.
        AddHandler c1.AnEvent, AddressOf EventHandler1
        ' Call a method to raise the event.
        c1.CauseTheEvent()
        ' Stop handling the event.
        RemoveHandler c1.AnEvent, AddressOf EventHandler1
        ' Now associate a different event handler with the event.
        AddHandler c1.AnEvent, AddressOf EventHandler2
        ' Call a method to raise the event.
        c1.CauseTheEvent()
        ' Stop handling the event.
        RemoveHandler c1.AnEvent, AddressOf EventHandler2
        ' This event will not be handled.
        c1.CauseTheEvent()
    End Sub

    Sub EventHandler1()
        ' Handle the event.
        MsgBox("EventHandler1 caught event.")
    End Sub

    Sub EventHandler2()
        ' Handle the event.
        MsgBox("EventHandler2 caught event.")
    End Sub

    Public Class Class1
        ' Declare an event.
```

```
        Public Event AnEvent()
        Sub CauseTheEvent()
            ' Raise an event.
            RaiseEvent AnEvent()
        End Sub
    End Class

End Module
```

Handling Events Inherited from a Base Class

Derived classes—classes that inherit characteristics from a base class—can handle events raised by their base class using the **Handles MyBase** statement.

To handle events from a base class

- Declare an event handler in the derived class by adding a **Handles MyBase.***eventname* statement to the declaration line of your event-handler procedure, where *eventname* is the name of the event in the base class you are handling. For example:

VB

```
Public Class BaseClass
    Public Event BaseEvent(ByVal i As Integer)
    ' Place methods and properties here.
End Class

Public Class DerivedClass
    Inherits BaseClass
    Sub EventHandler(ByVal x As Integer) Handles MyBase.BaseEvent
        ' Place code to handle events from BaseClass here.
    End Sub
End Class
```

Related Sections

Title	Description
Walkthrough: Declaring and Raising Events (Visual Basic)	Provides a step-by-step description of how to declare and raise events for a class.
Walkthrough: Handling Events (Visual Basic)	Demonstrates how to write an event-handler procedure.

How to: Declare Custom Events To Avoid Blocking (Visual Basic)	Demonstrates how to define a custom event that allows its event handlers to be called asynchronously.
How to: Declare Custom Events To Conserve Memory (Visual Basic)	Demonstrates how to define a custom event that uses memory only when the event is handled.
Troubleshooting Inherited Event Handlers in Visual Basic	Lists common issues that arise with event handlers in inherited components.
Handling and Raising Events	Provides an overview of the event model in the .NET Framework.
Creating Event Handlers in Windows Forms	Describes how to work with events associated with Windows Forms objects.
Delegates (Visual Basic)	Provides an overview of delegates in Visual Basic.

Walkthrough: Declaring and Raising Events (Visual Basic)

Visual Studio 2015

This walkthrough demonstrates how to declare and raise events for a class named `Widget`. After you complete the steps, you might want to read the companion topic, Walkthrough: Handling Events (Visual Basic), which shows how to use events from `Widget` objects to provide status information in an application.

The Widget Class

Assume for the moment that you have a `Widget` class. Your `Widget` class has a method that can take a long time to execute, and you want your application to be able to put up some kind of completion indicator.

Of course, you could make the `Widget` object show a percent-complete dialog box, but then you would be stuck with that dialog box in every project in which you used the `Widget` class. A good principle of object design is to let the application that uses an object handle the user interface—unless the whole purpose of the object is to manage a form or dialog box.

The purpose of `Widget` is to perform other tasks, so it is better to add a `PercentDone` event and let the procedure that calls `Widget`'s methods handle that event and display status updates. The `PercentDone` event can also provide a mechanism for canceling the task.

To build the code example for this topic

1. Open a new Visual Basic Windows Application project and create a form named `Form1`.

2. Add two buttons and a label to `Form1`.

3. Name the objects as shown in the following table.

Object	Property	Setting
`Button1`	`Text`	Start Task
`Button2`	`Text`	Cancel
`Label`	`(Name)`, `Text`	lblPercentDone, 0

4. On the **Project** menu, choose **Add Class** to add a class named `Widget.vb` to the project.

To declare an event for the Widget class

- Use the **Event** keyword to declare an event in the `Widget` class. Note that an event can have **ByVal** and **ByRef** arguments, as `Widget`'s `PercentDone` event demonstrates:

VB

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

When the calling object receives a `PercentDone` event, the `Percent` argument contains the percentage of the task that is complete. The `Cancel` argument can be set to **True** to cancel the method that raised the event.

Note

You can declare event arguments just as you do arguments of procedures, with the following exceptions: Events cannot have **Optional** or **ParamArray** arguments, and events do not have return values.

The `PercentDone` event is raised by the `LongTask` method of the `Widget` class. `LongTask` takes two arguments: the length of time the method pretends to be doing work, and the minimum time interval before `LongTask` pauses to raise the `PercentDone` event.

To raise the PercentDone event

1. To simplify access to the **Timer** property used by this class, add an **Imports** statement to the top of the declarations section of your class module, above the `Class Widget` statement.

VB

```
Imports Microsoft.VisualBasic.DateAndTime
```

2. Add the following code to the `Widget` class:

VB

```
Public Sub LongTask(ByVal Duration As Single,
                    ByVal MinimumInterval As Single)
    Dim Threshold As Single
    Dim Start As Single
    Dim blnCancel As Boolean

    ' The Timer property of the DateAndTime object returns the seconds
    ' and milliseconds that have passed since midnight.
    Start = CSng(Timer)
    Threshold = MinimumInterval

    Do While CSng(Timer) < (Start + Duration)
        ' In a real application, some unit of work would
        ' be done here each time through the loop.
        If CSng(Timer) > (Start + Threshold) Then
```

```
                RaiseEvent PercentDone(
                Threshold / Duration, blnCancel)
                ' Check to see if the operation was canceled.
                If blnCancel Then Exit Sub
                Threshold = Threshold + MinimumInterval
            End If
        Loop
    End Sub
```

When your application calls the `LongTask` method, the `Widget` class raises the `PercentDone` event every `MinimumInterval` seconds. When the event returns, `LongTask` checks to see if the `Cancel` argument was set to **True**.

A few disclaimers are necessary here. For simplicity, the `LongTask` procedure assumes you know in advance how long the task will take. This is almost never the case. Dividing tasks into chunks of even size can be difficult, and often what matters most to users is simply the amount of time that passes before they get an indication that something is happening.

You may have spotted another flaw in this sample. The **Timer** property returns the number of seconds that have passed since midnight; therefore, the application gets stuck if it is started just before midnight. A more careful approach to measuring time would take boundary conditions such as this into consideration, or avoid them altogether, using properties such as **Now**.

Now that the `Widget` class can raise events, you can move to the next walkthrough. Walkthrough: Handling Events (Visual Basic) demonstrates how to use **WithEvents** to associate an event handler with the `PercentDone` event.

See Also

Timer
Now
Walkthrough: Handling Events (Visual Basic)
Events (Visual Basic)

Walkthrough: Handling Events (Visual Basic)

Visual Studio 2015

This is the second of two topics that demonstrate how to work with events. The first topic, Walkthrough: Declaring and Raising Events, shows how to declare and raise events. This section uses the form and class from that walkthrough to show how to handle events when they take place.

The `Widget` class example uses traditional event-handling statements. Visual Basic provides other techniques for working with events. As an exercise, you can modify this example to use the **AddHandler** and **Handles** statements.

To handle the PercentDone event of the Widget class

1. Place the following code in `Form1`:

VB

```
Private WithEvents mWidget As Widget
Private mblnCancel As Boolean
```

The **WithEvents** keyword specifies that the variable `mWidget` is used to handle an object's events. You specify the kind of object by supplying the name of the class from which the object will be created.

The variable `mWidget` is declared in `Form1` because **WithEvents** variables must be class-level. This is true regardless of the type of class you place them in.

The variable `mblnCancel` is used to cancel the `LongTask` method.

Writing Code to Handle an Event

As soon as you declare a variable using **WithEvents**, the variable name appears in the left drop-down list of the class's **Code Editor**. When you select `mWidget`, the `Widget` class's events appear in the right drop-down list. Selecting an event displays the corresponding event procedure, with the prefix `mWidget` and an underscore. All the event procedures associated with a **WithEvents** variable are given the variable name as a prefix.

To handle an event

1. Select `mWidget` from the left drop-down list in the **Code Editor**.

2. Select the `PercentDone` event from the right drop-down list. The **Code Editor** opens the `mWidget_PercentDone` event procedure.

Note

The **Code Editor** is useful, but not required, for inserting new event handlers. In this walkthrough, it is more direct to just copy the event handlers directly into your code.

3. Add the following code to the `mWidget_PercentDone` event handler:

VB

```
Private Sub mWidget_PercentDone(
    ByVal Percent As Single,
    ByRef Cancel As Boolean
) Handles mWidget.PercentDone
    lblPercentDone.Text = CInt(100 * Percent) & "%"
    My.Application.DoEvents()
    If mblnCancel Then Cancel = True
End Sub
```

Whenever the `PercentDone` event is raised, the event procedure displays the percent complete in a **Label** control. The **DoEvents** method allows the label to repaint, and also gives the user the opportunity to click the **Cancel** button.

4. Add the following code for the `Button2_Click` event handler:

VB

```
Private Sub Button2_Click(
    ByVal sender As Object,
    ByVal e As System.EventArgs
) Handles Button2.Click
    mblnCancel = True
End Sub
```

If the user clicks the **Cancel** button while `LongTask` is running, the `Button2_Click` event is executed as soon as the **DoEvents** statement allows event processing to occur. The class-level variable `mblnCancel` is set to **True**, and the `mWidget_PercentDone` event then tests it and sets the `ByRef Cancel` argument to **True**.

Connecting a WithEvents Variable to an Object

`Form1` is now set up to handle a `Widget` object's events. All that remains is to find a `Widget` somewhere.

When you declare a variable **WithEvents** at design time, no object is associated with it. A **WithEvents** variable is just like any other object variable. You have to create an object and assign a reference to it with the **WithEvents** variable.

To create an object and assign a reference to it

1. Select **(Form1 Events)** from the left drop-down list in the **Code Editor**.

2. Select the `Load` event from the right drop-down list. The **Code Editor** opens the `Form1_Load` event procedure.

3. Add the following code for the `Form1_Load` event procedure to create the `Widget`:

VB

```
Private Sub Form1_Load(
    ByVal sender As System.Object,
    ByVal e As System.EventArgs
) Handles MyBase.Load
    mWidget = New Widget
End Sub
```

When this code executes, Visual Basic creates a `Widget` object and connects its events to the event procedures associated with `mWidget`. From that point on, whenever the `Widget` raises its `PercentDone` event, the `mWidget_PercentDone` event procedure is executed.

To call the LongTask method

- Add the following code to the `Button1_Click` event handler:

VB

```
Private Sub Button1_Click(
    ByVal sender As Object,
    ByVal e As System.EventArgs
) Handles Button1.Click
    mblnCancel = False
    lblPercentDone.Text = "0%"
    lblPercentDone.Refresh()
    mWidget.LongTask(12.2, 0.33)
    If Not mblnCancel Then lblPercentDone.Text = CStr(100) & "%"
End Sub
```

Before the `LongTask` method is called, the label that displays the percent complete must be initialized, and the class-level **Boolean** flag for canceling the method must be set to **False**.

`LongTask` is called with a task duration of 12.2 seconds. The `PercentDone` event is raised once every one-third of a second. Each time the event is raised, the `mWidget_PercentDone` event procedure is executed.

When `LongTask` is done, `mblnCancel` is tested to see if `LongTask` ended normally, or if it stopped because `mblnCancel` was set to **True**. The percent complete is updated only in the former case.

To run the program

1. Press F5 to put the project in run mode.
2. Click the **Start Task** button. Each time the `PercentDone` event is raised, the label is updated with the percentage of the task that is complete.
3. Click the **Cancel** button to stop the task. Notice that the appearance of the **Cancel** button does not change immediately when you click it. The **Click** event cannot happen until the **My.Application.DoEvents** statement

allows event processing.

> **Note**
>
> The **My.Application.DoEvents** method does not process events in exactly the same way as the form does. For example, in this walkthrough, you must click the **Cancel** button twice. To allow the form to handle the events directly, you can use multithreading. For more information, see Threading (C# and Visual Basic).

You may find it instructive to run the program with F11 and step through the code a line at a time. You can clearly see how execution enters `LongTask`, and then briefly re-enters `Form1` each time the `PercentDone` event is raised.

What would happen if, while execution was back in the code of `Form1`, the `LongTask` method were called again? At worst, a stack overflow might occur if `LongTask` were called every time the event was raised.

You can cause the variable `mWidget` to handle events for a different `Widget` object by assigning a reference to the new `Widget` to `mWidget`. In fact, you can make the code in `Button1_Click` do this every time you click the button.

To handle events for a different widget

- Add the following line of code to the `Button1_Click` procedure, immediately preceding the line that reads `mWidget.LongTask(12.2, 0.33)`:

VB

```
mWidget = New Widget
' Create a new Widget object.
```

The code above creates a new `Widget` each time the button is clicked. As soon as the `LongTask` method completes, the reference to the `Widget` is released, and the `Widget` is destroyed.

A **WithEvents** variable can contain only one object reference at a time, so if you assign a different `Widget` object to `mWidget`, the previous `Widget` object's events will no longer be handled. If `mWidget` is the only object variable containing a reference to the old `Widget`, the object is destroyed. If you want to handle events from several `Widget` objects, use the **AddHandler** statement to process events from each object separately.

> **Note**
>
> You can declare as many **WithEvents** variables as you need, but arrays of **WithEvents** variables are not supported.

See Also

Walkthrough: Declaring and Raising Events (Visual Basic)
Events (Visual Basic)

How to: Declare Custom Events To Avoid Blocking (Visual Basic)

Visual Studio 2015

There are several circumstances when it is important that one event handler not block subsequent event handlers. Custom events allow the event to call its event handlers asynchronously.

By default, the backing-store field for an event declaration is a multicast delegate that serially combines all the event handlers. This means that if one handler takes a long time to complete, it blocks the other handlers until it completes. (Well-behaved event handlers should never perform lengthy or potentially blocking operations.)

Instead of using the default implementation of events that Visual Basic provides, you can use a custom event to execute the event handlers asynchronously.

Example

In this example, the **AddHandler** accessor adds the delegate for each handler of the `Click` event to an ArrayList stored in the `EventHandlerList` field.

When code raises the `Click` event, the **RaiseEvent** accessor invokes all the event handler delegates asynchronously using the BeginInvoke method. That method invokes each handler on a worker thread and returns immediately, so handlers cannot block one another.

VB

```
Public NotInheritable Class ReliabilityOptimizedControl
    'Defines a list for storing the delegates
    Private EventHandlerList As New ArrayList

    'Defines the Click event using the custom event syntax.
    'The RaiseEvent always invokes the delegates asynchronously
    Public Custom Event Click As EventHandler
        AddHandler(ByVal value As EventHandler)
            EventHandlerList.Add(value)
        End AddHandler
        RemoveHandler(ByVal value As EventHandler)
            EventHandlerList.Remove(value)
        End RemoveHandler
        RaiseEvent(ByVal sender As Object, ByVal e As EventArgs)
            For Each handler As EventHandler In EventHandlerList
                If handler IsNot Nothing Then
                    handler.BeginInvoke(sender, e, Nothing, Nothing)
                End If
            Next
        End RaiseEvent
    End Event
End Class
```

See Also

ArrayList
BeginInvoke
Events (Visual Basic)
How to: Declare Custom Events To Conserve Memory (Visual Basic)

How to: Declare Custom Events To Conserve Memory (Visual Basic)

Visual Studio 2015

There are several circumstances when it is important that an application keep its memory usage low. Custom events allow the application to use memory only for the events that it handles.

By default, when a class declares an event, the compiler allocates memory for a field to store the event information. If a class has many unused events, they needlessly take up memory.

Instead of using the default implementation of events that Visual Basic provides, you can use custom events to manage the memory usage more carefully.

Example

In this example, the class uses one instance of the EventHandlerList class, stored in the `Events` field, to store information about the events in use. The EventHandlerList class is an optimized list class designed to hold delegates.

All events in the class use the `Events` field to keep track of what methods are handling each event.

VB

```
Public Class MemoryOptimizedBaseControl
    ' Define a delegate store for all event handlers.
    Private Events As New System.ComponentModel.EventHandlerList

    ' Define the Click event to use the delegate store.
    Public Custom Event Click As EventHandler
        AddHandler(ByVal value As EventHandler)
            Events.AddHandler("ClickEvent", value)
        End AddHandler
        RemoveHandler(ByVal value As EventHandler)
            Events.RemoveHandler("ClickEvent", value)
        End RemoveHandler
        RaiseEvent(ByVal sender As Object, ByVal e As EventArgs)
            CType(Events("ClickEvent"), EventHandler).Invoke(sender, e)
        End RaiseEvent
    End Event

    ' Define the DoubleClick event to use the same delegate store.
    Public Custom Event DoubleClick As EventHandler
        AddHandler(ByVal value As EventHandler)
            Events.AddHandler("DoubleClickEvent", value)
        End AddHandler
        RemoveHandler(ByVal value As EventHandler)
            Events.RemoveHandler("DoubleClickEvent", value)
        End RemoveHandler
```

```
            RaiseEvent(ByVal sender As Object, ByVal e As EventArgs)
                CType(Events("DoubleClickEvent"), EventHandler).Invoke(sender, e)
            End RaiseEvent
        End Event

        ' Define additional events to use the same delegate store.
        ' ...
    End Class
```

See Also

EventHandlerList
Events (Visual Basic)
How to: Declare Custom Events To Avoid Blocking (Visual Basic)

Troubleshooting Inherited Event Handlers in Visual Basic

Visual Studio 2015

This topic lists common issues that arise with event handlers in inherited components.

Procedures

Code in Event Handler Executes Twice for Every Call

- An inherited event handler must not include a Handles Clause (Visual Basic) clause. The method in the base class is already associated with the event and will fire accordingly. Remove the **Handles** clause from the inherited method.

VB

```
' INCORRECT
Protected Overrides Sub Button1_Click(
    ByVal sender As System.Object,
    ByVal e As System.EventArgs) Handles Button1.Click

    ' The Handles clause will cause all code
    ' in this block to be executed twice.
End Sub
```

- If the inherited method does not have a **Handles** keyword, verify that your code does not contain an extra AddHandler Statement or any additional methods that handle the same event.

See Also

Events (Visual Basic)

Handling and Raising Events

.NET Framework (current version)

Events in the .NET Framework are based on the delegate model. The delegate model follows the observer design pattern, which enables a subscriber to register with, and receive notifications from, a provider. An event sender pushes a notification that an event has happened, and an event receiver receives that notification and defines a response to it. This article describes the major components of the delegate model, how to consume events in applications, and how to implement events in your code.

For information about handling events in Windows 8.x Store apps, see Events and routed events overview (Windows store apps).

Events

An event is a message sent by an object to signal the occurrence of an action. The action could be caused by user interaction, such as a button click, or it could be raised by some other program logic, such as changing a property's value. The object that raises the event is called the *event sender*. The event sender doesn't know which object or method will receive (handle) the events it raises. The event is typically a member of the event sender; for example, the Click event is a member of the Button class, and the PropertyChanged event is a member of the class that implements the INotifyPropertyChanged interface.

To define an event, you use the **event** (in C#) or **Event** (in Visual Basic) keyword in the signature of your event class, and specify the type of delegate for the event. Delegates are described in the next section.

Typically, to raise an event, you add a method that is marked as **protected** and **virtual** (in C#) or **Protected** and **Overridable** (in Visual Basic). Name this method **On***EventName*; for example, `OnDataReceived`. The method should take one parameter that specifies an event data object. You provide this method to enable derived classes to override the logic for raising the event. A derived class should always call the **On***EventName* method of the base class to ensure that registered delegates receive the event.

The following example shows how to declare an event named `ThresholdReached`. The event is associated with the EventHandler delegate and raised in a method named `OnThresholdReached`.

VB

```
Public Class Counter
    Public Event ThresholdReached As EventHandler

    Protected Overridable Sub OnThresholdReached(e As EventArgs)
        RaiseEvent ThresholdReached(Me, e)
    End Sub

    ' provide remaining implementation for the class
End Class
```

Delegates

A delegate is a type that holds a reference to a method. A delegate is declared with a signature that shows the return type and parameters for the methods it references, and can hold references only to methods that match its signature. A delegate is thus equivalent to a type-safe function pointer or a callback. A delegate declaration is sufficient to define a delegate class.

Delegates have many uses in the .NET Framework. In the context of events, a delegate is an intermediary (or pointer-like mechanism) between the event source and the code that handles the event. You associate a delegate with an event by including the delegate type in the event declaration, as shown in the example in the previous section. For more information about delegates, see the Delegate class.

The .NET Framework provides the EventHandler and EventHandler(Of TEventArgs) delegates to support most event scenarios. Use the EventHandler delegate for all events that do not include event data. Use the EventHandler(Of TEventArgs) delegate for events that include data about the event. These delegates have no return type value and take two parameters (an object for the source of the event and an object for event data).

Delegates are multicast, which means that they can hold references to more than one event-handling method. For details, see the Delegate reference page. Delegates provide flexibility and fine-grained control in event handling. A delegate acts as an event dispatcher for the class that raises the event by maintaining a list of registered event handlers for the event.

For scenarios where the EventHandler and EventHandler(Of TEventArgs) delegates do not work, you can define a delegate. Scenarios that require you to define a delegate are very rare, such as when you must work with code that does not recognize generics. You mark a delegate with the **delegate** in (C#) and **Delegate** (in Visual Basic) keyword in the declaration. The following example shows how to declare a delegate named `ThresholdReachedEventHandler`.

VB

```
Public Delegate Sub ThresholdReachedEventHandler(e As ThresholdReachedEventArgs)
```

Event Data

Data that is associated with an event can be provided through an event data class. The .NET Framework provides many event data classes that you can use in your applications. For example, the SerialDataReceivedEventArgs class is the event data class for the SerialPort.DataReceived event. The .NET Framework follows a naming pattern of ending all event data classes with **EventArgs**. You determine which event data class is associated with an event by looking at the delegate for the event. For example, the SerialDataReceivedEventHandler delegate includes the SerialDataReceivedEventArgs class as one of its parameters.

The EventArgs class is the base type for all event data classes. EventArgs is also the class you use when an event does not have any data associated with it. When you create an event that is only meant to notify other classes that something happened and does not need to pass any data, include the EventArgs class as the second parameter in the delegate. You can pass the EventArgs.Empty value when no data is provided. The EventHandler delegate includes the EventArgs class as a parameter.

When you want to create a customized event data class, create a class that derives from EventArgs, and then provide any members needed to pass data that is related to the event. Typically, you should use the same naming pattern as the .NET Framework and end your event data class name with **EventArgs**.

The following example shows an event data class named `ThresholdReachedEventArgs`. It contains properties that are

specific to the event being raised.

VB

```
Public Class ThresholdReachedEventArgs
    Inherits EventArgs

    Public Property Threshold As Integer
    Public Property TimeReached As DateTime
End Class
```

Event Handlers

To respond to an event, you define an event handler method in the event receiver. This method must match the signature of the delegate for the event you are handling. In the event handler, you perform the actions that are required when the event is raised, such as collecting user input after the user clicks a button. To receive notifications when the event occurs, your event handler method must subscribe to the event.

The following example shows an event handler method named `c_ThresholdReached` that matches the signature for the EventHandler delegate. The method subscribes to the `ThresholdReached` event.

VB

```
Module Module1

    Sub Main()
        Dim c As Counter = New Counter()
        AddHandler c.ThresholdReached, AddressOf c_ThresholdReached

        ' provide remaining implementation for the class
    End Sub

    Sub c_ThresholdReached(sender As Object, e As EventArgs)
        Console.WriteLine("The threshold was reached.")
    End Sub
End Module
```

Static and Dynamic Event Handlers

The .NET Framework allows subscribers to register for event notifications either statically or dynamically. Static event handlers are in effect for the entire life of the class whose events they handle. Dynamic event handlers are explicitly activated and deactivated during program execution, usually in response to some conditional program logic. For example, they can be used if event notifications are needed only under certain conditions or if an application provides multiple event handlers and run-time conditions define the appropriate one to use. The example in the previous section shows how to dynamically add an event handler. For more information, see Events (Visual Basic) and Events (C# Programming Guide).

Raising Multiple Events

If your class raises multiple events, the compiler generates one field per event delegate instance. If the number of events is large, the storage cost of one field per delegate may not be acceptable. For those situations, the .NET Framework provides event properties that you can use with another data structure of your choice to store event delegates.

Event properties consist of event declarations accompanied by event accessors. Event accessors are methods that you define to add or remove event delegate instances from the storage data structure. Note that event properties are slower than event fields, because each event delegate must be retrieved before it can be invoked. The trade-off is between memory and speed. If your class defines many events that are infrequently raised, you will want to implement event properties. For more information, see How to: Handle Multiple Events Using Event Properties.

Related Topics

Title	Description
How to: Raise and Consume Events	Contains examples of raising and consuming events.
How to: Handle Multiple Events Using Event Properties	Shows how to use event properties to handle multiple events.
Observer Design Pattern	Describes the design pattern that enables a subscriber to register with, and receive notifications from, a provider.
How to: Consume Events in a Web Forms Application	Shows how to handle an event that is raised by a Web Forms control.

See Also

EventHandler
EventHandler(Of TEventArgs)
EventArgs
Delegate
Events and routed events overview (Windows store apps)
Events (Visual Basic)
Events (C# Programming Guide)

Creating Event Handlers in Windows Forms

.NET Framework (current version)

An event handler is a procedure in your code that determines what actions are performed when an event occurs, such as when the user clicks a button or a message queue receives a message. When an event is raised, the event handler or handlers that receive the event are executed. Events can be assigned to multiple handlers, and the methods that handle particular events can be changed dynamically. You can also use the Windows Forms Designer to create event handlers.

In This Section

Events Overview (Windows Forms)
: Explains the event model and the role of delegates.

Event Handlers Overview (Windows Forms)
: Describes how to handle events.

How to: Create Event Handlers at Run Time for Windows Forms
: Gives directions for responding to system or user events dynamically.

How to: Connect Multiple Events to a Single Event Handler in Windows Forms
: Gives directions for assigning the same functionality to multiple controls through events.

Order of Events in Windows Forms
: Describes the order in which events are raised in Windows Forms controls.

8461e9b8-14e8-406f-936e-3726732b23d2
: Describes how to use the Windows Forms Designer to create event handlers.

Related Sections

Handling and Raising Events
: Provides links to topics on handling and raising events using the .NET Framework.

Troubleshooting Inherited Event Handlers in Visual Basic
: Lists common issues that occur with event handlers in inherited components.

How to: Create Event Handlers at Run Time for Windows Forms

.NET Framework (current version)

In addition to creating events using the Windows Forms Designer, you can also create an event handler at run time. This action allows you to connect event handlers based on conditions in code at run time as opposed to having them connected when the program initially starts.

To create an event handler at run time

1. Open the form in the Code Editor that you want to add an event handler to.
2. Add a method to your form with the method signature for the event that you want to handle.

   For example, if you were handling the Click event of a Button control, you would create a method such as the following:

   VB

   ```
   Private Sub Button1_Click(ByVal sender As Object, ByVal e As EventArgs)
      ' Add event handler code here.
   End Sub
   ```

3. Add code to the event handler as appropriate to your application.
4. Determine which form or control you want to create an event handler for.
5. In a method within your form's class, add code that specifies the event handler to handle the event. For example, the following code specifies the event handler `button1_Click` handles the Click event of a Button control:

   VB

   ```
   AddHandler Button1.Click, AddressOf Button1_Click
   ```

   The AddHandler method demonstrated in the Visual Basic code above establishes a click event handler for the button.

See Also

Creating Event Handlers in Windows Forms
Event Handlers Overview (Windows Forms)

Troubleshooting Inherited Event Handlers in Visual Basic

How to: Connect Multiple Events to a Single Event Handler in Windows Forms

.NET Framework (current version)

In your application design, you may find it necessary to use a single event handler for multiple events or have multiple events perform the same procedure. For example, it is often a powerful time-saver to have a menu command raise the same event as a button on your form does if they expose the same functionality. You can do this by using the Events view of the Properties window in C# or using the `Handles` keyword and the **Class Name** and **Method Name** drop-down boxes in the Visual Basic Code Editor.

To connect multiple events to a single event handler in Visual Basic

1. Right-click the form and choose **View Code**.
2. From the **Class Name** drop-down box, select one of the controls that you want to have the event handler handle.
3. From the **Method Name** drop-down box, select one of the events that you want the event handler to handle.
4. The Code Editor inserts the appropriate event handler and positions the insertion point within the method. In the example below, it is the Click event for the Button control.

VB

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click
' Add event-handler code here.
End Sub
```

5. Append the other events you would like handled to the `Handles` clause.

VB

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click, Button2.Click
' Add event-handler code here.
End Sub
```

6. Add the appropriate code to the event handler.

To connect multiple events to a single event handler in C#

1. Select the control to which you want to connect an event handler.

2. In the Properties window, click the **Events** button ().
3. Click the name of the event that you want to handle.
4. In the value section next to the event name, click the drop-down button to display a list of existing event handlers that match the method signature of the event you want to handle.
5. Select the appropriate event handler from the list.

   Code will be added to the form to bind the event to the existing event handler.

See Also

Creating Event Handlers in Windows Forms
Event Handlers Overview (Windows Forms)

Order of Events in Windows Forms

.NET Framework (current version)

The order in which events are raised in Windows Forms applications is of particular interest to developers concerned with handling each of these events in turn. When a situation calls for meticulous handling of events, such as when you are redrawing parts of the form, an awareness of the precise order in which events are raised at run time is necessary. This topic provides some details on the order of events during several important stages in the lifetime of applications and controls. For specific details about the order of mouse input events, see Mouse Events in Windows Forms. For an overview of events in Windows Forms, see Events Overview (Windows Forms). For details about the makeup of event handlers, see Event Handlers Overview (Windows Forms).

Application Startup and Shutdown Events

The Form and Control classes expose a set of events related to application startup and shutdown. When a Windows Forms application starts, the startup events of the main form are raised in the following order:

- Control.HandleCreated
- Control.BindingContextChanged
- Form.Load
- Control.VisibleChanged
- Form.Activated
- Form.Shown

When an application closes, the shutdown events of the main form are raised in the following order:

- Form.Closing
- Form.FormClosing
- Form.Closed
- Form.FormClosed
- Form.Deactivate

The ApplicationExit event of the Application class is raised after the shutdown events of the main form.

Note

Visual Basic 2005 includes additional application events, such as WindowsFormsApplicationBase.Startup and WindowsFormsApplicationBase.Shutdown.

Focus and Validation Events

When you change the focus by using the keyboard (TAB, SHIFT+TAB, and so on), by calling the Select or SelectNextControl methods, or by setting the ActiveControl property to the current form, focus events of the Control class occur in the following order:

- Enter
- GotFocus
- Leave
- Validating
- Validated
- LostFocus

When you change the focus by using the mouse or by calling the Focus method, focus events of the Control class occur in the following order:

- Enter
- GotFocus
- LostFocus
- Leave
- Validating
- Validated

See Also

Creating Event Handlers in Windows Forms