


Handling and raising events

03/29/2017 • 7 minutes to read •  +9

In this article

[Events](#)

[Delegates](#)

[Event data](#)

[Event handlers](#)

[Static and dynamic event handlers](#)

[Raising multiple events](#)

[Related topics](#)

[See also](#)

Events in .NET 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](#).

Events

An event is a message sent by an object to signal the occurrence of an action. The action can be caused by user interaction, such as a button click, or it can result from 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 C# [event](#) or the Visual Basic [Event](#) 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 *onEventName*; for example, `OnDataReceived`. The method should take one parameter that specifies an event data object, which is an object of type [EventArgs](#) or a derived type. You provide this method to enable derived classes to override the logic for raising the event. A derived class should always call the *onEventName* 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`.

C#

Copy

```
class Counter
{
    public event EventHandler ThresholdReached;

    protected virtual void OnThresholdReached(EventArgs e)
    {
        EventHandler handler = ThresholdReached;
        handler?.Invoke(this, e);
    }

    // provide remaining implementation for the 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 it 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 .NET. 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.

.NET provides the [EventHandler](#) and [EventHandler<TEventArgs>](#) delegates to support most event scenarios. Use the [EventHandler](#) delegate for all events that do not include event data. Use the [EventHandler<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<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 C# [delegate](#) and Visual Basic [Delegate](#) keyword in the declaration. The following example shows how to declare a delegate named ThresholdReachedEventHandler.

C#

Copy

```
public delegate void ThresholdReachedEventHandler(object sender,
ThresholdReachedEventArgs e);
```


Event data

Data that is associated with an event can be provided through an event data class. .NET 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. .NET 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 .NET 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.

C#	 Copy
<pre>public class ThresholdReachedEventArgs : EventArgs { public int Threshold { get; set; } public DateTime TimeReached { get; set; } }</pre>	

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.

C#	 Copy
<pre>class Program { static void Main() { var c = new Counter(); c.ThresholdReached += c_ThresholdReached; // provide remaining implementation for the class } }</pre>	

```
static void c_ThresholdReached(object sender, EventArgs e)
{
    Console.WriteLine("The threshold was reached.");
}
```

Static and dynamic event handlers

.NET 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](#) (in Visual Basic) and [Events](#) (in C#).

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, .NET 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<TEventArgs>](#)
- [EventArgs](#)
- [Delegate](#)
- [Events \(Visual Basic\)](#)
- [Events \(C# Programming Guide\)](#)
- [Events and routed events overview \(UWP apps\)](#)

Is this page helpful?

 Yes  No
