.NET EVENTS

The following procedure demonstrates how to add events that follow the standard .NET Framework pattern to your classes and structs. All events in the .NET Framework class library are based on the EventHandler delegate, which is defined as follows:

```
public delegate void EventHandler(object sender, EventArgs e);
```

The .NET Framework 2.0 introduces a generic version of this delegate, EventHandler<TEventArgs>.

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.

To define an event, you use the **event** keyword in the signature of your event class, and specify the type of delegate for the event.

Typically, to raise an event, you add a method that is marked as **protected** and **virtual**. Name this method **OnEventName**; for example, OnDataReceived. The method should take one parameter that specifies an event data object. 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.

```
class Counter
{
    public event EventHandler ThresholdReached;//ThresholdReached EventName

    protected virtual void OnThresholdReached(EventArgs e)
    {
        EventHandler handler = ThresholdReached;
        if (handler != null)
        {
            handler(this, e);
        }
    }

    // provide remaining implementation for the class
}
```

Delegates

The .NET Framework 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).

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 **delegate** keyword in the declaration.

Event Data

Data that is associated with an event can be provided through an event data class. The .NET Framework follows a naming pattern of ending all event data classes with **EventArgs**.

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.

The following example shows an event data class named ThresholdReachedEventArgs. It contains properties that are specific to the event being raised.

```
public class ThresholdReachedEventArgs : EventArgs
{
    public int Threshold { get; set; }
    public DateTime TimeReached { get; set; }
}
```

The first example shows how to raise and consume an event that doesn't have data.

It contains a class named Counter that has an event named ThresholdReached. This event is raised when a counter value equals or exceeds a threshold value. The EventHandler delegate is associated with the event, because no event data is provided.

Event class

```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Counter c = new Counter(new Random().Next(10));
            c.ThresholdReached += c_ThresholdReached;

            Console.WriteLine("press 'a' key to increase total");
            while (Console.ReadKey(true).KeyChar == 'a')
            {
                Console.WriteLine("adding one");
                c.Add(1);
            }
        }

        static void c_ThresholdReached(object sender, EventArgs e)
        {
            Console.WriteLine("The threshold was reached.");
            Environment.Exit(0);
        }
    }
class Counter
    {
        private int threshold;
        private int total;

        public Counter(int passedThreshold)
        {
            threshold = passedThreshold;
        }

        public void Add(int x)
        {
            total += x;
            if (total >= threshold)
            {
                OnThresholdReached(EventArgs.Empty);
            }
        }

        protected virtual void OnThresholdReached(EventArgs e)
        {
            EventHandler handler = ThresholdReached;
```

```
            if (handler != null)
            {
                handler(this, e);
            }
        }

        public event EventHandler ThresholdReached;
    }
}
```

The second example shows how to raise and consume an event that provides data.

The EventHandler<TEventArgs> delegate is associated with the event, and an instance of a custom event data object is provided.

```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Counter c = new Counter(new Random().Next(10));
            c.ThresholdReached += c_ThresholdReached;

            Console.WriteLine("press 'a' key to increase total");
            while (Console.ReadKey(true).KeyChar == 'a')
            {
                Console.WriteLine("adding one");
                c.Add(1);
            }
        }

   static void c_ThresholdReached(object sender, ThresholdReachedEventArgs e)
        {
            Console.WriteLine("The threshold of {0} was reached at {1}.",
                e.Threshold,  e.TimeReached);
            Environment.Exit(0);
        }
    }//end class Program

    class Counter
    {
        private int threshold;
        private int total;

        public Counter(int passedThreshold)
        {
            threshold = passedThreshold;
        }

        public void Add(int x)
        {
```

```
            total += x;
            if (total >= threshold)
            {
                ThresholdReachedEventArgs args = new
                                        ThresholdReachedEventArgs();
                args.Threshold = threshold;
                args.TimeReached = DateTime.Now;
                OnThresholdReached(args);
            }
        }

 protected virtual void OnThresholdReached(ThresholdReachedEventArgs e)
 {
    EventHandler<ThresholdReachedEventArgs> handler = ThresholdReached;
            if (handler != null)
            {
                handler(this, e);
            }
 }

     public event EventHandler<ThresholdReachedEventArgs> ThresholdReached;

}//end class Counter

    public class ThresholdReachedEventArgs : EventArgs
    {
        public int Threshold { get; set; }
        public DateTime TimeReached { get; set; }
    }
}
```

The third example shows how to declare a delegate for an event.

The delegate is named ThresholdReachedEventHandler. This is just an illustration. Typically, you do not have to declare a delegate for an event, because you can use either the EventHandler or the EventHandler<TEventArgs> delegate. You should declare a delegate only in rare scenarios, such as making your class available to legacy code that cannot use generics.

```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Counter c = new Counter(new Random().Next(10));
            c.ThresholdReached += c_ThresholdReached;

            Console.WriteLine("press 'a' key to increase total");
            while (Console.ReadKey(true).KeyChar == 'a')
            {
                Console.WriteLine("adding one");
                c.Add(1);
            }
```

```
        }

    static void c_ThresholdReached(Object sender, ThresholdReachedEventArgs e)
        {
            Console.WriteLine("The threshold of {0} was reached at {1}.",
                              e.Threshold, e.TimeReached);
            Environment.Exit(0);
        }
    }

    class Counter
    {
        private int threshold;
        private int total;

        public Counter(int passedThreshold)
        {
            threshold = passedThreshold;
        }

        public void Add(int x)
        {
            total += x;
            if (total >= threshold)
            {
                ThresholdReachedEventArgs args = new
                                                 ThresholdReachedEventArgs();
                args.Threshold = threshold;
                args.TimeReached = DateTime.Now;
                OnThresholdReached(args);
            }
        }

       protected virtual void OnThresholdReached(ThresholdReachedEventArgs e)
        {
            ThresholdReachedEventHandler handler = ThresholdReached;
            if (handler != null)
            {
                handler(this, e);
            }
        }

        public event ThresholdReachedEventHandler ThresholdReached;
    }

    public class ThresholdReachedEventArgs : EventArgs
    {
        public int Threshold { get; set; }
        public DateTime TimeReached { get; set; }
    }

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