Contents

Events (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Events enable a class or object to notify other classes or objects when something of interest occurs. The class that sends (or *raises*) the event is called the *publisher* and the classes that receive (or *handle*) the event are called *subscribers*.

In a typical C# Windows Forms or Web application, you subscribe to events raised by controls such as buttons and list boxes. You can use the Visual C# integrated development environment (IDE) to browse the events that a control publishes and select the ones that you want to handle. The IDE provides an easy way to automatically add an empty event handler method and the code to subscribe to the event. For more information, see How to: Subscribe to and Unsubscribe from Events.

Events Overview

Events have the following properties:

- The publisher determines when an event is raised; the subscribers determine what action is taken in response to the event.
- An event can have multiple subscribers. A subscriber can handle multiple events from multiple publishers.
- Events that have no subscribers are never raised.
- Events are typically used to signal user actions such as button clicks or menu selections in graphical user interfaces.
- When an event has multiple subscribers, the event handlers are invoked synchronously when an event is raised. To invoke events asynchronously, see Calling Synchronous Methods Asynchronously.
- In the .NET Framework class library, events are based on the EventHandler delegate and the EventArgs base class.

Related Sections

For more information, see:

- How to: Subscribe to and Unsubscribe from Events
- How to: Publish Events that Conform to .NET Framework Guidelines
- How to: Raise Base Class Events in Derived Classes
- How to: Implement Interface Events
- How to: Use a Dictionary to Store Event Instances
- How to: Implement Custom Event Accessors

C# Language Specification

For more information, see Events in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

Featured Book Chapters

Delegates, Events, and Lambda Expressions in C# 3.0 Cookbook, Third Edition: More than 250 solutions for C# 3.0 programmers

Delegates and Events in Learning C# 3.0: Master the fundamentals of C# 3.0

See also

- EventHandler
- C# Programming Guide
- Delegates
- Creating Event Handlers in Windows Forms

How to: Subscribe to and Unsubscribe from Events (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

You subscribe to an event that is published by another class when you want to write custom code that is called when that event is raised. For example, you might subscribe to a button's `click` event in order to make your application do something useful when the user clicks the button.

To subscribe to events by using the Visual Studio IDE

1. If you cannot see the **Properties** window, in **Design** view, right-click the form or control for which you want to create an event handler, and select **Properties**.

2. On top of the **Properties** window, click the **Events** icon.

3. Double-click the event that you want to create, for example the `Load` event.

   Visual C# creates an empty event handler method and adds it to your code. Alternatively you can add the code manually in **Code** view. For example, the following lines of code declare an event handler method that will be called when the `Form` class raises the `Load` event.

   ```
   private void Form1_Load(object sender, System.EventArgs e)
   {
       // Add your form load event handling code here.
   }
   ```

   The line of code that is required to subscribe to the event is also automatically generated in the `InitializeComponent` method in the Form1.Designer.cs file in your project. It resembles this:

   ```
   this.Load += new System.EventHandler(this.Form1_Load);
   ```

To subscribe to events programmatically

1. Define an event handler method whose signature matches the delegate signature for the event. For example, if the event is based on the EventHandler delegate type, the following code represents the method stub:

   ```
   void HandleCustomEvent(object sender, CustomEventArgs a)
   {
      // Do something useful here.
   }
   ```

2. Use the addition assignment operator (`+=`) to attach your event handler to the event. In the following example, assume that an object named `publisher` has an event named `RaiseCustomEvent`. Note that the subscriber class needs a reference to the publisher class in order to subscribe to its events.

   ```
   publisher.RaiseCustomEvent += HandleCustomEvent;
   ```

   Note that the previous syntax is new in C# 2.0. It is exactly equivalent to the C# 1.0 syntax in which the encapsulating delegate must be explicitly created by using the `new` keyword:

```
publisher.RaiseCustomEvent += new CustomEventHandler(HandleCustomEvent);
```

An event handler can also be added by using a lambda expression:

```
public Form1()
{
    InitializeComponent();
    // Use a lambda expression to define an event handler.
    this.Click += (s,e) => { MessageBox.Show(
       ((MouseEventArgs)e).Location.ToString());};
}
```

For more information, see How to: Use Lambda Expressions Outside LINQ.

To subscribe to events by using an anonymous method

- If you will not have to unsubscribe to an event later, you can use the addition assignment operator (`+=`) to attach an anonymous method to the event. In the following example, assume that an object named `publisher` has an event named `RaiseCustomEvent` and that a `CustomEventArgs` class has also been defined to carry some kind of specialized event information. Note that the subscriber class needs a reference to `publisher` in order to subscribe to its events.

```
publisher.RaiseCustomEvent += delegate(object o, CustomEventArgs e)
{
  string s = o.ToString() + " " + e.ToString();
  Console.WriteLine(s);
};
```

It is important to notice that you cannot easily unsubscribe from an event if you used an anonymous function to subscribe to it. To unsubscribe in this scenario, it is necessary to go back to the code where you subscribe to the event, store the anonymous method in a delegate variable, and then add the delegate to the event. In general, we recommend that you do not use anonymous functions to subscribe to events if you will have to unsubscribe from the event at some later point in your code. For more information about anonymous functions, see Anonymous Functions.

Unsubscribing

To prevent your event handler from being invoked when the event is raised, unsubscribe from the event. In order to prevent resource leaks, you should unsubscribe from events before you dispose of a subscriber object. Until you unsubscribe from an event, the multicast delegate that underlies the event in the publishing object has a reference to the delegate that encapsulates the subscriber's event handler. As long as the publishing object holds that reference, garbage collection will not delete your subscriber object.

To unsubscribe from an event

- Use the subtraction assignment operator (`-=`) to unsubscribe from an event:

```
publisher.RaiseCustomEvent -= HandleCustomEvent;
```

When all subscribers have unsubscribed from an event, the event instance in the publisher class is set to `null`.

See also

- Events

- event
- How to: Publish Events that Conform to .NET Framework Guidelines
- -= Operator (C# Reference)
- += Operator

How to: Publish Events that Conform to .NET Framework Guidelines (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

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);
```

NOTE

The .NET Framework 2.0 introduces a generic version of this delegate, EventHandler<TEventArgs>. The following examples show how to use both versions.

Although events in classes that you define can be based on any valid delegate type, even delegates that return a value, it is generally recommended that you base your events on the .NET Framework pattern by using EventHandler, as shown in the following example.

To publish events based on the EventHandler pattern

1. (Skip this step and go to Step 3a if you do not have to send custom data with your event.) Declare the class for your custom data at a scope that is visible to both your publisher and subscriber classes. Then add the required members to hold your custom event data. In this example, a simple string is returned.

   ```
   public class CustomEventArgs : EventArgs
   {
       public CustomEventArgs(string s)
       {
           msg = s;
       }
       private string msg;
       public string Message
       {
           get { return msg; }
       }
   }
   ```

2. (Skip this step if you are using the generic version of EventHandler<TEventArgs> .) Declare a delegate in your publishing class. Give it a name that ends with *EventHandler*. The second parameter specifies your custom EventArgs type.

   ```
   public delegate void CustomEventHandler(object sender, CustomEventArgs a);
   ```

3. Declare the event in your publishing class by using one of the following steps.

   a. If you have no custom EventArgs class, your Event type will be the non-generic EventHandler delegate. You do not have to declare the delegate because it is already declared in the System namespace that is included when you create your C# project. Add the following code to your publisher class.

```
public event EventHandler RaiseCustomEvent;
```

b. If you are using the non-generic version of EventHandler and you have a custom class derived from EventArgs, declare your event inside your publishing class and use your delegate from step 2 as the type.

```
public event CustomEventHandler RaiseCustomEvent;
```

c. If you are using the generic version, you do not need a custom delegate. Instead, in your publishing class, you specify your event type as `EventHandler<CustomEventArgs>`, substituting the name of your own class between the angle brackets.

```
public event EventHandler<CustomEventArgs> RaiseCustomEvent;
```

Example

The following example demonstrates the previous steps by using a custom EventArgs class and EventHandler<TEventArgs> as the event type.

```
namespace DotNetEvents
{
    using System;
    using System.Collections.Generic;

    // Define a class to hold custom event info
    public class CustomEventArgs : EventArgs
    {
        public CustomEventArgs(string s)
        {
            message = s;
        }
        private string message;

        public string Message
        {
            get { return message; }
            set { message = value; }
        }
    }

    // Class that publishes an event
    class Publisher
    {

        // Declare the event using EventHandler<T>
        public event EventHandler<CustomEventArgs> RaiseCustomEvent;

        public void DoSomething()
        {
            // Write some code that does something useful here
            // then raise the event. You can also raise an event
            // before you execute a block of code.
            OnRaiseCustomEvent(new CustomEventArgs("Did something"));

        }

        // Wrap event invocations inside a protected virtual method
        // to allow derived classes to override the event invocation behavior
        protected virtual void OnRaiseCustomEvent(CustomEventArgs e)
        {
```

```
        {
            // Make a temporary copy of the event to avoid possibility of
            // a race condition if the last subscriber unsubscribes
            // immediately after the null check and before the event is raised.
            EventHandler<CustomEventArgs> handler = RaiseCustomEvent;

            // Event will be null if there are no subscribers
            if (handler != null)
            {
                // Format the string to send inside the CustomEventArgs parameter
                e.Message += $" at {DateTime.Now}";

                // Use the () operator to raise the event.
                handler(this, e);
            }
        }
    }

    //Class that subscribes to an event
    class Subscriber
    {
        private string id;
        public Subscriber(string ID, Publisher pub)
        {
            id = ID;
            // Subscribe to the event using C# 2.0 syntax
            pub.RaiseCustomEvent += HandleCustomEvent;
        }

        // Define what actions to take when the event is raised.
        void HandleCustomEvent(object sender, CustomEventArgs e)
        {
            Console.WriteLine(id + " received this message: {0}", e.Message);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Publisher pub = new Publisher();
            Subscriber sub1 = new Subscriber("sub1", pub);
            Subscriber sub2 = new Subscriber("sub2", pub);

            // Call the method that raises the event.
            pub.DoSomething();

            // Keep the console window open
            Console.WriteLine("Press Enter to close this window.");
            Console.ReadLine();

        }
    }
}
```

See also

- Delegate
- C# Programming Guide
- Events
- Delegates

How to: Raise Base Class Events in Derived Classes (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

The following simple example shows the standard way to declare events in a base class so that they can also be raised from derived classes. This pattern is used extensively in Windows Forms classes in the .NET Framework class library.

When you create a class that can be used as a base class for other classes, you should consider the fact that events are a special type of delegate that can only be invoked from within the class that declared them. Derived classes cannot directly invoke events that are declared within the base class. Although sometimes you may want an event that can only be raised by the base class, most of the time, you should enable the derived class to invoke base class events. To do this, you can create a protected invoking method in the base class that wraps the event. By calling or overriding this invoking method, derived classes can invoke the event indirectly.

NOTE

Do not declare virtual events in a base class and override them in a derived class. The C# compiler does not handle these correctly and it is unpredictable whether a subscriber to the derived event will actually be subscribing to the base class event.

Example

```
namespace BaseClassEvents
{
    using System;
    using System.Collections.Generic;

    // Special EventArgs class to hold info about Shapes.
    public class ShapeEventArgs : EventArgs
    {
        private double newArea;

        public ShapeEventArgs(double d)
        {
            newArea = d;
        }
        public double NewArea
        {
            get { return newArea; }
        }
    }

    // Base class event publisher
    public abstract class Shape
    {
        protected double area;

        public double Area
        {
            get { return area; }
            set { area = value; }
        }
        // The event. Note that by using the generic EventHandler<T> event type
        // we do not need to declare a separate delegate type.
        public event EventHandler<ShapeEventArgs> ShapeChanged;
        public event EventHandler<ShapeEventArgs> ShapeChanged;

        public abstract void Draw();

        //The event-invoking method that derived classes can override.
        protected virtual void OnShapeChanged(ShapeEventArgs e)
        {
            // Make a temporary copy of the event to avoid possibility of
            // a race condition if the last subscriber unsubscribes
            // immediately after the null check and before the event is raised.
            EventHandler<ShapeEventArgs> handler = ShapeChanged;
            if (handler != null)
            {
                handler(this, e);
            }
        }
    }

    public class Circle : Shape
    {
        private double radius;
        public Circle(double d)
        {
            radius = d;
            area = 3.14 * radius * radius;
        }
        public void Update(double d)
        {
            radius = d;
            area = 3.14 * radius * radius;
            OnShapeChanged(new ShapeEventArgs(area));
        }
        protected override void OnShapeChanged(ShapeEventArgs e)
        {
            // Do any circle-specific processing here.

            // Call the base class event invocation method.
            base.OnShapeChanged(e);
        }
        public override void Draw()
        {
            Console.WriteLine("Drawing a circle");
        }
    }

    public class Rectangle : Shape
    {
        private double length;
        private double width;
        public Rectangle(double length, double width)
        {
            this.length = length;
            this.width = width;
            area = length * width;
        }
        public void Update(double length, double width)
        {
            this.length = length;
            this.width = width;
            area = length * width;
            OnShapeChanged(new ShapeEventArgs(area));
        }
        protected override void OnShapeChanged(ShapeEventArgs e)
        {
            // Do any rectangle-specific processing here.

            // Call the base class event invocation method.
            base.OnShapeChanged(e);
        }
        public override void Draw()
        public override void Draw()
        {
            Console.WriteLine("Drawing a rectangle");
        }

    }

    // Represents the surface on which the shapes are drawn
    // Subscribes to shape events so that it knows
    // when to redraw a shape.
    public class ShapeContainer
    {
        List<Shape> _list;

        public ShapeContainer()
        {
            _list = new List<Shape>();
        }

        public void AddShape(Shape s)
        {
            _list.Add(s);
            // Subscribe to the base class event.
            s.ShapeChanged += HandleShapeChanged;
        }

        // ...Other methods to draw, resize, etc.

        private void HandleShapeChanged(object sender, ShapeEventArgs e)
        {
            Shape s = (Shape)sender;

            // Diagnostic message for demonstration purposes.
            Console.WriteLine("Received event. Shape area is now {0}", e.NewArea);

            // Redraw the shape here.
            s.Draw();
        }
    }

    class Test
    {

        static void Main(string[] args)
        {
            //Create the event publishers and subscriber
            Circle c1 = new Circle(54);
            Rectangle r1 = new Rectangle(12, 9);
            ShapeContainer sc = new ShapeContainer();

            // Add the shapes to the container.
            sc.AddShape(c1);
            sc.AddShape(r1);

            // Cause some events to be raised.
            c1.Update(57);
            r1.Update(7, 7);

            // Keep the console window open in debug mode.
            System.Console.WriteLine("Press any key to exit.");
            System.Console.ReadKey();
        }
    }
}
/* Output:
        Received event. Shape area is now 10201.86
        Drawing a circle
        Received event. Shape area is now 49
        Drawing a rectangle
```

```
  */
```

See also

- C# Programming Guide
- Events
- Delegates
- Access Modifiers
- Creating Event Handlers in Windows Forms

How to: Implement Interface Events (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

An interface can declare an event. The following example shows how to implement interface events in a class. Basically the rules are the same as when you implement any interface method or property.

To implement interface events in a class

Declare the event in your class and then invoke it in the appropriate areas.

```
namespace ImplementInterfaceEvents
{
    public interface IDrawingObject
    {
        event EventHandler ShapeChanged;
    }
    public class MyEventArgs : EventArgs
    {
        // class members
    }
    public class Shape : IDrawingObject
    {
        public event EventHandler ShapeChanged;
        void ChangeShape()
        {
            // Do something here before the event…

            OnShapeChanged(new MyEventArgs(/*arguments*/));

            // or do something here after the event.
        }
        protected virtual void OnShapeChanged(MyEventArgs e)
        {
            ShapeChanged?.Invoke(this, e);
        }
    }

}
```

Example

The following example shows how to handle the less-common situation in which your class inherits from two or more interfaces and each interface has an event with the same name. In this situation, you must provide an explicit interface implementation for at least one of the events. When you write an explicit interface implementation for an event, you must also write the `add` and `remove` event accessors. Normally these are provided by the compiler, but in this case the compiler cannot provide them.

By providing your own accessors, you can specify whether the two events are represented by the same event in your class, or by different events. For example, if the events should be raised at different times according to the interface specifications, you can associate each event with a separate implementation in your class. In the following example, subscribers determine which `OnDraw` event they will receive by casting the shape reference to either an `IShape` or an `IDrawingObject`.

```
namespace WrapTwoInterfaceEvents
{
    using System;

    public interface IDrawingObject
    {
        // Raise this event before drawing
        // the object.
        event EventHandler OnDraw;
    }
    public interface IShape
    {
        // Raise this event after drawing
        // the shape.
        event EventHandler OnDraw;
    }


    // Base class event publisher inherits two
    // interfaces, each with an OnDraw event
    public class Shape : IDrawingObject, IShape
    {
        // Create an event for each interface event
        event EventHandler PreDrawEvent;
        event EventHandler PostDrawEvent;

        object objectLock = new Object();

        // Explicit interface implementation required.
        // Associate IDrawingObject's event with
        // PreDrawEvent
        event EventHandler IDrawingObject.OnDraw
        {
            add
            {
                lock (objectLock)
                {
                    PreDrawEvent += value;
                }
            }
            remove
            {
                lock (objectLock)
                {
                    PreDrawEvent -= value;
                }
            }
        }

        // Explicit interface implementation required.
        // Associate IShape's event with
        // PostDrawEvent
        event EventHandler IShape.OnDraw
        {
            add
            {
                lock (objectLock)
                {
                    PostDrawEvent += value;
                }
            }
            remove
            {
                lock (objectLock)
                {
                    PostDrawEvent -= value;
                }
            }
        }

        // For the sake of simplicity this one method
        // implements both interfaces.
        public void Draw()
        {
            // Raise IDrawingObject's event before the object is drawn.
            PreDrawEvent?.Invoke(this, EventArgs.Empty);

            Console.WriteLine("Drawing a shape.");

            // Raise IShape's event after the object is drawn.
            PostDrawEvent?.Invoke(this, EventArgs.Empty);
        }
    }
    public class Subscriber1
    {
        // References the shape object as an IDrawingObject
        public Subscriber1(Shape shape)
        {
            IDrawingObject d = (IDrawingObject)shape;
            d.OnDraw += d_OnDraw;
        }

        void d_OnDraw(object sender, EventArgs e)
        {
            Console.WriteLine("Sub1 receives the IDrawingObject event.");
        }
    }
    // References the shape object as an IShape
    public class Subscriber2
    {
        public Subscriber2(Shape shape)
        {
            IShape d = (IShape)shape;
            d.OnDraw += d_OnDraw;
        }

        void d_OnDraw(object sender, EventArgs e)
        {
            Console.WriteLine("Sub2 receives the IShape event.");
        }
    }


    public class Program
    {
        static void Main(string[] args)
        {
            Shape shape = new Shape();
            Subscriber1 sub = new Subscriber1(shape);
            Subscriber2 sub2 = new Subscriber2(shape);
            shape.Draw();

            // Keep the console window open in debug mode.
            System.Console.WriteLine("Press any key to exit.");
            System.Console.ReadKey();
        }
    }

}
/* Output:
    Sub1 receives the IDrawingObject event.
    Drawing a shape.
    Sub2 receives the IShape event.
*/
```

See also

- C# Programming Guide
- Events
- Delegates
- Explicit Interface Implementation
- How to: Raise Base Class Events in Derived Classes

How to: Use a Dictionary to Store Event Instances (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

One use for `accessor-declarations` is to expose many events without allocating a field for each event, but instead using a Dictionary to store the event instances. This is only useful if you have many events, but you expect most of the events will not be implemented.

Example

```
using System;
using System.Collections.Generic;

public delegate void EventHandler1(int i);

public delegate void EventHandler2(string s);

public class PropertyEventsSample
{
    private readonly Dictionary<string, Delegate> _eventTable;
    private readonly List<EventHandler1> _event1List = new List<EventHandler1>();
    private readonly List<EventHandler2> _event2List = new List<EventHandler2>();
    public PropertyEventsSample()
    {
        _eventTable = new Dictionary<string, Delegate>
        {
            {"Event1", null},
            {"Event2", null}
        };
    }

    public event EventHandler1 Event1
    {
        add
        {
            _event1List.Add(value);
            lock (_eventTable)
            {
                _eventTable["Event1"] = (EventHandler1) _eventTable["Event1"] + value;
            }
        }
        remove
        {
            if (!_event1List.Contains(value)) return;
            _event1List.Remove(value);
            lock (_eventTable)
            {
                _eventTable["Event1"] = null;
                foreach (var event1 in _event1List)
                {
                    _eventTable["Event1"] = (EventHandler1) _eventTable["Event1"] + event1;
                }
            }
        }
    }

    public event EventHandler2 Event2
    {
        add
        {
            _event2List.Add(value);
            lock (_eventTable)
            {
                _eventTable["Event2"] = (EventHandler2) _eventTable["Event2"] + value;
            }
        }
        remove
        {
            if (!_event2List.Contains(value)) return;
            _event2List.Remove(value);
            lock (_eventTable)
            {
                _eventTable["Event2"] = null;
                foreach (var event2 in _event2List)
                {
                    _eventTable["Event2"] = (EventHandler2) _eventTable["Event2"] + event2;
                }
            }
        }
    }

    internal void RaiseEvent1(int i)
    {
        lock (_eventTable)
        {
            var handler1 = (EventHandler1) _eventTable["Event1"];
            handler1?.Invoke(i);
        }
    }

    internal void RaiseEvent2(string s)
    {
        lock (_eventTable)
        {
            var handler2 = (EventHandler2) _eventTable["Event2"];
            handler2?.Invoke(s);
        }
    }
}

public static class TestClass
{
    private static void Delegate1Method(int i)
    {
        Console.WriteLine(i);
    }

    private static void Delegate2Method(string s)
    {
        Console.WriteLine(s);
    }

    private static void Main()
    {
        var p = new PropertyEventsSample();

        p.Event1 += Delegate1Method;
        p.Event1 += Delegate1Method;
        p.Event1 -= Delegate1Method;
        p.RaiseEvent1(2);

        p.Event2 += Delegate2Method;
        p.Event2 += Delegate2Method;
        p.Event2 -= Delegate2Method;
        p.RaiseEvent2("TestString");

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    2
    TestString
*/
```

See also

- C# Programming Guide
- Events
- Delegates

How to: Implement Custom Event Accessors (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

An event is a special kind of multicast delegate that can only be invoked from within the class that it is declared in. Client code subscribes to the event by providing a reference to a method that should be invoked when the event is fired. These methods are added to the delegate's invocation list through event accessors, which resemble property accessors, except that event accessors are named `add` and `remove`. In most cases, you do not have to supply custom event accessors. When no custom event accessors are supplied in your code, the compiler will add them automatically. However, in some cases you may have to provide custom behavior. One such case is shown in the topic How to: Implement Interface Events.

Example

The following example shows how to implement custom add and remove event accessors. Although you can substitute any code inside the accessors, we recommend that you lock the event before you add or remove a new event handler method.

```
event EventHandler IDrawingObject.OnDraw
{
    add
    {
        lock (objectLock)
        {
            PreDrawEvent += value;
        }
    }
    remove
    {
        lock (objectLock)
        {
            PreDrawEvent -= value;
        }
    }
}
```

See also

- Events
- event