Events in C#

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.

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.
- In the .NET class library, events are based on the `EventHandler` delegate and the `EventArgs` base class.

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 message)
    {
        Message = message;
    }
    public string Message { get; set; }
}
```

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
                                                                    args);
```

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-1:

```
using System;
namespace DotNetEvents
{
    // Define a class to hold custom event info
    public class CustomEventArgs : EventArgs
    {
        public CustomEventArgs(string message)
        {
            Message = message;
        }
        public string Message { get; set; }
    }
    // 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("Event triggered"));
        }
        // 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> raiseEvent = RaiseCustomEvent;
            // Event will be null if there are no subscribers
            if (raiseEvent != null)
```

```
            {
                // Format the string to send inside the CustomEventArgs parameter
                e.Message += $" at {DateTime.Now}";
                // Call to raise the event.
                raiseEvent(this, e);
            }
        }
    }
    //Class that subscribes to an event
    class Subscriber
    {
        private readonly string _id;
        public Subscriber(string id, Publisher pub)
        {
            _id = id;
            // Subscribe to the event
            pub.RaiseCustomEvent += HandleCustomEvent;
        }
        // Define what actions to take when the event is raised.
        void HandleCustomEvent(object sender, CustomEventArgs e)
        {
            Console.WriteLine($"{_id} received this message: {e.Message}");
        }
    }
    class Program
    {
        static void Main()
        {
            var pub = new Publisher();
            var sub1 = new Subscriber("sub1", pub);
            var sub2 = new Subscriber("sub2", pub);
            // Call the method that raises the event.
            pub.DoSomething();
            // Keep the console window open
            Console.WriteLine("Press any key to continue...");
            Console.ReadLine();
        }
    }
}
```

Example-2:

```
namespace BaseClassEvents
{
    // Special EventArgs class to hold info about Shapes.
    public class ShapeEventArgs : EventArgs
    {
        public ShapeEventArgs(double area)
        {
            NewArea = area;
        }
        public double NewArea { get; }
```

```
    }
    // Base class event publisher
    public abstract class Shape
    {
        protected double _area;
        public double Area
        {
            get => _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 abstract void Draw();
        //The event-invoking method that derived classes can override.
        protected virtual void OnShapeChanged(ShapeEventArgs e)
        {
            // Safely raise the event for all subscribers
            ShapeChanged?.Invoke(this, e);
        }
    }
    public class Circle : Shape
    {
        private double _radius;
        public Circle(double radius)
        {
            _radius = radius;
            _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)
        {
```

```
            _length = length;
            _width = width;
            _area = _length * _width;
        }
        public void Update(double length, double width)
        {
            _length = length;
            _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()
        {
            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
    {
        private readonly List<Shape> _list;
        public ShapeContainer()
        {
            _list = new List<Shape>();
        }
        public void AddShape(Shape shape)
        {
            _list.Add(shape);
            // Subscribe to the base class event.
            shape.ShapeChanged += HandleShapeChanged;
        }
        // ...Other methods to draw, resize, etc.
        private void HandleShapeChanged(object sender, ShapeEventArgs e)
        {
            if (sender is Shape shape)
            {
                // Diagnostic message for demonstration purposes.
                Console.WriteLine($"Received event. Shape area is now
{e.NewArea}");
                // Redraw the shape here.
                shape.Draw();
            }
        }
    }
    class Test
```

```
    {
        static void Main()
        {
            //Create the event publishers and subscriber
            var circle = new Circle(54);
            var rectangle = new Rectangle(12, 9);
            var container = new ShapeContainer();
            // Add the shapes to the container.
            container.AddShape(circle);
            container.AddShape(rectangle);
            // Cause some events to be raised.
            circle.Update(57);
            rectangle.Update(7, 7);
        }
    }
}
/* Output:
        Received event. Shape area is now 10201.86
        Drawing a circle
        Received event. Shape area is now 49
        Drawing a rectangle
 */
```