## **Events**

When working with delegates in a notification scenario, two distinct roles emerge:

* **Broadcaster (Publisher):** This is the type that *contains* a delegate field. The broadcaster decides *when* to signal an event by invoking its delegate.
* **Subscribers:** These are the method target recipients. A subscriber decides *when to start and stop listening* by adding (+=) or removing (-=) its method from the broadcaster's delegate. Crucially, a subscriber does not directly know about, or interfere with, other subscribers.

**Events** are a language construct that formalizes this pattern, building upon delegates. The primary purpose of events is to **prevent subscribers from interfering with one another**, providing a safer and more controlled mechanism for notifications.

### **Declaring an Event**

The simplest way to declare an event is to place the event keyword in front of a delegate member:

|  |
| --- |
| // 1. Delegate definition (defines the signature of the event handler) public delegate void PriceChangedHandler(decimal oldPrice, decimal newPrice);  public class Broadcaster {  // 2. Event declaration  public event PriceChangedHandler PriceChanged; // 'PriceChanged' is an event } |

**Key Distinction:**

* **Inside the Broadcaster type:** Code within Broadcaster has full access to PriceChanged and can treat it like an ordinary delegate field (e.g., invoke it).
* **Outside the Broadcaster type:** Code outside Broadcaster can **only** perform += (add a subscriber) and -= (remove a subscriber) operations on the PriceChanged event. It **cannot** directly invoke the event, reassign it, or clear all subscribers, thus enforcing the pattern's safety.

## **How Events Work Under the Hood**

When you declare an event using the simple syntax:

|  |
| --- |
| public class Broadcaster {  public event PriceChangedHandler PriceChanged; } |

The C# compiler performs three main actions behind the scenes:

1. **Generates a Private Delegate Field:** The compiler translates the event declaration into a private delegate field:

|  |
| --- |
| private PriceChangedHandler priceChanged; // This is the actual delegate field |

1. **Generates Public Event Accessors:** The compiler generates a public pair of methods, known as **event accessors**, similar to property get and set accessors. These are add\_EventName and remove\_EventName (though you'll see add and remove blocks in explicit declarations):

|  |
| --- |
| public event PriceChangedHandler PriceChanged // This is the 'event' {  add { priceChanged += value; } // 'value' here is the delegate being added  remove { priceChanged -= value; } // 'value' here is the delegate being removed } |

1. **Redirects Operations:**
   * **Inside the Broadcaster class:** Any direct invocations of PriceChanged or assignments to it (e.g., PriceChanged(old, new); or PriceChanged = null;) are redirected by the compiler to operate on the **private priceChanged delegate field**.
   * **Outside the Broadcaster class:** Any += or -= operations on the PriceChanged event are translated into calls to the event's generated add and remove accessors. This is why += and -= behave uniquely for events; they are not simply shortcuts for assignment.

### **Example: Stock Class Firing an Event**

Let's look at a Stock class that fires a PriceChanged event whenever its Price property changes:

|  |
| --- |
| public delegate void PriceChangedHandler(decimal oldPrice, decimal newPrice);  public class Stock {  string symbol;  decimal price;   public Stock(string symbol) => this.symbol = symbol;   public event PriceChangedHandler PriceChanged; // Event declaration   public decimal Price  {  get => price;  set  {  if (price == value) return; // Exit if price hasn't changed   decimal oldPrice = price;  price = value;   // Only invoke if there are subscribers  if (PriceChanged != null) // Check for null (no subscribers)  PriceChanged(oldPrice, price); // Fire the event (invokes the delegate)  }  } } |

If we were to remove the event keyword, PriceChanged would become an ordinary public delegate field. While the example *might* work the same in simple scenarios, it would introduce significant fragility:

* **Subscriber Interference:** Other subscribers could accidentally or maliciously overwrite PriceChanged (stock.PriceChanged = null; or stock.PriceChanged = anotherMethod;), removing all other subscribers.
* **External Invocation:** Outside code could directly invoke stock.PriceChanged(...), causing notifications when the price hasn't actually changed, or manipulating the event flow.

The event keyword prevents these issues by restricting external access to only += and -=.

## **Standard Event Pattern**

For consistency and best practices, events in .NET libraries almost always adhere to a **standard event pattern**. This pattern promotes uniformity and reusability across different events.

The core components of this pattern are:

1. **EventArgs Subclass (for event data):**
   * Inherit from System.EventArgs (a lightweight base class).
   * Named ending in EventArgs (e.g., PriceChangedEventArgs).
   * Contains public, read-only properties or fields to convey information about the event.

|  |
| --- |
| public class PriceChangedEventArgs : System.EventArgs {  public readonly decimal LastPrice;  public readonly decimal NewPrice;   public PriceChangedEventArgs(decimal lastPrice, decimal newPrice)  {  LastPrice = lastPrice;  NewPrice = newPrice;  } } |

1. **Delegate Type (EventHandler<TEventArgs> or custom):**
   * Must have a void return type.
   * Must accept two arguments:
     + object sender: The object that raised the event (the broadcaster).
     + TEventArgs e: An instance of a class derived from EventArgs, containing event data.
   * Conventionally named ending in EventHandler.
2. .NET provides a generic delegate System.EventHandler<TEventArgs> for this:

|  |
| --- |
| public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e) where TEventArgs : EventArgs; |

1. If your event has no data to pass, you can use the non-generic EventHandler delegate, and pass EventArgs.Empty for the e argument.
2. **Event Declaration:**
   * Declare the event using the event keyword with the chosen delegate type.

|  |
| --- |
| public class Stock {  // ...  public event EventHandler<PriceChangedEventArgs> PriceChanged; } |

1. **Protected Virtual "On" Method (Event Raiser):**
   * A protected virtual method that encapsulates the logic for firing the event.
   * Named On followed by the event name (e.g., OnPriceChanged).
   * Accepts a single EventArgs argument.
   * This provides a central, extensible point for subclasses to invoke or override event raising.

|  |
| --- |
| public class Stock {  // ...  public event EventHandler<PriceChangedEventArgs> PriceChanged;   protected virtual void OnPriceChanged(PriceChangedEventArgs e)  {  // Thread-safe invocation: Use null-conditional operator for conciseness  PriceChanged?.Invoke(this, e);  // Equivalent to:  // var temp = PriceChanged;  // if (temp != null) temp(this, e);  } } |

1. The null-conditional operator (?.) is the recommended way to invoke events robustly in multithreaded scenarios, as it prevents a NullReferenceException if the last subscriber unsubscribes just before the event is invoked.

**Complete Example with Standard Pattern:**

|  |
| --- |
| using System;  // 1. EventArgs Subclass public class PriceChangedEventArgs : EventArgs {  public readonly decimal LastPrice;  public readonly decimal NewPrice;  public PriceChangedEventArgs(decimal lastPrice, decimal newPrice)  {  LastPrice = lastPrice; NewPrice = newPrice;  } }  public class Stock {  string symbol;  decimal price;   public Stock(string symbol) => this.symbol = symbol;   // 2. Event Declaration (using generic EventHandler)  public event EventHandler<PriceChangedEventArgs> PriceChanged;   // 3. Protected Virtual "On" Method  protected virtual void OnPriceChanged(PriceChangedEventArgs e)  {  PriceChanged?.Invoke(this, e); // Thread-safe invocation  }   public decimal Price  {  get => price;  set  {  if (price == value) return; // Exit if nothing has changed  decimal oldPrice = price;  price = value;  OnPriceChanged(new PriceChangedEventArgs(oldPrice, price)); // Raise the event  }  } }  // ... in your main code: Stock stock = new Stock("THPW"); stock.Price = 27.10M;  // Register with the PriceChanged event (subscribe) stock.PriceChanged += stock\_PriceChanged;  stock.Price = 31.59M; // This will trigger the event  void stock\_PriceChanged(object sender, PriceChangedEventArgs e) // Subscriber method {  if ((e.NewPrice - e.LastPrice) / e.LastPrice > 0.1M)  Console.WriteLine("Alert, 10% stock price increase!"); } |

## **Event Accessors (Explicit Implementation)**

By default, the compiler implicitly generates the private delegate field and the add/remove accessors for an event. However, you can provide **explicit event accessors** to customize this behavior.

|  |
| --- |
| private EventHandler priceChanged; // Manually declare the private delegate field  public event EventHandler PriceChanged // Explicit accessors {  add { priceChanged += value; } // 'value' is the delegate being added  remove { priceChanged -= value; } // 'value' is the delegate being removed } |

When you explicitly define accessors, the compiler *does not* generate the default field and accessor logic.

Explicit accessors are useful in a few scenarios:

* **Relaying Events:** When your class simply acts as a proxy, forwarding events from another underlying object.
* **Memory Optimization (Sparse Events):** For classes that expose many events but only a few are typically subscribed to (e.g., UI controls), you might store the delegate instances in a Dictionary<string, Delegate> instead of having dozens of null delegate fields. This saves memory if most events have no subscribers.
* **Explicit Interface Implementation of Events:** When a class explicitly implements an event declared in an interface.

|  |
| --- |
| public interface IFoo { event EventHandler Ev; }  class Foo : IFoo {  private EventHandler ev; // Private backing field for the event   event EventHandler IFoo.Ev // Explicit interface implementation  {  add { ev += value; }  remove { ev -= value; }  } } |

* Note that the add and remove parts of an event are compiled into add\_XXX and remove\_XXX methods in the IL.

## **Event Modifiers**

Like methods, events can be decorated with various modifiers:

* virtual: Allows derived classes to override the event.
* override: Overrides a virtual event from a base class.
* abstract: Declares an event that must be implemented by derived non-abstract classes.
* sealed: Prevents a virtual event from being overridden by derived classes.
* static: Declares an event that belongs to the class itself, not an instance.

|  |
| --- |
| public class Foo {  public static event EventHandler StaticEvent; // Static event  public virtual event EventHandler VirtualEvent; // Virtual event  // public abstract event EventHandler AbstractEvent; // In an abstract class } |