## **Events**

When working with delegates in a notification scenario, two distinct roles emerge: Ketika bekerja dengan delegasi dalam skenario notifikasi, ada dua peran yang muncul:

* **Broadcaster (Publisher):** This is the type that *contains* a delegate field. The broadcaster decides *when* to signal an event by invoking its delegate. Penyiar (Penerbit): Ini adalah jenis yang berisi bidang delegasi. Penyiar memutuskan kapan harus memberi sinyal sebuah acara dengan memanggil delegasinya.
* **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. Pelanggan: Ini adalah penerima target metode. Seorang pelanggan memutuskan kapan mulai dan berhenti mendengarkan dengan menambahkan (+=) atau menghapus (-=) metodenya dari delegasi penyiar. Yang terpenting, seorang pelanggan tidak secara langsung mengetahui, atau mengganggu, pelanggan lain.

**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. Peristiwa adalah konstruksi bahasa yang memformalkan pola ini, yang dibangun di atas delegasi. Tujuan utama event adalah untuk mencegah pelanggan saling mengganggu satu sama lain, menyediakan mekanisme yang lebih aman dan terkendali untuk pemberitahuan.

### **Declaring an Event**

The simplest way to declare an event is to place the event keyword in front of a delegate member: Cara paling sederhana untuk mendeklarasikan sebuah event adalah dengan menempatkan kata kunci event di depan anggota delegasi:

|  |
| --- |
| // 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). Di dalam tipe Penyiar: Kode di dalam Broadcaster memiliki akses penuh ke PriceChanged dan dapat memperlakukannya seperti bidang delegasi biasa (misalnya, memanggilnya).
* **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. Di luar tipe Penyiar: Kode di luar Broadcaster hanya dapat melakukan operasi += (menambah pelanggan) dan -= (menghapus pelanggan) pada peristiwa PriceChanged. Kode tersebut tidak dapat secara langsung memanggil event, menetapkan ulang, atau menghapus semua pelanggan, sehingga menegakkan keamanan pola.

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

When you declare an event using the simple syntax: Ketika Anda mendeklarasikan sebuah event dengan menggunakan sintaks sederhana:

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

The C# compiler performs three main actions behind the scenes: Kompiler C# melakukan tiga tindakan utama di belakang layar:

1. **Generates a Private Delegate Field:** The compiler translates the event declaration into a private delegate field: Menghasilkan Bidang Delegasi Pribadi: Kompiler menerjemahkan deklarasi event ke dalam bidang delegasi privat:

|  |
| --- |
| 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): Menghasilkan Pengakses Peristiwa Publik: Kompiler menghasilkan sepasang metode publik, yang dikenal sebagai pengakses event, mirip dengan pengakses properti get dan set. Ini adalah add\_EventName dan remove\_EventName (meskipun Anda akan melihat blok tambah dan hapus dalam deklarasi eksplisit):

|  |
| --- |
| 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**. Di dalam kelas Penyiar: Setiap pemanggilan langsung dari PriceChanged atau penugasan padanya (misalnya, PriceChanged(old, new); atau PriceChanged = null;) akan diarahkan oleh kompiler untuk beroperasi pada bidang delegasi private priceChanged.
   * **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. Di luar kelas Penyiar: Setiap operasi += atau -= pada event PriceChanged diterjemahkan ke dalam pemanggilan ke pengakses tambah dan pengakses hapus yang dibuat oleh event. Inilah sebabnya mengapa += dan -= berperilaku unik untuk event; mereka bukan hanya jalan pintas untuk penugasan.

### **Example: Stock Class Firing an Event** Contoh: Kelas Saham Menembakkan Peristiwa

Let's look at a Stock class that fires a PriceChanged event whenever its Price property changes: Mari kita lihat kelas Saham yang menembakkan event PriceChanged setiap kali properti Price-nya berubah:

|  |
| --- |
| 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: Jika kita menghapus kata kunci event, PriceChanged akan menjadi bidang delegasi publik biasa. Meskipun contoh ini mungkin bekerja sama dalam skenario sederhana, namun akan menimbulkan kerapuhan yang signifikan:

* **Subscriber Interference:** Other subscribers could accidentally or maliciously overwrite PriceChanged (stock.PriceChanged = null; or stock.PriceChanged = anotherMethod;), removing all other subscribers. Gangguan Pelanggan: Pelanggan lain dapat secara tidak sengaja atau jahat menimpa PriceChanged (stock.PriceChanged = null; atau stock.PriceChanged = anotherMethod;), menghapus semua pelanggan lain.
* **External Invocation:** Outside code could directly invoke stock.PriceChanged(...), causing notifications when the price hasn't actually changed, or manipulating the event flow. Pemanggilan Eksternal: Kode dari luar dapat secara langsung memanggil stock.PriceChanged(...), menyebabkan notifikasi ketika harga tidak benar-benar berubah, atau memanipulasi alur kejadian.

The event keyword prevents these issues by restricting external access to only += and -=. Kata kunci event mencegah masalah ini dengan membatasi akses eksternal hanya pada += dan -=.

## **Standard Event Pattern** **Pola Acara Standar**

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. Untuk konsistensi dan praktik terbaik, event di perpustakaan .NET hampir selalu mengikuti pola event standar. Pola ini mendorong keseragaman dan penggunaan ulang di berbagai event yang berbeda.

The core components of this pattern are: Komponen inti dari pola ini adalah:

1. **EventArgs Subclass (for event data):** **Subkelas EventArgs (untuk data peristiwa):**
   * Inherit from System.EventArgs (a lightweight base class). Mewarisi dari System.EventArgs (kelas dasar yang ringan).
   * Named ending in EventArgs (e.g., PriceChangedEventArgs).Nama yang diakhiri dengan EventArgs (misal, PriceChangedEventArgs).
   * Contains public, read-only properties or fields to convey information about the event. Berisi properti atau bidang publik dan hanya-baca untuk menyampaikan informasi tentang acara tersebut.

|  |
| --- |
| 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):** **2**. Tipe Delegasi (EventHandler<TEventArgs> atau custom):  
   * Must have a void return type. Harus memiliki tipe pengembalian void.
   * Must accept two arguments: Harus menerima dua argumen:
     + object sender: The object that raised the event (the broadcaster). objek pengirim: Objek yang memunculkan acara (penyiar).
     + TEventArgs e: An instance of a class derived from EventArgs, containing event data. TEventArgs e: Instance dari kelas yang diturunkan dari EventArgs, yang berisi data peristiwa.
   * Conventionally named ending in EventHandler. Nama konvensional yang diakhiri dengan EventHandler.
2. .NET provides a generic delegate System.EventHandler<TEventArgs> for this: 3. .NET menyediakan delegasi generik System.EventHandler<TEventArgs> untuk ini:

|  |
| --- |
| 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. 4. Jika event Anda tidak memiliki data untuk dilewatkan, Anda bisa menggunakan delegasi EventHandler non-generik, dan melewatkan EventArgs.Empty untuk argumen e.
2. **Event Declaration:**
   * Declare the event using the event keyword with the chosen delegate type. Deklarasikan event menggunakan kata kunci event dengan jenis delegasi yang dipilih.

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

1. **Protected Virtual "On" Method (Event Raiser):** **6. Metode “On” Virtual Terproteksi (Pembangkit Peristiwa):**
   * A protected virtual method that encapsulates the logic for firing the event. Metode virtual terproteksi yang merangkum logika untuk menjalankan event.
   * Named On followed by the event name (e.g., OnPriceChanged). Bernama On diikuti dengan nama acara (mis., OnPriceChanged).
   * Accepts a single EventArgs argument. Menerima argumen EventArgs tunggal.
   * This provides a central, extensible point for subclasses to invoke or override event raising. Ini menyediakan titik pusat yang dapat diperluas bagi subkelas untuk memanggil atau menimpa pemunculan event.

|  |
| --- |
| 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. Operator null-conditional (?.) adalah cara yang direkomendasikan untuk memanggil event secara kuat dalam skenario multithread, karena operator ini mencegah NullReferenceException jika pelanggan terakhir berhenti berlangganan sebelum event dipanggil.

**Complete Example with Standard Pattern:** **Contoh Lengkap dengan Pola Standar:**

|  |
| --- |
| 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)** **Pengakses Peristiwa (Implementasi Eksplisit)**

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. Secara default, kompiler secara implisit membuat bidang delegasi pribadi dan pengakses tambah/hapus untuk sebuah event. Namun, Anda dapat menyediakan pengakses event secara eksplisit untuk menyesuaikan perilaku ini.

|  |
| --- |
| 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. Ketika Anda mendefinisikan pengakses secara eksplisit, kompiler tidak akan menghasilkan field dan logika pengakses default.

Explicit accessors are useful in a few scenarios: Pengakses eksplisit berguna dalam beberapa skenario:

* **Relaying Events:** When your class simply acts as a proxy, forwarding events from another underlying object. Meneruskan Peristiwa: Ketika kelas Anda hanya bertindak sebagai proxy, meneruskan kejadian dari objek lain yang mendasarinya.
* **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. Pengoptimalan Memori (Kejadian Jarang): Untuk kelas yang mengekspos banyak kejadian namun biasanya hanya sedikit yang menjadi pelanggan (misalnya, kontrol UI), Anda dapat menyimpan contoh delegasi dalam Dictionary<string, Delegate> alih-alih memiliki lusinan bidang delegasi nol. Hal ini akan menghemat memori jika sebagian besar event tidak memiliki pelanggan.
* **Explicit Interface Implementation of Events:** When a class explicitly implements an event declared in an interface.mplementasi Antarmuka Eksplisit dari Event: Ketika sebuah kelas secara eksplisit mengimplementasikan event yang dideklarasikan dalam antarmuka.

|  |
| --- |
| 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. Perhatikan bahwa bagian tambah dan hapus dari sebuah event dikompilasi ke dalam metode add\_XXX dan remove\_XXX di dalam IL.

## **Event Modifiers**

Like methods, events can be decorated with various modifiers: Seperti metode, acara dapat didekorasi dengan berbagai pengubah:

* virtual: Allows derived classes to override the event. Mengizinkan kelas turunan untuk menimpa event.
* override: Overrides a virtual event from a base class. Menimpa acara virtual dari kelas dasar.
* abstract: Declares an event that must be implemented by derived non-abstract classes. Mendeklarasikan sebuah kejadian yang harus diimplementasikan oleh kelas non-abstrak turunan.
* sealed: Prevents a virtual event from being overridden by derived classes. Mencegah kejadian virtual ditimpa oleh kelas turunan.
* static: Declares an event that belongs to the class itself, not an instance. Mendeklarasikan sebuah kejadian yang merupakan bagian dari kelas itu sendiri, bukan sebuah 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 } |