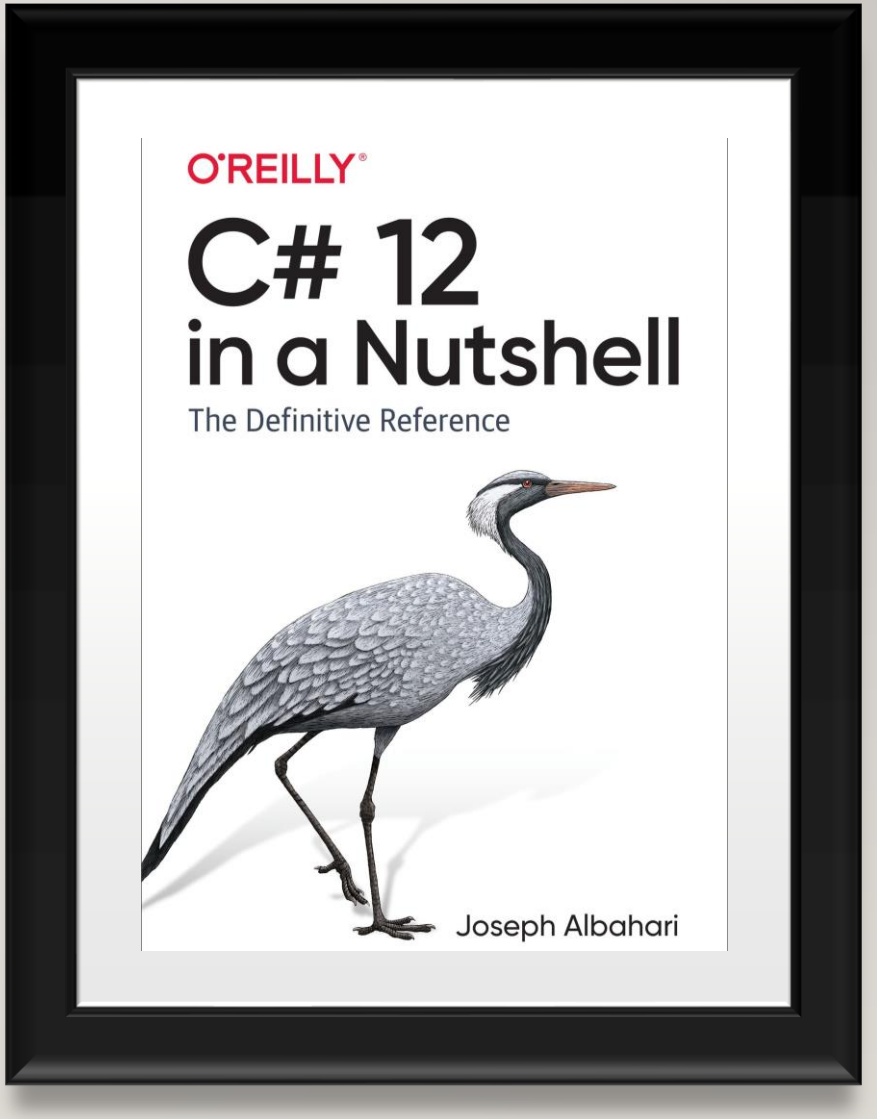
CHAPTER 4 - EVENTS

- broadcaster and subscriber
- Events
- Standard Event Pattern
- Event Accessors
- Event Modifiers



broadcaster and subscriber

وقتی از delegate ها استفاده میکنیم معمولا دو نقش مهم وارد بازی میشند: broadcaster و subscriber

Broadcaster یک نوعی است که شامل یک فیلد delegate هست، broadcaster تصمیم میگیره که چه زمانی با فراخوانی اون delegate کارشو انجام بده.

Subscriberها متدهایی هستند که به عنوان target-method به نمونه ای که از delegate ساختیم معرفی شدند. subscriber با کمک =+ و =- در broadcast-delegate تصمیم میگیره که چه زمانی listening رو شروع یا متوقف کنه. همچنین یه subscriber در مورد بقیه subscriber ها چیزی نمیدونه یا اگر هم بدونه در کارشون دخالتی نمیکنه.

ازونجایی که مطمئن نیستم منظور رو درست تونسته باشم منتقل کنم در اسلاید بعدی متن اصلی رو هم میارم.

broadcaster and subscriber

When using delegates, two emergent roles commonly appear: broadcaster and subscriber.

The broadcaster is a type that contains a delegate field. The broadcaster decides when to broadcast, by invoking the delegate.

The subscribers are the method target recipients. A subscriber decides when to start and stop listening by calling += and -= on the broadcaster’s delegate. A subscriber does not know about, or interfere with, other subscribers.

Events

event ها یک ویژگی زبان سی شارپ هستند که الگوی رویداد محور رو formalize میکنند.

یک event ساختاری هستش که مجموعه ای از delegate های مورد نیاز برای مدل broadcaster/subscriber را نمایش میدهد.

هدف اصلی eventها اینه که نذارن یه subscriber روی کار اون یکی تاثیر بذاره و یا دخالتی کنه.

متن اصلی رو تو اسلاید بعد میتونید ببینید.

Events

Events are a language feature that formalizes this pattern.

An event is a construct that exposes just the subset of delegate features required for the broadcaster/subscriber model.

The main purpose of events is to prevent subscribers from interfering with one another.

Events

برای تعریف یک event کافیه فقط کلمه کلیدی event رو جلوی فیلد delegate قرار بدیم:

```
public delegate void PriceChangedHandler(decimal oldPrice, decimal newPrice);
```

```
0 references | 0 changes | 0 authors, 0 changes
internal class SampleBroadcaster
{
    public event PriceChangedHandler PriceChanged;
}
```

کدهای داخلی broadcaster به priceChanged دسترسی کامل دارند و میتونند از اون به عنوان یه delegate استفاده کنند. کدهایی که قراره از این کلاس استفاده کنند فقط میتونند از عملگرهای =+ و =- در مواجهه با priceChanged استفاده کنند.

Events
HOW DO EVENTS WORK ON THE INSIDE?

```
0 references | 0 changes | 0 authors, 0 changes
internal class SampleBroadcaster
{
    public event PriceChangedHandler PriceChanged;
}
```

```
PriceChangedHandler priceChanged;   // private delegate
public event PriceChangedHandler PriceChanged
{
    add { priceChanged += value; }
    remove { priceChanged -= value; }
}
```

وقتی ما یک ایونت تعریف میکنیم سه اتفاق میفته:

I. کامپایلر کد رو به یه همچین کدی تبدیل میکنه:

II. کامپایلر توی کلاس broadcaster میگرده تا ببینه کجاها این ایونت رفرنس داره و عملیاتی غیر از =+ یا =– انجام میدند و اونا رو به priceChange ریدایرکت میکنه.

III. کامپایلر عملگرهای =+ و =– رو به add و remove ترجمه میکنه. جالبه که این باعث میشه عملکرد این دو عملگر منحصر به فرد بشه برای ایونت ها و برخلاف بقیه موارد معمول این عملگرها ساده شده + و – نیستند که در assigment ازشون استفاده میکنیم.

Events
HOW DO EVENTS WORK ON THE INSIDE?

اینم متن اصلی، چون زیاد بود عکس گذاشتم:

Three things happen under the hood when you declare an event as follows:

```
public class Broadcaster
{
  public event PriceChangedHandler PriceChanged;
}
```

First, the compiler translates the event declaration into something close to the following:

```
PriceChangedHandler priceChanged;    // private delegate
public event PriceChangedHandler PriceChanged
{
  add     { priceChanged += value; }
  remove  { priceChanged -= value; }
}
```

The `add` and `remove` keywords denote explicit *event accessors*—which act rather like property accessors. We describe how to write these later.

Second, the compiler looks *within* the `Broadcaster` class for references to `PriceChanged` that perform operations other than `+=` or `-=` and redirects them to the underlying `priceChanged` delegate field.

Third, the compiler translates `+=` and `-=` operations on the event to calls to the event's `add` and `remove` accessors. Interestingly, this makes the behavior of `+=` and `-=` unique when applied to events: unlike in other scenarios, it's not simply a shortcut for `+` and `-` followed by an assignment.

Events

```
2 references | 0 changes | 0 authors, 0 changes
public class Stock
{
    string symbol;
    decimal price;

    1 reference | 0 changes | 0 authors, 0 changes
    public Stock(string symbol) => this.symbol = symbol;

    public event PriceChangedHandler PriceChanged;

    1 reference | 0 changes | 0 authors, 0 changes
    public decimal Price
    {
        get => price;
        set
        {
            if (price == value) return;      // Exit if nothing has changed
            decimal oldPrice = price;
            price = value;
            if (PriceChanged != null)           // If invocation list not
                PriceChanged(oldPrice, price);   // empty, fire event.
        }
    }
}
```

این مثال رو ببینید:

اگه کلمه کلیدی event رو حذف کنیم فیلدمون تبدیل به یه فیلد معمولی میشه ولی همچنان این مثال همون نتیجه قبل رو میده. و البته subscriberها میتونند با روشهای زیر تو کار هم دخالت کنند:

1. مقداردهی مجدد subscriber (به جای =+)
2. پاک کردن همه subscriberها
3. فراخوانی بقیه subscriberها به صورت دستی

Events

حالا بیاییم برگردونیم فیلد رو به ایونت. همونطور که میبینید اجازه بهمون نمیده تا اون سه تا کاری که در اسلاید قبل اومد رو انجام بدیم و خیالمون از بابت دخالت subscriber ها تو کار همدیگه راحته.

به عبارتی کلا برای ایونت ها دو عملگر =+ و =- تعریف شده.

```
var stockPC = new Stock("PC");

stockPC.PriceChanged += (s, e) =>
{
    Console.WriteLine($"PriceChanged...");
};

        stockPC.PriceChanged = (s, e) => { Console.WriteLine("It's me..."); }; // cannot assign
        stockPC.PriceChanged = null; // cannot set null;
        stockPC.PriceChanged(100, 200); // cannot invoke

stockPC.Price = 100;
```

PriceChangedHandler Stock.PriceChanged

CS0070: The event 'Stock.PriceChanged' can only appear on the left hand side of += or -= (except when used from within the type 'Stock')

Standard Event Pattern

برای استفاده از ایونت ها در دات نت یک الگوی استاندارد وجود داره که بهتره از اون استفاده کرد. هسته اصلی این الگو System.EventArgs (یک کلاس از پیش تعریف شده در دات نت که به جز فیلد استاتیک Empty هیچ عضوی نداره) هستش.

```
public class EventArgs
{
    public static readonly EventArgs Empty = new EventArgs();

    public EventArgs()
    {
    }
}
```

EventArgs یک کلاس base هست که برای انتقال داده به ایونت ها ازش استفاده میشه.

Standard Event Pattern

حالا بیاییم تو مثال قبلی یه subclass از EventArgs بسازیم که بتونه دیتای مورد نیازمون رو وقتی ایونت priceChanged فراخوانی میشه به اون بفرسته.

معمولا برای اینکه بعدا هرجا نیاز داشتیم از این subclass استفاده کنیم اسم اونو مطابق با دیتایی که قراره منتقل کنه در نظر میگیریم.

همچنین این کلاس معمولا دیتا رو به صورت پراپرتی یا فیلد فقط خواندنی نگهداری میکنه.

```
1 reference | 0 changes | 0 authors, 0 changes
public class PriceChangedEventArgs : System.EventArgs
{
    public readonly decimal LastPrice;
    public readonly decimal NewPrice;

    0 references | 0 changes | 0 authors, 0 changes
    public PriceChangedEventArgs(decimal lastPrice, decimal newPrice)
    {
        LastPrice = lastPrice;
        NewPrice = newPrice;
    }
}
```

Standard Event Pattern

بعد از تعریف subclass از EventArgs قدم بعدی انتخاب یا نوشتن یه delegate برای اون هستش که باید این سه قانون رو رعایت کنیم:

I. متد مدنظر **نباید** خروجی داشته باشه. (void باشه)

II. حتما دوتا پارامتر ورودی داشته باشه، اولی از نوع object و دومی از نوع کلاس مشتق شده از EventArgs. پارامتر اول broadcaster رو مشخص میکنه و پارامتر دوم شامل اطلاعاتی است که قراره ارسال بشه.

III. اسم متد حتما به EventHandler ختم بشه.

* دات نت خودش برای کمک به این موضوع یه generic delegate به اسم System.EventHandler<> داره.

```
public delegate void EventHandler<TEventArgs>(object? sender, TEventArgs e);
```

Standard Event Pattern

قبل اینکه جنریک ها معرفی بشن (قبل از C# 2.0) ما باید delegate ها رو خودمون سفارشی مینوشتیم مثل این:

```
public delegate void PriceChangedHandler(object sender, PriceChangedEventArgs e);
```

به همین علت هنوز خیلی از ایونت هایی که توی دات نت هست از این روش برای پیاده سازیشون استفاده شده.

Standard Event Pattern

قدم بعدی اینه که یه ایونت از نوع delegate ی که در اسلاید قبل ساختیم بسازیم(اینجا ما از جنریک EventHandler استفاده کردیم).

```
//public event PriceChangedHandler PriceChanged;  // old way without generic delegate
public event EventHandler<PriceChangedEventArgs> PriceChanged;
0 references | 0 changes | 0 authors, 0 changes
protected virtual void OnPriceChanged(PriceChangedEventArgs e)
{
    if (PriceChanged != null) PriceChanged(this, e);
}
```

و در نهایت یه متد protected virtual میخواییم که ایونت رو صدا بزنه.

این متد حتما باید اسمش همون اسم ایونت باشه با یک پیشوند On و یک پارامتر از جنس EventArgs هم نیاز داریم.

Standard Event Pattern

برای اینکه در سناریوهای multi-thread عملکرد بهتری داشته باشیم بهتره که یه متغیر موقتی برای بررسی و فراخوانی delegate داشته باشیم.

```
//way 1
var temp = PriceChanged;
if (temp != null) temp(this, e);

//way 2
PriceChanged?.Invoke(this, e);
```

البته با استفاده از عملگر ? هم میشه به این هدف رسید.

هر دوی این روشها thread-safe هستند و بهترین روش برای فراخوانی ایونت ها است.

Standard Event Pattern

حالا ما یه ساختار مرکزی داریم که به راحتی کلاسهای مشتق شده از stock میتونند ایونت رو فراخوانی کنند یا اونو مجدد override کنند.

تنها چیزی که نیازه این خط هست که باید به setter فیلد price اضافه کنیم تا به درستی کارشو انجام بده:

```
OnPriceChanged(new PriceChangedEventArgs(oldPrice, price));
```

Event Accessors

Event Accessor ها پیاده سازی برای عملگرهای =+ و =- هستش. به صورت پیش فرض این accessorها توسط خود کامپایلر ایجاد میشه.

```
public event EventHandler PriceChanged;
```

وقتی ما یه همچین ایونتی تعریف میکنیم کامپایلر اونو تبدیل میکنه به موارد زیر:

- یک فیلد private از جنس همون delegate
- دوتا متد add_PriceChanged و remove_PriceChanged که پیاده سازی های =+ و =- رو به فیلد private که ایجاد کرده ارسال کند.

```
private EventHandler priceChanged;          // Declare a private delegate
public event EventHandler PriceChanged
{
    add { priceChanged += value; }
    remove { priceChanged -= value; }
}
```

اینکار رو به صورت دستی هم میتونید خودتون انجام بدید:

Event Accessors

با تعریف دستی accessor برای ایونت ها میشه سناریوهای پیچیده تری رو مدیریت کرد. سه تا سناریو تو کتاب اومده برای نمونه که عین متن رو میارم:

- When the event accessors are merely relays for another class that is broadcasting the event.
- When the class exposes many events, for which most of the time very few subscribers exist, such as a Windows control. In such cases, it is better to store the subscriber's delegate instances in a dictionary because a dictionary will contain less storage overhead than dozens of null delegate field references.
- When explicitly implementing an interface that declares an event.

Event Modifiers

مثل متدها، ایونت ها هم میتونند virtual، overridden، abstract و یا sealed باشند.حتی static هم میتونند باشند:

```
0 references | 0 changes | 0 authors, 0 changes
public class Foo
{
    public static event EventHandler<EventArgs> StaticEvent;
    public virtual event EventHandler<EventArgs> VirtualEvent;
}
```