43. What are events?

Brief summary: Events are the .NET way of implementing the Observer design pattern. They are used to send a notification from an object to all objects subscribed.

An event is a message sent by an object to signal the occurrence of an action. Events are the .NET implementation of the Observer design pattern.

Let's implement the same logic we had in the lecture about the Observer design pattern. We want the BitcoinPriceReader object to notify other objects about the reading of the Bitcoin price.

This time, we will do so using events. We will start by defining a delegate that will represent the function or functions that will be executed once the price reading event has been raised:

```
public delegate void PriceRead(decimal price);
```

The BitcoinPriceReader will own an **event** of this delegate's type:

```
public class BitcoinPriceReader
{
    private int _currentBitcoinPrice;
    public event PriceRead? PriceRead;
```

Let's take a moment to stop and think about what happened here. We declared an **event belonging** to the BitcoinPriceReader class. An event is always of a delegate type. Remember, a delegate is a type whose instances hold a reference to a method with a particular parameter list and return type.

So what is the difference between an event, and a regular field of delegate type? In other words, what will be the difference between those two?

```
public class BitcoinPriceReader
{
    private int _currentBitcoinPrice;

    public event PriceRead? PriceRead;
    public PriceRead? PriceReadNotAnEvent;
```

I will try to attach some method to both of them.

```
void SomeMethod(decimal price)
{
    Console.WriteLine($"Price is {price}");
}
```

```
var bitcoinPriceReader = new BitcoinPriceReader();
bitcoinPriceReader.PriceRead += SomeMethod;
bitcoinPriceReader.PriceReadNotAnEvent += SomeMethod;
```

So far, an event and a field of delegate type act exactly the same. But here is the difference:

```
bitcoinPriceReader.PriceRead(100);
bitcoinPriceReader.PriceReadNotAnEvent(100);
```

As you can see, I can't invoke the event from outside the class it belongs to. I can invoke the non-event delegate without a problem.

This is a **critical difference**. Only the class that owns an event can raise it. Events are used to send notifications about some action, so imagine what would happen if any class could raise them: any code could raise the PriceRead event with any price they want, triggering invalid notifications all around the system. The event delegate **must be public** so the subscribers can subscribe to it, but it must only be invokable from within the class that owns it. And this is what the "event" keyword enforces.

All right. Let's move on with the implementation. We defined an event in the BitcoinPriceReader class.

```
public event PriceRead? PriceRead;
```

As you can see I declared it as nullable because before any subscriber subscribes to it, it will be null.

All right. When the Bitcoin price is read, we want to "raise" the event, so simply invoke all methods stored in the event delegate. That means, all subscribers will be notified that the event occurred:

```
public class BitcoinPriceReader
{
    private int _currentBitcoinPrice;

    public event PriceRead? PriceRead;
    public PriceRead? PriceReadNonEvent;

    1 reference
    public void ReadCurrentPrice()
    {
        _currentBitcoinPrice = new Random().Next(0, 50000);
        OnPriceRead(_currentBitcoinPrice);
    }

    1 reference
    protected virtual void OnPriceRead(decimal price)
    {
        PriceRead?.Invoke(price);
    }
}
```

As you can see I execute the event delegate by the **Invoke** method. This is because the "normal" execution does not allow using the null-conditional operator, and as we said, the PriceRead event might be null:

```
1 reference
protected virtual void OnPriceRead(decimal price)
{
    PriceRead?(price);
}
```

As you can see, the above code doesn't compile. I don't want to simply call "PriceRead(price)" because I would be at risk of causing the NullReferenceException.

All right. Let's summarize what happened in this class. We declared an event, which is of a delegate type. When the Bitcoin price is read, we want to raise the event so all subscribers are notified.

Let's move on to the subscribers. Each subscriber must contain a method that is compatible with the event delegate (in our case, a void method accepting a decimal).

```
public delegate void PriceRead(decimal price);
```

Let's see such a method in the PushPriceChangeNotifier class:

```
public class PushPriceChangeNotifier
{
    private readonly decimal _notificationThreshold;

    1 reference
    public PushPriceChangeNotifier(decimal notificationThreshold)
    {
        _notificationThreshold = notificationThreshold;
    }

    0 references
    public void Update(decimal price)
    {
        if (price > _notificationThreshold)
        {
            Console.WriteLine($"Sending a push notification saying that " +
                $"the Bitcoin price exceeded {_notificationThreshold} " +
                $"and is now {price}\n");
        }
    }
}
```

A very similar method exists in EmailPriceChangeNotifier class. We can now subscribe those two classes to be notified when the event is raised:

```
var emailPriceChangeNotifier = new EmailPriceChangeNotifier(25000);
var pushPriceChangeNotifier = new PushPriceChangeNotifier(40000);

bitcoinPriceReader.PriceRead += emailPriceChangeNotifier.Update;
bitcoinPriceReader.PriceRead += pushPriceChangeNotifier.Update;
```

And that's it! All that's left is to call the ReadCurrentPrice method:

```
bitcoinPriceReader.ReadCurrentPrice();
```

And the result is:

```
Sending an email saying that the Bitcoin price exceeded 25000
and is now 49741

Sending a push notification saying that the Bitcoin price exce
eded 40000 and is now 49741
```

It seems like everything is working. Let's see again what happened. When the ReadCurrentPrice method is executed, it raises the PriceRead event.

```
public void ReadCurrentPrice()
{
    _currentBitcoinPrice = new Random().Next(0, 50000);
    OnPriceRead(_currentBitcoinPrice);
}

1 reference
protected virtual void OnPriceRead(decimal price)
{
    PriceRead?.Invoke(price);
}
```

The PriceRead event is invoked, and since the Update methods from EmailPriceChangeNotifier and PushPriceChangeNotifier are attached to the event delegate, they get executed.

All right. This works as expected. There is one improvement we can make, though. Instead of using our own PriceRead delegate for the event, we can use the EventHandler delegate that is predefined in C#:

```
public event EventHandler PriceRead;
```

This is the signature of this delegate:

```
public delegate void EventHandler(object? sender, EventArgs e);
```

As you can see it carries the information about the object that raised the event (sender) and event arguments. EventArgs is a base class for any event arguments we want. In our case, the argument of an event is the Bitcoin price that has been read. Let's create our own type derived from EventArgs:

```
public class PriceReadEventArgs : EventArgs
{
    1 reference
    public decimal Price { get; }

    0 references
    public PriceReadEventArgs(decimal price)
    {
        Price = price;
    }
}
```

To make sure this type of event argument will be used, we must use the generic EventHandler:

```
public event EventHandler<PriceReadEventArgs> PriceRead;
```

We must now change the code that raises the event, to match the EventHandler<PriceReadEventArgs> delegate type:

```
protected virtual void OnPriceRead(decimal price)
{
    PriceRead?.Invoke(this, new PriceReadEventArgs(price));
}
```

As you can see, as the first argument we pass “this” so the sender of the event. The second argument is the PriceReadEventArgs object holding the price.

The last thing left to do is to change the Update method in EmailPriceChangeNotifier and PushPriceChangeNotifier classes:

```
public void Update(object sender, PriceReadEventArgs eventArgs)
{
    if (eventArgs.Price > _notificationThreshold)
    {
        Console.WriteLine($"Sending an email saying that " +
            $"the Bitcoin price exceeded {_notificationThreshold} " +
            $"and is now {eventArgs.Price}\n");
    }
}
```

All right. Now everything works as expected.

You may think that this is more complicated than the code that we had before, and this is true. Nevertheless, I wanted to show you this EventHandler type, as it is used in many Microsoft’s frameworks based on events. For example, in Windows Forms or Windows Presentation Foundation desktop applications. All user actions like clicking a button or closing a window trigger events and those events are defined with the EventHandler delegate. In those use cases, the sender argument is used by the subscribers, which we did not need to do in our code.

One last thing before we wrap up. This is generally a good practice to unsubscribe from the event handler before the object getting notified about the event is discarded:

```
bitcoinPriceReader.PriceRead -= emailPriceChangeNotifier.Update;
bitcoinPriceReader.PriceRead -= pushPriceChangeNotifier.Update;
```

I don’t want to get into details about why is it important (as this lecture is quite long already), so let me just give you a very quick overview. When a subscriber subscribes to be notified about a state change in some object, a **hidden reference** is created between them. It is needed because the EventHandler holds a reference to a method stored in the object that will be notified about the event, so a reference to this object is necessary. It may happen that we think an object is no longer in use, while the Garbage Collector still sees a reference to it, and will not remove it from memory. For example, consider a desktop application with some MainWindow, and a ChildWindow that opens when we click some button. The ChildWindow is subscribed to an event of the MainWindow, so the reference from the MainWindow to ChildWindow exists. When we close the ChildWindow it should be removed from memory, but the Garbage Collector will see this reference and will decide this object should not be removed. As the application’s user keeps opening and closing ChildWindows, more and more memory is being used, but none of it is being freed. This situation is called a “memory leak”. Over time, it may slow the application down, or even cause it to crash due to OutOfMemoryException. Unsubscribing from events when it’s possible (for example, when the ChildWindow is being closed) is a way of preventing that.

Let’s summarize. Events are .NET way of implementing the Observer design pattern. They are used to send a notification from an object to all objects subscribed. The pattern of using events is as follows:

- the class that will be sending notifications owns an event, which is a delegate
- objects that want to be notified about an event can attach their own methods to this delegate
- when the observable class raises the event, it does so by invoking the methods stored in the delegate. This way, all methods from the observers will be executed

Bonus questions:

- "**What is the difference between an event and a field of the delegate type?**"
  A public field of a delegate type can be invoked from anywhere in the code. Events can only be invoked from the class they belong to.

- "**Why is it a good practice to unsubscribe from events when a subscribed object is no longer needed?**"
  Because as long as it is subscribed, a hidden reference between the observable and the observer exists, and it will prevent the Garbage Collector from removing the observer object from memory.