https://github.com/head-first-csharp/fifth-edition

O'REILLY®

Check out our GitHub page for videos, downloads, and more!

Fifth Edition

Also covers .NET MAUI & Unity

Head First C#

A Learner's Guide to Real-World Programming with C# and .NET

Andrew Stellman
& Jennifer Greene

Events and Delegates

This is a bonus downloadable chapter.

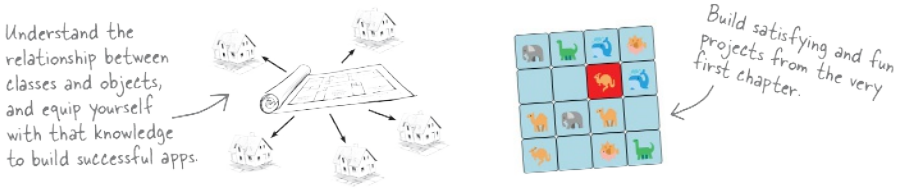
A Brain-Friendly Guide

Head First

C#

What will you learn from this book?

Create apps, games, and more using this engaging, highly visual introduction to C#, .NET, and software development. You'll learn how to use classes and object-oriented programming, create 3D games in Unity, and query data with LINQ. And you'll do it all by solving puzzles, completing hands-on exercises, and building real-world applications. Interested in a development career? You'll learn important development techniques and ideas—just like many others who've learned to code with this book and are now professional developers, team leads, coding streamers, and more. There's no experience required except the desire to learn. And this is the best place to start.



What's so special about this book?

If you've read a Head First book, you know what to expect: a visually rich format designed for the way your brain works. If you haven't, you're in for a treat. With this book, you'll learn C# through a multisensory experience that engages your mind—rather than a text-heavy approach that puts you to sleep.

"Thank you so much! Your books have helped me to launch my career."

–Ryan White
Game Developer

"In a sea of dry technical manuals, *Head First C#* stands out as a beacon of brilliance. Its unique teaching style not only imparts essential knowledge but also sparks curiosity and fuels passion for coding. An indispensable resource for beginners!"

–Gerald Versluis
Senior Software Engineer at Microsoft

"Andrew and Jennifer have written a concise, authoritative, and, most of all, fun introduction to C# development."

–Jon Galloway
Senior Program Manager on the .NET Community Team at Microsoft

C# / .NET

US $79.99 **CAN $99.99**

ISBN: 978-1-098-14178-3

O'REILLY®

Praise for Head First C#

"In a sea of dry technical manuals, *Head First C#* stands out as a beacon of brilliance. Its unique teaching style not only imparts essential knowledge but also sparks curiosity and fuels passion for coding. An indispensable resource for beginners!"

—Gerald Versluis, Senior Software Engineer at Microsoft

"*Head First C#* started my career as a software engineer and backend developer. I am now leading a team in a tech company and an open source contributor."

—Zakaria Soleymani, Development Team Lead

"Thank you so much! Your books have helped me to launch my career."

—Ryan White, Game Developer

"If you're a new C# developer (welcome to the party!), I highly recommend *Head First C#*. Andrew and Jennifer have written a concise, authoritative, and most of all, fun introduction to C# development. I wish I'd had this book when I was first learning C#!"

—Jon Galloway, Senior Program Manager on the .NET Community Team, Microsoft

"Not only does *Head First C#* cover all the nuances it took me a long time to understand, it has that Head First magic going on where it is just a super fun read."

—Jeff Counts, Senior C# Developer

"*Head First C#* is a great book with fun examples that keep learning interesting."

—Lindsey Bieda, Lead Software Engineer

"*Head First C#* is a great book, both for brand-new developers and developers like myself coming from a Java background. No assumptions are made as to the reader's proficiency, yet the material builds up quickly enough for those who are not complete newbies—a hard balance to strike. This book got me up to speed in no time for my first large-scale C# development project at work—I highly recommend it."

—Shalewa Odusanya, Principal

"*Head First C#* is an excellent, simple, and fun way of learning C#. It's the best piece for C# beginners I've ever seen—the samples are clear, the topics are concise and well written. The mini-games that guide you through the different programming challenges will definitely stick the knowledge to your brain. A great learn-by-doing book!"

—Johnny Halife, Partner

"*Head First C#* is a comprehensive guide to learning C# that reads like a conversation with a friend. The many coding challenges keep it fun, even when the concepts are tough."

—Rebeca Dunn-Krahn, Founding Partner, Sempahore Solutions

Praise for Head First C#

"I've never read a computer book cover to cover, but this one held my interest from the first page to the last. If you want to learn C# in depth and have fun doing it, this is THE book for you."

—Andy Parker, fledgling C# Programmer

"It's hard to really learn a programming language without good, engaging examples, and this book is full of them! *Head First C#* will guide beginners of all sorts to a long and productive relationship with C# and the .NET Framework."

—Chris Burrows, Software Engineer

"With *Head First C#*, Andrew and Jenny have presented an excellent tutorial on learning C#. It is very approachable while covering a great amount of detail in a unique style. If you've been turned off by more conventional books on C#, you'll love this one."

—Jay Hilyard, Director and Software Security Architect, and author of *C# 6.0 Cookbook*

"I'd recommend this book to anyone looking for a great introduction into the world of programming and C#. From the first page onward, the authors walk the reader through some of the more challenging concepts of C# in a simple, easy-to-follow way. At the end of some of the larger projects/labs, the reader can look back at their programs and stand in awe of what they've accomplished."

—David Sterling, Principal Software Developer

"*Head First C#* is a highly enjoyable tutorial, full of memorable examples and entertaining exercises. Its lively style is sure to captivate readers—from the humorously annotated examples to the Fireside Chats, where the abstract class and interface butt heads in a heated argument! For anyone new to programming, there's no better way to dive in."

—Joseph Albahari, inventor of LINQPad, and coauthor of *C# 12 in a Nutshell* and *C# 12 Pocket Reference*

"[*Head First C#*] was an easy book to read and understand. I will recommend this book to any developer wanting to jump into the C# waters. I will recommend it to the advanced developer that wants to understand better what is happening with their code. [I will recommend it to developers who] want to find a better way to explain how C# works to their less-seasoned developer friends."

—Giuseppe Turitto, Director of Engineering

"Andrew and Jenny have crafted another stimulating Head First learning experience. Grab a pencil, a computer, and enjoy the ride as you engage your left brain, right brain, and funny bone."

—Bill Mietelski, Advanced Systems Analyst

"Going through this *Head First C#* book was a great experience. I have not come across a book series which actually teaches you so well.…This is a book I would definitely recommend to people wanting to learn C#."

—Krishna Pala, MCP

Praise for the Head First Approach

"I received the book yesterday and started to read it…and I couldn't stop. This is definitely très 'cool.' It is fun, but they cover a lot of ground and they are right to the point. I'm really impressed."

—Erich Gamma, IBM Distinguished Engineer, and coauthor of *Design Patterns*

"One of the funniest and smartest books on software design I've ever read."

— Aaron LaBerge, SVP Technology & Product Development, ESPN

"What used to be a long trial and error learning process has now been reduced neatly into an engaging paperback."

— Mike Davidson, former VP of Design, Twitter, and founder of Newsvine

"Elegant design is at the core of every chapter here, each concept conveyed with equal doses of pragmatism and wit."

— Ken Goldstein, Executive VP & Managing Director, Disney Online

"Usually when reading through a book or article on design patterns, I'd have to occasionally stick myself in the eye with something just to make sure I was paying attention. Not with this book. Odd as it may sound, this book makes learning about design patterns fun.

"While other books on design patterns are saying 'Bueller…Bueller…Bueller…' this book is on the float belting out 'Shake it up, baby!'"

— Eric Wuehler

"I literally love this book. In fact, I kissed this book in front of my wife."

— Satish Kumar

Head First C#

Andrew Stellman
Jennifer Greene

O'REILLY®

Beijing • Boston • Farnham • Sebastopol • Tokyo

13 Bonus Chapter

Events and Delegates

Your objects are starting to think for themselves.

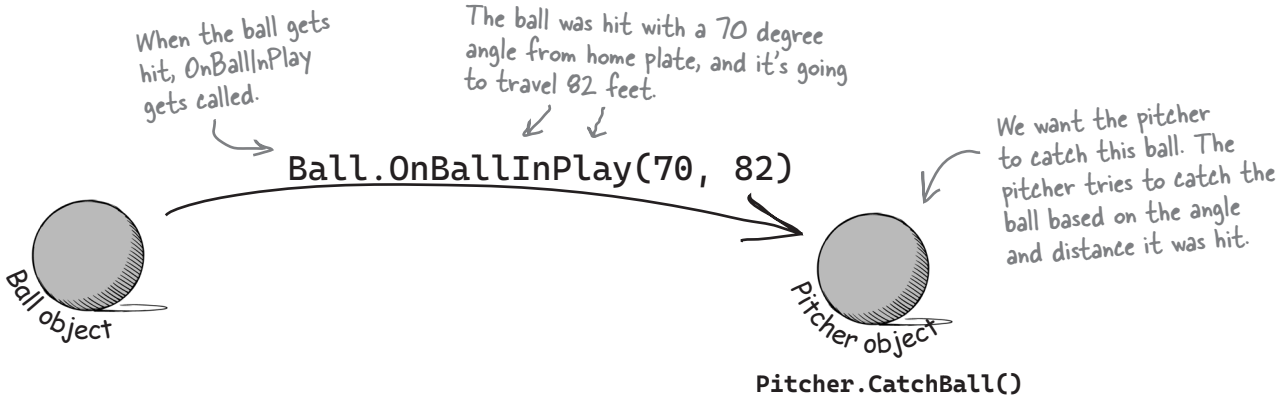
You can't always control what your objects are doing. Sometimes things…happen. When they do, you want your objects to be smart enough to **respond to anything** that pops up, and that's what events are all about. One object *publishes* an event, other objects *subscribe*, and everyone works together to keep things moving… which is great, until you want your object to take control over who can listen. That's when **callbacks** will come in handy. What makes all of that work? **Delegates**—like Func and Action—that let you create references to methods.

Ever wish your objects could think for themselves?

Suppose you're writing a baseball simulator. You're going to model a game, sell the software to the Yankees (they've got deep pockets, right?), and make a million bucks. You create your Ball, Pitcher, Umpire, and Fan objects, and a whole lot more. You even write code so that the Pitcher object can catch a ball.

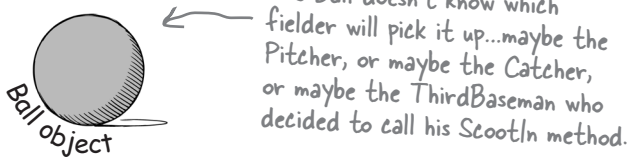
We named this method OnBallInPlay because it happens ON the occasion that the ball is in play.

Now you just need to connect everything together. You add an OnBallInPlay method to Ball, and now you want your Pitcher object to respond with its event handler method. Once the methods are written, you just need to tie the separate methods together:



But how does an object KNOW to respond?

Here's the problem. You want your Ball object to only worry about getting hit, and your Pitcher object to only worry about catching balls that come its way. In other words, you don't want the Ball telling the Pitcher, "I'm coming to you."



You want an object to worry about itself, not other objects. You're separating the concerns of each object.

This doesn't mean that objects can't interact. It just means that a Ball shouldn't determine who fields it. That's not the Ball's job.

When an EVENT occurs...objects listen

What you need to do when the ball is hit is to use an **event**. An event is simply something that's happened in your program. Then, other objects can respond to that event—like our Pitcher object.

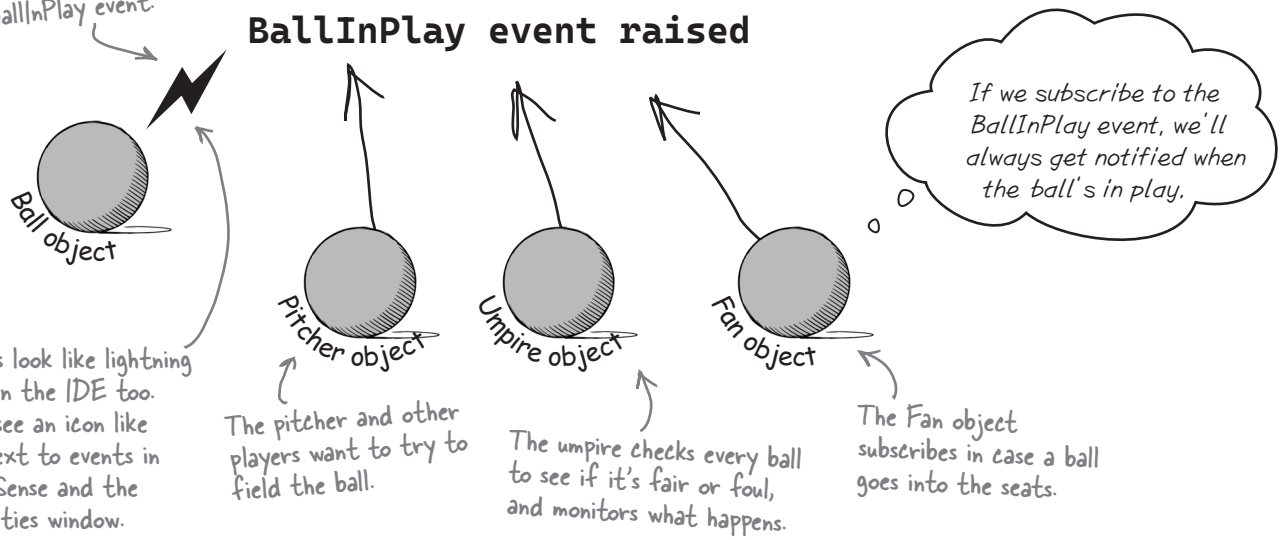
Even better, more than one object can listen to the same event. The Pitcher could listen for a ball-being-hit event, as well as a Catcher, a ThirdBaseman, an Umpire, even a Fan. Each object can respond to the event differently.

So what we want is a Ball object that can **raise an event**. Then, we want to have other objects to **subscribe to that particular type of event**—that just means to listen for it and get notified when that event occurs.

event, noun.
a **thing** that happens, especially something of importance. *The solar eclipse was an amazing **event** to behold.*

Any object can subscribe to this event...and the Ball object doesn't need to know which objects are subscribed.

When a Ball gets hit, it raises a BallInPlay event.



Events look like lightning bolts in the IDE too. You'll see an icon like this next to events in IntelliSense and the properties window.

The pitcher and other players want to try to field the ball.

The umpire checks every ball to see if it's fair or foul, and monitors what happens.

The Fan object subscribes in case a ball goes into the seats.

Want to DO SOMETHING with an event? You need an event handler

Once your object "hears" about an event, you can set up some code to run. That code is called an **event handler**. An event handler gets information about the event and runs every time that event occurs.

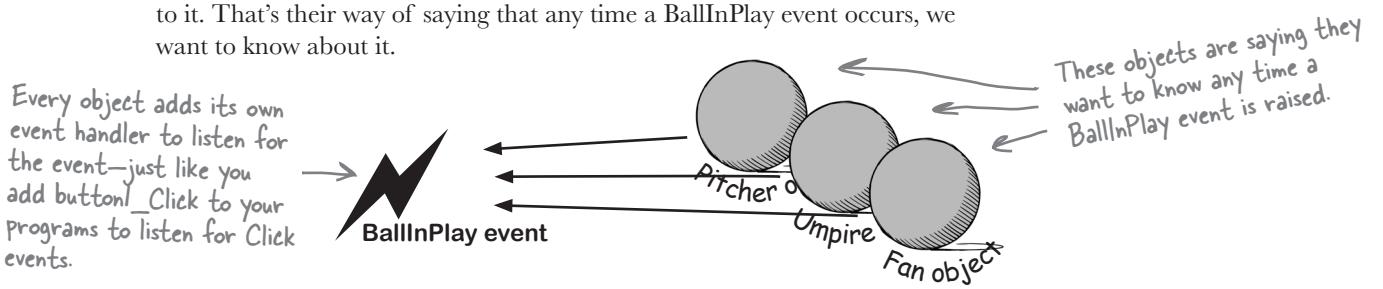
Remember, all this happens ***without your intervention*** at runtime. So you write code to raise an event, and then you write code to handle those events and fire up your application. Then, whenever an event is raised, your handler kicks into action…*without you doing anything*. Best of all, your objects have separate concerns. They're worrying about themselves, not other objects.

We've been doing this all along. Every time you click a button, an event is raised, and your code responds to that event.

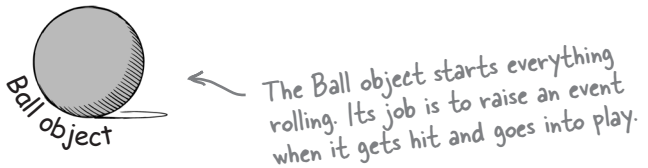
One object raises its event, others listen for it...

An event has a **publisher** and can have multiple **subscribers**. Let's take a look at how events, event handlers, and subscriptions work in C#:

1 **First, other objects subscribe to the event.**
Before the Ball can raise its BallInPlay event, other objects need to subscribe to it. That's their way of saying that any time a BallInPlay event occurs, we want to know about it.

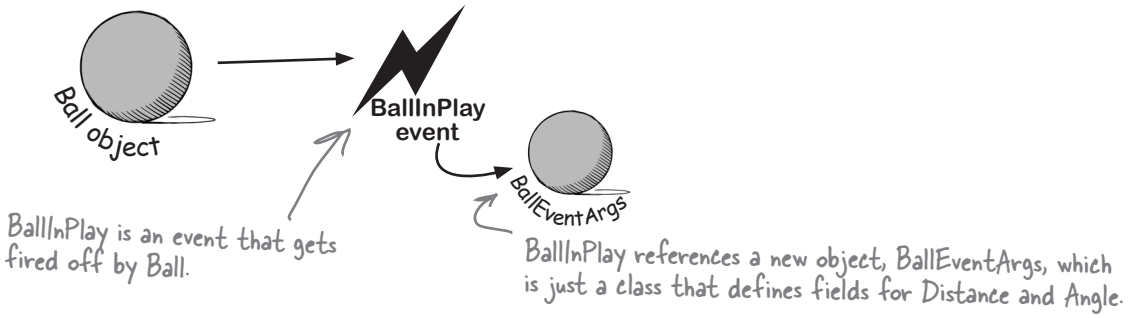


2 **Something triggers an event.**
The ball gets hit. It's time for the Ball object to raise a new event.



Sometimes we'll talk about raising an event, or firing it, or invoking it—they're all the same thing. People just use different names for it.

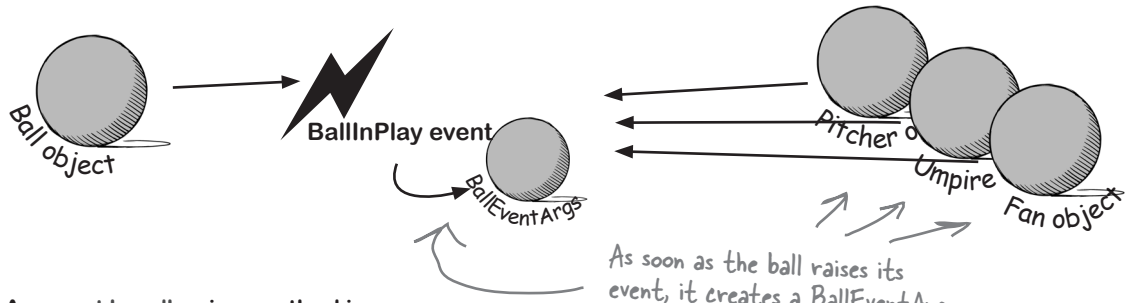
3 **The ball raises an event.**
A new event gets raised (we'll talk about exactly how that works in just a minute). That event also has some arguments, like the velocity of the ball, as well as its angle. Those arguments are attached to the event as an instance of an EventArgs object, and then the event is sent off, available to anyone listening for it.



...then, the other objects handle the event

Once an event is raised, all the objects subscribed to that event get a notification it happened, which lets them do something:

4 **Subscribers get notified.**
Since the Pitcher, Umpire, and Fan object subscribed to the Ball object's BallInPlay event, they all get notified—all of their event handler methods get called one after another.

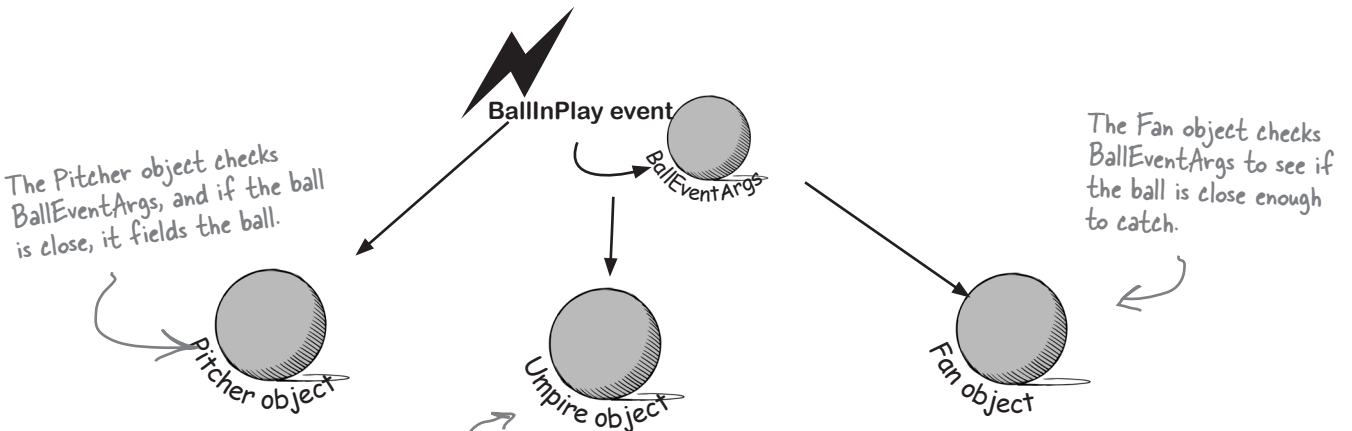


An event handler is a method in the subscriber object that gets run when the event is raised.

As soon as the ball raises its event, it creates a BallEventArgs object with the ball's angle and distance so it can pass it to the subscribers' event handlers.

Events are handled on a first-come, first-served basis—the object that subscribes first gets notified first.

5 **Each object handles the event.**
Now, Pitcher, Umpire, and Fan can all handle the BallInPlay event in their own way. But they don't all run at the same time—their event handlers get called one after another, with a reference to a BallEventArgs object as its parameter.



The Pitcher object checks BallEventArgs, and if the ball is close, it fields the ball.

The Fan object checks BallEventArgs to see if the ball is close enough to catch.

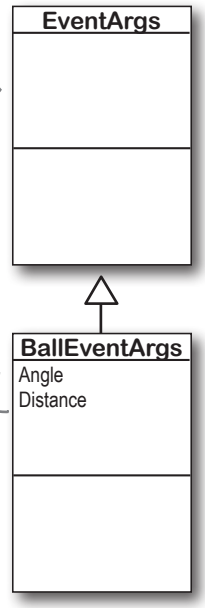
The Umpire watches. It might even subscribe to other events, like BallFielded or BallThrown, to further react to what happens.

How events work in a real app

Now that you've got a handle on what's going on, let's take a closer look at how the pieces fit together. Luckily, there are only a few moving parts. Let's see how they fit together *before* we start writing code (but don't worry, we'll use events in an app soon!).

It's a good idea (although not required) for your event argument objects to inherit from EventArgs. That's an **empty class**—it has no public members.

It means that you can **upcast** your EventArgs object in case you need to send it to an event that doesn't handle it in particular.



The ball will use these properties to pass information to the event handlers about where the ball's been hit.

1 **We need an object for the event arguments.**
Remember, our BallInPlay event has a few arguments that it carries along. So we need a very simple object for those arguments. .NET has a standard class for it called **`EventArgs`**. We'll extend that class so we can pass our arguments to the event—in this case, the angle of the baseball and the distance it was hit.

```
class BallEventArgs : EventArgs {
    public int Angle { get; private set; }
    public int Distance { get; private set; }
}
```

2 **Next, we'll need to define the event in the class that'll raise it.**
The ball class will have a line with the **`event` keyword**—this is how it informs other objects about the event, so they can subscribe to it. This line can be anywhere in the class—it's usually near the property declarations. But as long as it's in the Ball class, other objects can subscribe to a ball's event. You saw the event keyword when you fired PropertyChanged events. Here's the BallInPlay event declaration:

```
public event EventHandler? BallInPlay;
```

After the event keyword comes EventHandler. That's not a reserved C# keyword—it's a class that's part of .NET.

Events are usually public. This event is defined in the Ball class, but we'll want Pitcher, Umpire, etc., to be able to reference it. You could make it private if you only wanted other instances of the same class to subscribe to it.

3 The subscribing classes need event handler methods.

Every object that has to subscribe to the Ball's BallInPlay event needs to have an event handler. You already know how event handlers work—you've added methods in WPF, Blazor, and Unity that are called any time buttons are clicked. The Ball's BallInPlay event is no different, and an event handler for it should look pretty familiar:

```
void Ball_BallInPlay(object? sender, EventArgs e)
```

There's no C# rule that says your event handlers need to be named a certain way, but there's a pretty standard naming convention: the name of the object reference, followed by an underscore, followed by the name of the event.

The class that has this particular event handler method has a Ball reference variable called ball, so its BallInPlay event handler starts with "ball_", followed by the name of the event being handled, "BallInPlay".

The BallInPlay event declaration listed its event type as EventHandler, which means that it needs to take two parameters—an object called sender and an EventArgs called e—and have no return value.

4 Each individual object subscribes to the event.

Once we've got the event handler set up, the various Pitcher, Umpire, ThirdBaseman, and Fan objects need to hook up their own event handlers. Each one of them will have its own specific Ball_BallInPlay method that responds differently to the event. So if there's a Ball object reference variable or field called ball, then the += operator will hook up the event handler:

```
ball.BallInPlay += new EventHandler(Ball_BallInPlay);
```

This tells C# to hook the event handler up to the BallInPlay event of whatever object the ball reference is pointing to.

The += operator tells C# to subscribe an event handler to an event.

This part specifies which event handler method to subscribe to the event.

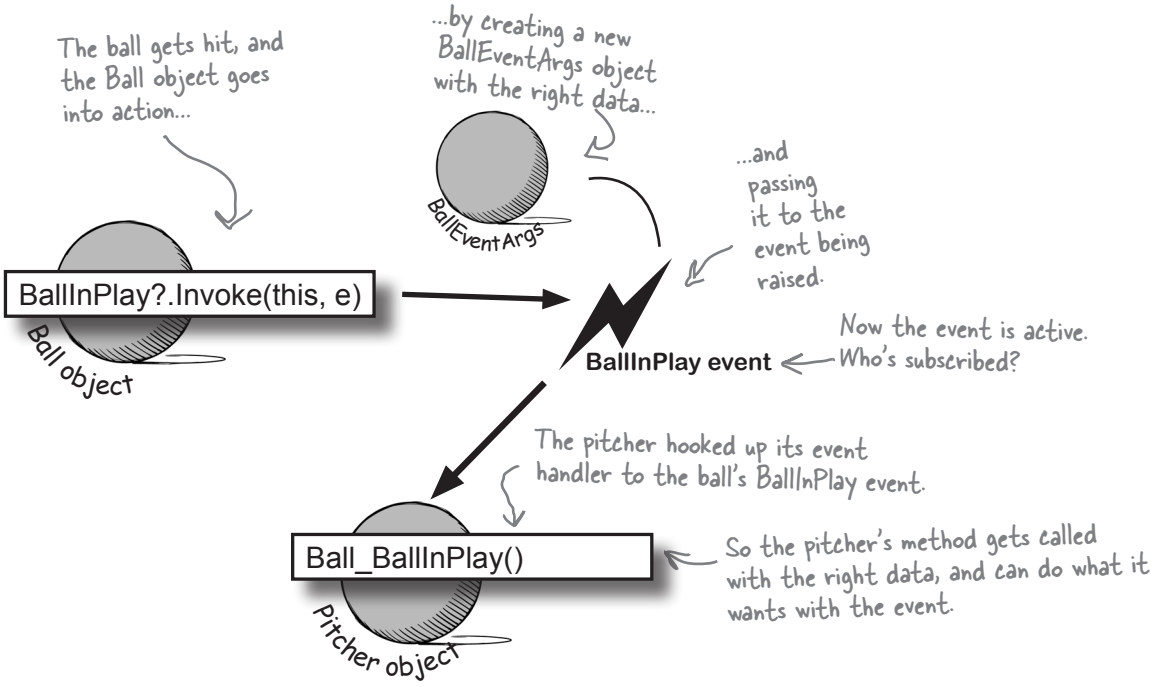
The event handler method's signature (its parameters and return value) has to **match** the one defined by EventHandler or the program **won't compile**.

Turn the page; there's a little more...

5 **A Ball object *raises* its event to notify subscribers that it's in play.**

Now that the events are all set up, the Ball can **raise its event** in response to something else that happens in the simulator. Use the **null conditional operator ?. to call the BallInPlay.Invoke method**:

```
var e = new BallEventArgs(75, 105);
BallInPlay?.Invoke(this, e);
```



Use the ?. null conditional operator to raise events

When you've declared an event like BallInPlay, you can actually call it like this: BallInPlay(this, e)

There's one problem: if you raise an event with no handlers, it'll throw an exception. That means no other objects have used += to add their event handlers to the BallInPlay event, BallInPlay will be null, so calling BallInPlay(this, e) will throw a NullReferenceException.

That's why you use the **?. null conditional operator** – here's how you'd use it: ***BallInPlay.?Invoke(this, e);***

That's the same as doing this: ***var ballInPlay = BallInPlay;if (ballInPlay != null) BallInPlay(this, e);***

It's a shorter, easier-to-read way to do the same thing. Plus, there's a very rare case where BallInPlay is not null when you check it, but it could actually become null before the next statement is executed, which ?. takes care of for you.

there are no Dumb Questions

Q: Why do I need to include the word EventHandler when I declare an event? I thought the event handler was what the other objects used to subscribe to the events.

A: That's true—when you need to subscribe to an event, you write a method called an event handler. But did you notice how we used EventHandler in the event declaration (step #2) and in the line to subscribe the event handler to it (step #4)? What EventHandler does is define the signature of the event—it tells the objects subscribing to the event exactly how they need to define their event handler methods. Specifically, it says that if you want to subscribe a method to this event, it needs to take two parameters (an object and an EventArgs reference) and have a void return value.

Q: What happens if I try to use a method that doesn't match the ones that are defined by EventHandler?

A: Then your program won't compile. The compiler will make sure that you don't ever accidentally subscribe an incompatible event handler method to an event. That's why the standard event handler, EventHandler, is so useful—as soon as you see it, you know exactly what your event handler method needs to look like.

Q: Wait, "standard" event handler? There are other kinds of event handlers?

A: Yes! Your events don't have to send an object and an EventArgs. In fact, they can send anything at all—or nothing at all! Look at the IntelliSense window at the bottom of the facing page. Notice how the OnDragEnter method takes a DragEventArgs reference instead of an EventArgs reference? DragEventArgs inherits from EventArgs, just like BallEventArgs does. The page's DragDrop event doesn't use EventHandler. It uses something else, DragEventHandler, and if you want to handle it, your event handler method needs to take an object and a DragEventArgs reference.

The parameters of the event are defined by a delegate—EventHandler and DragEventHandler are two examples of delegates. We'll talk more about delegates later in the chapter.

Q: So I can probably have my event handlers return something other than void, too, right?

A: Well, you can, but it's often a bad idea. If you don't return void from your handler, you can't chain event handlers. That means you can't connect more than one handler to each event. Since chaining is a handy feature, you'd do best to always return void from your event handlers.

Q: Chaining? What's that?

A: It's how more than one object can subscribe to the same event—they chain their event handlers onto the event, one after another. We'll talk a lot more about that in a minute, too.

Q: Is that why we use += when when when we add event handlers? Like We're somehow adding a new handler to existing handlers?

A: Exactly! Any time you add an event handler, you want to use +=. That way, your handler doesn't replace existing handlers. It just becomes one in what may be a very long chain of other event handlers, all of which are listening to the same event.

Q: Why does the ball use "this" when it raises the BallInPlay event?

A: Because that's the first parameter of the standard event handler. Have you noticed how every Click event handler method has a parameter "object? sender"? That parameter is a reference to the object that's raising the event. So if you're handling a button click, sender points to the button that was clicked. If you're handling a BallInPlay event, sender will point to the Ball object that's in play—and the ball sets that parameter to this when it raises the event.

A SINGLE event is always raised by a SINGLE object.

But a SINGLE event can be responded to by MULTIPLE objects.

The IDE generates event handlers for you automatically

Many programmers follow the same convention for naming their event handlers. If there's a Ball object that has a BallInPlay event and the name of the reference holding the object is called ball, then the event handler would typically be named Ball_BallInPlay. That's not a hard-and-fast rule, but if you write your code like that, it'll be a lot easier for other programmers to read.

Luckily, the IDE makes it easy to name your event handlers this way. It has a feature that **automatically adds event handler methods for you** when you're working with a class that raises an event. It shouldn't be too surprising that the IDE can do this for you—after all, this is exactly what it does when you double-click on a button in the designer. (This may seem familiar because you've done it in earlier chapters.)

1 **Create a new console app and add the Ball and BallEventArgs classes.**
First add this BallEventArgs class:

```
class BallEventArgs : EventArgs {
    public int Angle { get; private set; }
    public int Distance { get; private set; }

    public BallEventArgs(int angle, int distance) {
        this.Angle = angle;
        this.Distance = distance;
    }
}
```

Here's the BallEventArgs class we showed you earlier. We added a constructor to make the code easier to read.

Then add this Ball class:

```
class Ball
{
    public event EventHandler? BallInPlay;

    public void OnBallInPlay(BallEventArgs e) => BallInPlay?.Invoke(this, e);
}
```

2 **Add a Pitche class and start creating the Pitcher's constructor.**
Add a **new Pitcher class** to your project. Then give it a constructor that takes a Ball reference called ball as a parameter. There will be one line of code in the constructor to add its event handler to ball.BallInPlay. Start typing the statement, but **don't type += yet**.

```
public Pitcher(Ball ball) => ball.BallInPlay
```

3 Type += and the IDE will finish the statement for you.

As soon as you type **+=** in the statement, the IDE displays a very useful little box:

```
ball.BallInPlay += Ball_BallInPlay;    (Press TAB to insert)
```

When you press the Tab key, the IDE will finish the statement for you. It'll look like this:

```
public Pitcher(Ball ball) => ball.BallInPlay += Ball_BallInPlay;
```

VSCode might not pop up the "Press TAB to insert" box, so instead you can also finish this statement and then use the IDE's Generate Method feature.

You have a choice – you can either press Tab to add the method (which VSCode may not support), or you can type in the whole line of code and use the Generate Method from the Quick Actions menu (Ctrl+. or ⌘+.) to create the method for you.

4 Examine the new event handler method that the IDE created.

Afte you used Tab create the method or typed in the whole line of code for the consttuctor and used Generate Method to create it, the IDE created a new nmethod for you:

```
void Ball_BallInPlay(object? sender, EventArgs e) {
    throw new NotImplementedException();
}
```

Take a look at the arguments. The first argument is an a nullable object reference called Sender. If this is not null, it will contain a reference to the object that fired the event. The second argument is the EventArgs that we just leaned about. We'll learn more about how to use this.

Just like you've seen before, the IDE always fills in this NotImplementedException as a placeholder, so if you run the code it'll throw an exception that tells you that you still need to implement something it filled in automatically.

5 Finish the pitcher's event handler.

Now that you've got the event handler's skeleton added to your class, fill in the rest of its code. The pitcher should catch any low balls; otherwise, he covers first base.

```
private int pitchNumber = 0;
void Ball_BallInPlay(object? sender, EventArgs e)
{
    pitchNumber++;
    if (e is BallEventArgs ballEventArgs)
    {
        if ((ballEventArgs.Distance < 95) && (ballEventArgs.Angle < 60))
            Console.WriteLine($"Pitch #{pitchNumber}: I caught the ball");
        else
            Console.WriteLine($"Pitch #{pitchNumber}: I covered first base");
    }
}
```

Since BallEventArgs is a subclass of EventArgs, we'll downcast it using the is keyword so we can use its properties.

Here's what you have so far in your app

Your app now has four parts:

- The top-level statements.
- The Ball class with a BallInPlay event.
- The BallEventArgs class that you use to pass arguments to the BallInPlay event.
- The Pitcher class with an event handler method that takes a BallEventArgs argument.

Here's how they all work together to make the app work.

1 **The Pitcher class listens to the Ball.BallInPlay event.**
The Pitcher class's constructor takes a Ball reference and uses += to add the event handler to its BallInPlay event:

```
public Pitcher(Ball ball) => ball.BallInPlay += Ball_BallInPlay;
```

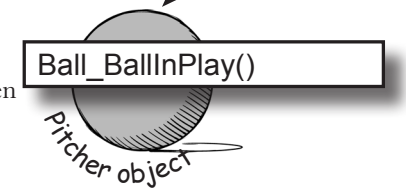
2 **The top-level statements calls the Ball.OnBallInPlay method to tell the ball that it's in play.**
Your Ball class provided a convenient OnBallInPlay method to tell it the play has started, so the top-level statements just need to call it:

```
BallEventArgs ballEventArgs = new BallEventArgs(angle, distance);
ball.OnBallInPlay(ballEventArgs);
```

3 **The Ball class raises its BallInPlay event.**
The ball uses the ?. null conditional operator to raise the event:

```
BallInPlay?.Invoke(this, e);
```

4 **The event calls any event handlers that are listening to it.**
Since the Pitcher already added its to the Ball.BallIinPlay event, when the Ball raises its event the Pitcher's Ball_BallInPlay method gets called ***automatically***.



Exercise

It's time to put what you've learned so far into practice. Your job is to complete the Ball and Pitcher classes, add a Fan class, and make sure they all work together with a very basic version of your baseball simulator.

Step 1: Add a Fan class.

Create another class called Fan. Fan should also subscribe to the BallInPlay event in its constructor. The fan's event handler should see if the distance is greater than 400 feet and the angle is greater than 30 (a home run), and grab for a glove to try to catch the ball if it is. If not, the fan should scream and yell. Everything that the fan screams and yells should be written to the console.

Your Fan class will be very similar to the Pitcher class that we gave you. That's okay for this exercise! You could use inheritance to reduce duplicated code, but for this exercise keep the two classes separate because it will help you learn about events more effectively.

Look closely at the output from the console app to see exactly what the Fan class writes to the console.

Step 2: Implement the Top-level statements.

Here's the output that your app should produce. Your job is to get your app to match this output exactly. We put the user put in boldface so you could see exactly how it works.

```
Enter a number for the angle (or anything else to quit): 75
Enter a number for the distance (or anything else to quit): 105
Pitch #1: I covered first base
Pitch #1: Woo-hoo! Yeah!
Enter a number for the angle (or anything else to quit): 48
Enter a number for the distance (or anything else to quit): 80
Pitch #2: I caught the ball
Pitch #2: Woo-hoo! Yeah!
Enter a number for the angle (or anything else to quit): 40
Enter a number for the distance (or anything else to quit): 435
Pitch #3: I covered first base
Pitch #3: Home run! I'm going for the ball!
Enter a number for the angle (or anything else to quit): 125
Enter a number for the distance (or anything else to quit): 25
Pitch #4: I covered first base
Pitch #4: Woo-hoo! Yeah!
Enter a number for the angle (or anything else to quit): bye
Thanks for playing!
```

Exercise Solution

Here's the Fan class. It works just like the Pitcher class:

```
class Fan
{
    private int pitchNumber = 0;

    public Fan(Ball ball) => ball.BallInPlay += Ball_BallInPlay;

    void Ball_BallInPlay(object? sender, EventArgs e)
    {
        pitchNumber++;
        if (e is BallEventArgs ballEventArgs)
        {
            if (ballEventArgs.Distance > 400 && ballEventArgs.Angle > 30)
               Console.WriteLine(
                         $"Pitch #{pitchNumber}: Home run! I'm going for the ball!");
            else
               Console.WriteLine($"Pitch #{pitchNumber}: Woo-hoo! Yeah!");
        }
    }
}
```

The fan's BallInPlay event handler looks for any ball that's high and long.

Here's are the top-level statements. There are references to the Ball, Pitcher, and Fan objects. This code calls the Ball object's OnBallInPlay method to raise its BallInPlay event, which the Pitcher and Fan handle:

```
var ball = new Ball();
var pitcher = new Pitcher(ball);
var fan = new Fan(ball);

var running = true;
while (running)
{
    Console.Write("Enter a number for the angle (or anything else to quit): ");
    if (int.TryParse(Console.ReadLine(), out int angle))
    {
       Console.Write("Enter a number for the distance (or anything else to quit): ");
       if (int.TryParse(Console.ReadLine(), out int distance))
       {
           BallEventArgs ballEventArgs = new BallEventArgs(angle, distance);
           ball.OnBallInPlay(ballEventArgs);
       }
       else
           running = false;
    }
    else
        running = false;
}
Console.WriteLine("Thanks for playing!");
```

The Fan and Pitcher object constructors chain their event handlers onto the BallInPlay event.

The top-level statements just need to call the Ball object's OnBallInPlay method. The event takes care of signaling to the Pitcher and Fan that the ball is in play. The top-level statements don't even need to know that's happening.

You've been using event handlers throughout the book

Let's do a little experiment to see where you've been using event handlers. Create a **new .NET MAUI app** project and add this button to the top of the page:

```
<Button x:Name="WasIClickedButton" Text="Was I clicked?"/>
```

Now go to the MainPage constructor in MainPage.xaml.cs and start typing this statement:

```
public MainPage()
{
    InitializeComponent();

    WasIClickedButton.Clicked +=
}
```

As soon as you press += the IDE will pop up this box:

```
WasIClickedButton.Clicked +=
                              WasIClickedButton_Clicked;    (Press TAB to insert)
```

Press the Tab key to add the WasIClickedButton_Clicked event handle method:

```
private void WasIClickedButton_Clicked(object? sender, EventArgs e)
{
    throw new NotImplementedException();
}
```

Compare the method declaration to any other button clicked event handler from any of your other apps. It takes exactly the same arguments. Add code to safely cast sender to a Button and use it to update the text:

```
private void WasIClickedButton_Clicked(object? sender, EventArgs e)
{
    if (sender is Button wasIClickedButton)
        wasIClickedButton.Text = "Yes, I was clicked!";
}
```

Run the app and test it. This event handler works just like the BallInPlay event you just created.

You've used events with Blazor, too

Have you been using the Blazor Learner's Guide? Blazor buttons don't use event handlers the same way MAUI does. But you have used an event in a Blazor app. Go back to Chapter 1 and have find this line of code in the OnInitialized method that sets up the game's timer:

```
timer.Elapsed += Timer_Tick;
```

The Timer class fires its Elapsed event every time it ticks. Now look at the method that it calls:

```
private void Timer_Elapsed(object? sender, ElapsedEventArgs e)
```

That's an event handler method, just like the ones that you've been using. The only difference is that it uses ElapsedEventArgs, which extends EventArgs—just like your BallEventArgs class extended EventArgs.

Use a generic EventHandlers with your own event types

Take a look at the event declaration in your Ball class:

```
public event EventHandler? BallInPlay;
```

We know that definitely works, because you've been using it in your app.

But this EventHandler can take any of those types—you could pass it an ElapsedEventArgs, TextChangeEventArgs, or ChangeEventArgs.

We know that the BallEventHandler will always pass it a BallEventArgs when the event is fired. Luckily, .NET gives us a great tool to communicate that information very easily: a generic EventHandler. **Change your ball's BallInPlay event declaration** so it looks like this:

```
public event EventHandler<BallEventArgs>? BallInPlay;
```

Run your app again. It should still work.

Modify your event handlers to use specific types

Now that you're using a generic event handler that only accepts BallEventArgs, your event handler will always be passed an argument of that type.

There's an advantage to using a generic event handler. Right now your Pitcher class has this line of code:

```
if (e is BallEventArgs ballEventArgs)
```

When you use a generic event handler, you can ***use the specific event type in the declaration***. You just need to make sure that the event type you're using—like BallEventArgs—is a subclass of EventHandler.

That means don't need to use the `is` keyword to downcast the EventArgs.

Modify the event handler in your Pitcher class to take a BallEventArgs parameter, and remove the `if` statement with the `is` keyword:

```
void Ball_BallInPlay(object? sender, BallEventArgs e)
{
    pitchNumber++;
    if ((e.Distance < 95) && (e.Angle < 60))
        Console.WriteLine($"Pitch #{pitchNumber}: I caught the ball");
    else
        Console.WriteLine($"Pitch #{pitchNumber}: I covered first base");
}
```

Now that your event handler takes a BallEventArgs, you can use its Distance and Angle properties without upcasting.

Run your app again. It builds and works just like before, but now that you're using a BallEventArgs parameter, you don't need to downcast it.

Try making the same change to your Fan class—change the Ball_BallInPlay event handler declaration to use BallEventArgs, then remove the if statement with the upcast and use the event properties directly.

Add multiple event handlers to the same event

Here's a really useful thing that you can do with events: you can **chain** them so that one event or delegate calls many methods, one after another. Let's see how this works—and get some practice with generic event handlers.

Create a new Console app – we'll use it to test chained event handlers.

Do this!

1. **Add a TalkEventArgs class to send to your event.**

   We'll use this to pass messages to our events. It has a string property with a message to send, and a constructor that sets the property:

```
class TalkEventArgs : EventArgs
{
    public string Message { get; private set; }

    public TalkEventArgs(string message) => Message = message;
}
```

2. **Add a Talker class with the event to raise.**

   It has an event called TalkToMe and a method that uses `?.` to raise that event:

```
class Talker
{
    public event EventHandler<TalkEventArgs>? TalkToMe;

    public void OnTalkToMe(string message) =>
                    TalkToMe?.Invoke(this, new TalkEventArgs(message));
}
```

You're using a generic event handler to work with your TalkEventArgs event class.

3. **Add the top level statements.**

   Start by adding the methods we'll chain onto the event:

```
int count = 0;

void SaySomething(object? sender, TalkEventArgs e) {
    Console.WriteLine($"Call #{count++}: I said something: {e.Message}");
}

void SaySomethingElse(object? sender, TalkEventArgs e) {
    Console.WriteLine($"Call #{count++}: I said something else: {e.Message}");
}
```

When you use += to chain multiple event handlers to the same event, they're called in the order they were added.

Finish the top-level statements and run your app

The top-level statements prompt the user to chain additional methods or raise the event.

```
var myEvent = new Talker();
while (true)
{
    Console.Write("1 to chain SaySomething, 2 to chain SaySomethingElse, or a message: ");
    var line = Console.ReadLine();
    switch (line)
    {
        case "1":
            Console.WriteLine("Adding SaySomething");
            myEvent.TalkToMe += SaySomething;
            break;
        case "2":
            Console.WriteLine("Adding SaySomethingElse");
            myEvent.TalkToMe += SaySomethingElse;
            break;
        case "":
            return;
        default:
            count = 1;
            Console.WriteLine("Raising the TalkToMe event");
            if (!string.IsNullOrEmpty(line))
                myEvent.OnTalkToMe(line);
            break;
    }
}
```

Now run your app. Try adding SaySometing then sending a message. It will call the event:

```
1 to chain SaySomething, 2 to chain SaySomethingElse, or a message: 1
Adding SaySomething
1 to chain SaySomething, 2 to chain SaySomethingElse, or a message: Hello
Raising the TalkToMe event
Call #1: I said something: Hello
```

Keep your program running. Chain SaySomethingElse—now it calls that after it calls SaySomething:

```
1 to chain SaySomething, 2 to chain SaySomethingElse, or a message: 2
Adding SaySomethingElse
1 to chain SaySomething, 2 to chain SaySomethingElse, or a message: Talking
Raising the TalkToMe event
Call #1: I said something: Talking
Call #2: I said something else: Talking
```

Chain the same methods several times. It will call them in the order that you added them:

```
1 to chain SaySomething, 2 to chain SaySomethingElse, or a message: 2
Adding SaySomethingElse
1 to chain SaySomething, 2 to chain SaySomethingElse, or a message: 1
Adding SaySomething
1 to chain SaySomething, 2 to chain SaySomethingElse, or a message: 1
Adding SaySomething
1 to chain SaySomething, 2 to chain SaySomethingElse, or a message: More
Raising the TalkToMe event
Call #1: I said something: More
Call #2: I said something else: More
Call #3: I said something else: More
Call #4: I said something: More
Call #5: I said something: More
```

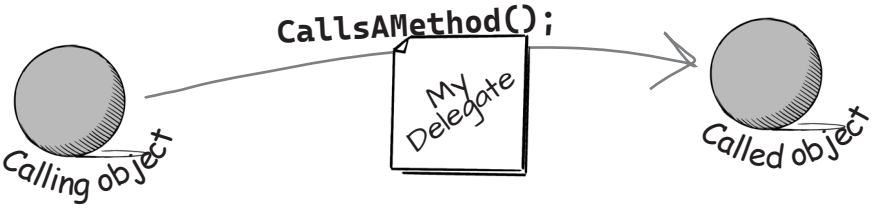
You can chain the same method onto an event multiple times. When the event is raised, it will call all of the chained methods in the order that they were added.

Something is different about EventHandler. It's ***like a class*** *– we use it as a type. But we never use += with classes, and we can't use classes to call methods. It's something different, right?*

You're right! EventHandler is a delegate.

A delegate is a type that represents a **reference to a method**. In Chapter 4, we talked about types and references, and how a reference is like a sticky note for an object. Delegates are also references, but instead of using a sticky note to label an object, you're using one to label a method instead.

Back in Chapter 4 we talked about how creating a reference is like writing a sticky note and putting it on an object. A delegate is like writing a sticky note and putting it on a method.



EventHandler is a delegate

Let's take a minute and use Visual Studio to explore the syntax for delegates. Open any class in and add this event:

```
public event EventHandler? MyEvent;
```

Now use ***Go to Definition (F12)*** to see how the EventHandler type is defined. Here's what you'll see (you may need to expand the XMLDoc comments):

```
//
// Summary:
//     Represents the method that will handle an event when the event provides data.
//
//
// Parameters:
//   sender:
//     The source of the event.
//
//   e:
//     An object that contains the event data.
//
// Type parameters:
//   TEventArgs:
//     The type of the event data generated by the event.
public delegate void EventHandler<TEventArgs>(object? sender, TEventArgs e);
```

When you use the `delegate` keyword to define a delegate, you can use it as a type, just like how you can declare a class, and then use it in your code as a type.

Here's how a delegate works:

1. **Use the delegate keyword to declare a delegate.**
   You declare a delegate just like you declare a method: it has a return value, a name, and parameters. You can also add access modifiers. Here's a delegate that returns a string and takes one int parameter:

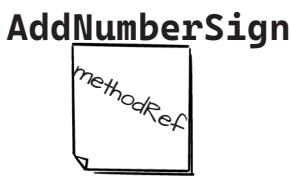
   ```
   delegate string IntToString(int i)
   ```

2. **Use the delegate to declare a variable.**
   If you have a method with an int parameter that returns a string:

   ```
   string AddNumberSign(int i) => $"#{i}";
   ```

   You can use the delegate in a variable declaration

   ```
   IntToString methodRef;
   methodRef = AddNumbeSign;
   ```



3. **Use the delegate to call the method.**
   Use your new variable to call the method that it's pointing to:

   ```
   var output = methodRef(12345);
   ```

Create a simple app and use a delegate.

Do this!

Let's take a few minutes and write some simple code to explore how the delegate keyword works. **Create a new console app.** Replace the top-level statements with this line:

```
delegate string IntToString(int i);
```

Congratulations, you've now declared a delegate. Now let's use it.

del-e-gate, noun.
a person sent or authorized to represent others. *The president sent a* ***delegate*** *to the summit.*

Use your delegate to declare a variable that references a method

Add a method to your Program class—you'll call it from Main, so make sure it's static. It just takes an int and adds a number sign to it:

```
delegate string IntToString(int i);

public static string AddNumberSign(int i) => $"#{i}";
```

Now use the IntToString delegate to declare a variable, and call it:

```
IntToString methodRef = AddNumberSign;
Console.WriteLine(methodRef(12345));
```

Run your app—it writes `#12345` to the console.

AddNumberSign

Point your delegate to a different method

Add the PlusOne method to your app. Then modify your

```
public static string AddNumberSign(int i) => $"#{i}";
public static string PlusOne(int i) => $"{i} plus one equals {i + 1}";

static void Main(string[] args)
{
    IntToString methodRef = AddNumberSign;
    Console.WriteLine(methodRef(12345));

    methodRef = PlusOne;
    Console.WriteLine(methodRef(12345));
}
```

You pointed the methodRef variable to your AddNumberSign method, then you used it to call the method and printed its return value.

You changed methodRef to point to your PlusOne method, so now when you call methodRef(12345) it calls PlusOne instead of AddNumberSign.

Run your app again.

Now it prints a second line: `12345 plus one equals 12346`

PlusOne

IntToString is a delegate that can be used as a type. methodRef is a variable that has IntToString as its type. First it pointed to the AddNumberSign method, then it pointed to the PlusOne method.

Use delegates to call methods in objects

You can point a delegate to a method in a specific object. Let's see how this works by creating an app to help a restaurant owner sort out their top chef's secret ingredients.

Do this!

1 **Create a new Console Application project and add a delegate.**

Delegates usually appear outside of any other classes, so add a new class file to your project and call it *GetSecretIngredient.cs*. It will have exactly one line of code inside the `namespace { ... }`:

```
delegate string GetSecretIngredient(int amount);
```

Make sure you delete the class declaration entirely, so this is the only line in the file. This is a delegate with an int parameter that returns a string, just like the one you used on the previous page.

2 **Add a class for the first chef, Adrian.**

Adrian.cs will hold a class that keeps track of the first chef's secret ingredient. It has a private method called AdriansSecretIngredient with a signature that matches GetSecretIngredient. But it also has a read-only property—and check out that property's type. It returns a GetSecretIngredient. So other objects can use that property to get a reference to her AdriansSecretIngredient method—the property can return a delegate reference to it, even though it's private.

Adrian's secret ingredient method takes an int called amount and returns a string that describes her secret ingredient.

```
class Adrian {
  public GetSecretIngredient MySecretIngredientMethod {
    get {
      return AddAdriansSecretIngredient;
    }
  }
  private string AddAdriansSecretIngredient(int amount) {
      return $"{amount} ounces of cloves";
  }
}
```

3 **Add a class for the second chef, Harper.**

The HarpersSecretIngredientMethod property returns a new instance of the GetSecretIngredient delegate that's pointing to her secret ingredient method.

Harper's method works a lot like Adrian's:

Harper's secret ingredient method also takes an int called amount and returns a string, but it returns a different string from Adrian's.

```
class Harper {
    public GetSecretIngredient HarpersSecretIngredientMethod {
        get {
            return AddHarpersSecretIngredient;
        }
    }
    private int total = 20;
    private string AddHarpersSecretIngredient(int amount) {
        if (total - amount < 0)
            return $"I don't have {amount} cans of sardines!";
        else {
            total -= amount;
            return $"{amount} cans of sardines";
        }
    }
}
```

4 **Add code to prompt the user for a chef or an amount.**

Here's the code for the top-level statements:

```
GetSecretIngredient addIngredientMethod = null;

while (true)
{
    Console.Write("Enter A for Adrian, H for Harper, or an amount: ");
    var line = Console.ReadLine();
    switch (line)
    {
        case "A":
            Console.WriteLine("Selected Adrian");
            addIngredientMethod = adrian.MySecretIngredientMethod;
            break;
        case "H":
            Console.WriteLine("Selected Harper");
            addIngredientMethod = harper.HarpersSecretIngredientMethod;
            break;
        default:
            if (addIngredientMethod is null)
                Console.WriteLine("Please select a chef!");
            else if (int.TryParse(line, out int amount))
                Console.WriteLine(addIngredientMethod(amount));
            else
                return;
            break;
    }
}
```

5 **Run the app.**

Enter A set the delegate to use the Adrian object's MySecerentInredientMethod property to set the delegate to its private ingredient, then enter an amount. Then switch to the Harper object and enter an amount—it calls the other object's private method. Switch between them to get different secret ingredients.

```
Enter A for Adrian, H for Harper, or an amount: A
Selected Adrian
Enter A for Adrian, H for Harper, or an amount: 14
14 ounces of cloves
Enter A for Adrian, H for Harper, or an amount: H
Selected Harper
Enter A for Adrian, H for Harper, or an amount: 16
16 cans of sardines
Enter A for Adrian, H for Harper, or an amount: 5
I don't have 5 cans of sardines!
Enter A for Adrian, H for Harper, or an amount: A
Selected Adrian
Enter A for Adrian, H for Harper, or an amount: 5
5 ounces of cloves
Enter A for Adrian, H for Harper, or an amount:
```

When you type H the app switches from the Adrian object's secret ingredient method to point the delegate to the Harper object's method.

Use the debugger to explore how delegates work.

You've got a great tool—the IDE's debugger—that can really help you get a handle on how delegates work:

- Start by running your program. Enter the input we gave it on the previous page and make sure your output looks the same: enter **A** to select Adrian's method, give it an amount of **14**, then enter **H** to switch to Harper's method, enter **26** and then **3**, and then enter **A** to switch back to Adrian's method and enter **3**. **Stop the program.**

- **Place a breakpoint** on each of the lines that sets the addIngredientMethod variable:

```
case "A":
    Console.WriteLine("Selected Adrian");
    addIngredientMethod = adrian.MySecretIngredientMethod;
    break;
case "H":
    Console.WriteLine("Selected Harper");
    addIngredientMethod = harper.HarpersSecretIngredientMethod;
    break;
```

- Run the program and enter **A** to select Adrian's method. When it breaks on the first breakpoint, watch the locals window. addIngredientMethod should be null:

Name	Value
addIngredientMethod	null

- Step over the statement that sets the addIngredientMethod delegate. It now points to the private AddAdriensSecretIngredient.

Name	Value
addIngredientMethod	{Method = {System.String AddAdriansSecretIngredient(Int32)}}

- Enter H to select Harper's method, then step over the breakpoint and watch the delegate change:

Name	Value
addIngredientMethod	{Method = {System.String AddHarpersSecretIngredient(Int32)}}

- Place a breakpoint on the first line of the Harper.AddHarpersSecretIngredient method:

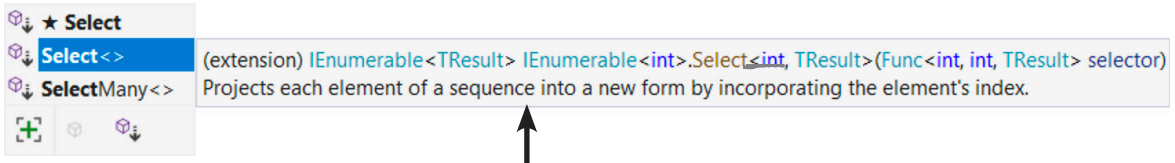
```
private string AddHarpersSecretIngredient(int amount)
{
    if (total - amount < 0)
        return $"I don't have {amount} cans of sardines!";
```

Enter a number. The breakpoint breaks inside that method. Thetop-level statements call a private method inside an object, which is accessing a private field in that object.

LINQ and List<T> use the Func and Action delegates

In Chapter 9 you saw that the LINQ Select method takes a parameter of type Func. Let's use the IDE to explore it. **Create a new Console Application called ExploreFuncAndAction** and add these lines:

```
Enumerable.Range(1, 5)
    .Select
```



When the IDE shows you an IntelliSense window for the Select method, look for the parameter type: Func. Func is a delegate that can point to a method that returns a value. If you declare a variable of type Func<int, string> you can use it to reference any method that takes an int parameter and returns a string. LINQ uses Func delegates in its extension methods so you can pass them methods and lambda expressions.

Let's use the IDE to explore Func. **Replace the top-level statements** with these lines:

```
Func<int, string> timesFour = (int i) => $"-> {i * 4} <-";

Enumerable.Range(1, 5)
    .Select(timesFour);
```

Func is a delegate with on in parameter and one out parameter, which means it takes one argument and returns a value. There are Func delegates defined with up to 15 parameters and a return value.

The Action delegate can point to a menthod that doesn't return a value

There's another delegate that .NET classes use: **the Action delegate**, which can execute actions by referencing methods without a return value. Update your code so it converts the IEnumerable<string> to a List<string> and calls ForEach:

Use the IDE to go to the definition of the Action delegate. It has an in parameter but no out parameter.

```
Func<int, string> timesFour = (int i) => $"-> {i * 4} <-";

int lineNumber = 1;
Action<string> writeLine = (string s) =>
                  Console.WriteLine($"Line {lineNumber++} is {s}");

Enumerable.Range(1, 5)
    .Select(timesFour)
    .ToList()
    .ForEach(writeLine);
```

Action is a delegate that can point to a method that does not return a value. The List<T> class has a ForEach method that takes an Action parameter and iterates through the list.

Func and Action Up Close

Right-click on the `Func` keyword in the console app you just created and choose Go to Definition (F12). Expand the comments if necessary (these are generated automatically from XMLDoc):

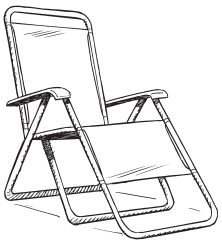
```
//
// Summary:
//     Encapsulates a method that has one parameter and returns a value of the type
//     specified by the TResult parameter.
//
// Parameters:
//   arg:
//     The parameter of the method that this delegate encapsulates.
//
// Type parameters:
//   T:
//     The type of the parameter of the method that this delegate encapsulates.
//
//   TResult:
//     The type of the return value of the method that this delegate encapsulates.
//
// Returns:
//     The return value of the method that this delegate encapsulates.
public delegate TResult Func<in T, out TResult>(T arg);
```

Now right-click on `Action` and view its definition:

```
//
// Summary:
//     Encapsulates a method that has a single parameter and does not return a value.
//
//
// Parameters:
//   obj:
//     The parameter of the method that this delegate encapsulates.
//
// Type parameters:
//   T:
//     The type of the parameter of the method that this delegate encapsulates.
public delegate void Action<in T>(T obj);
```

Compare the two Summary sections. They both talk about how delegates **encapsulate** methods. Func encapsulates a method that returns a value, Action encapsulates a method that does not return a value.

Look carefully at the two declarations. There's one difference—Func has an extra parameter. You'll learn about the `out` keyword in Chapter 11, but you've been using it throughout the book when you've used the int.TryParse method. But you don't need to know the details to get a sense of how the two delegates work.



Pool Puzzle

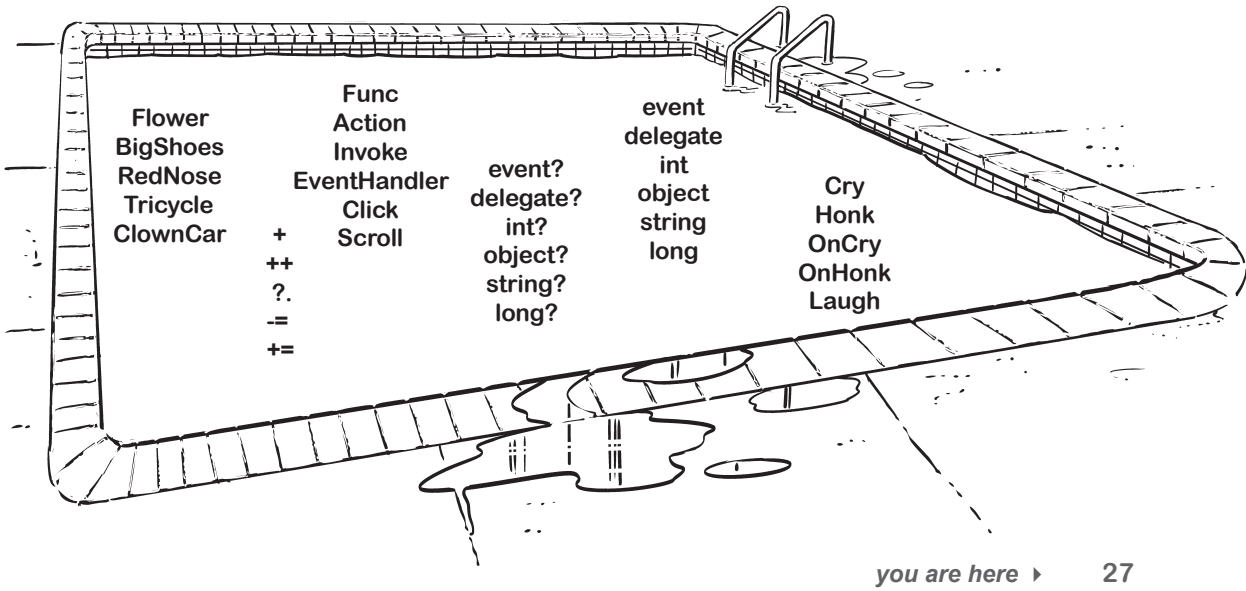
Your ***job*** is to take snippets from the pool and place them into the blank lines in the code. You can use the same snippet more than once, and you won't need to use all the snippets. Your ***goal*** is to complete the code for a form that writes this output to the console when it runs.

Output

```
Fingers is coming to get you!
```

```
____________<_______.___, _______.___> evil = (string s) => $"{s}ming t{s}";

____________<_______.___, string, _______.___> kill = (string x, string y) => $"{y}{x}";

____________<_______.___, string> slice = (string q) => " " + q;

____________<_______.___> terrify = (string s) => Console.WriteLine(s);

____________<_______.___> laugh = (____.______ sender, _______.___ e) => terrify(e);

var laughter = new __________();
laughter.Honk _________ laugh;
laughter.___________(kill(evil("o"), "gers is c"), kill(slice("you"), "get"));

class RedNose
{
    public event _________________<string>? _________;

    public void _________(string noise, string fun) =>
                  Honk ______  ________  (this, $"Fin{noise} {fun}");
}
```

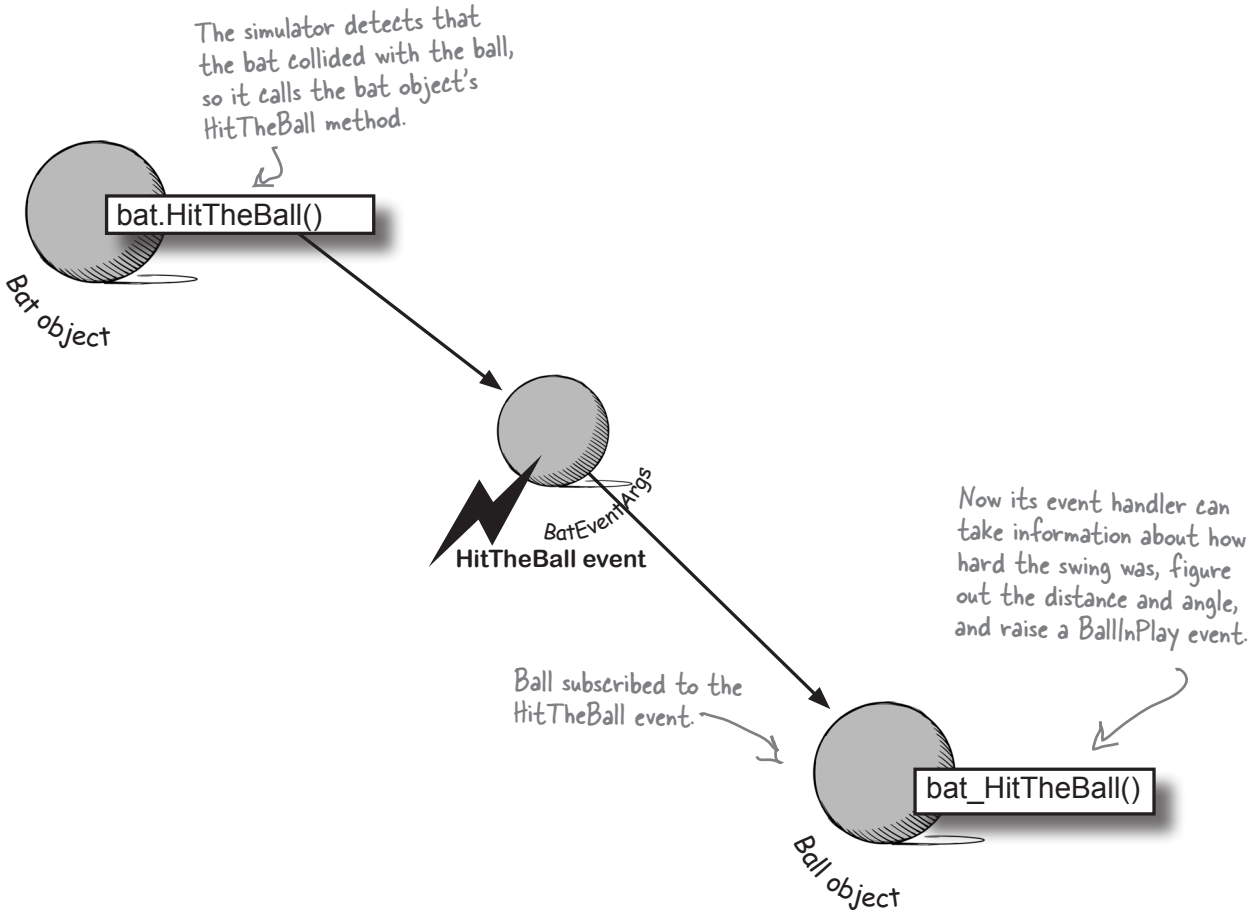
Note: Each thing from the pool can be used more than once.



An object can subscribe to an event...

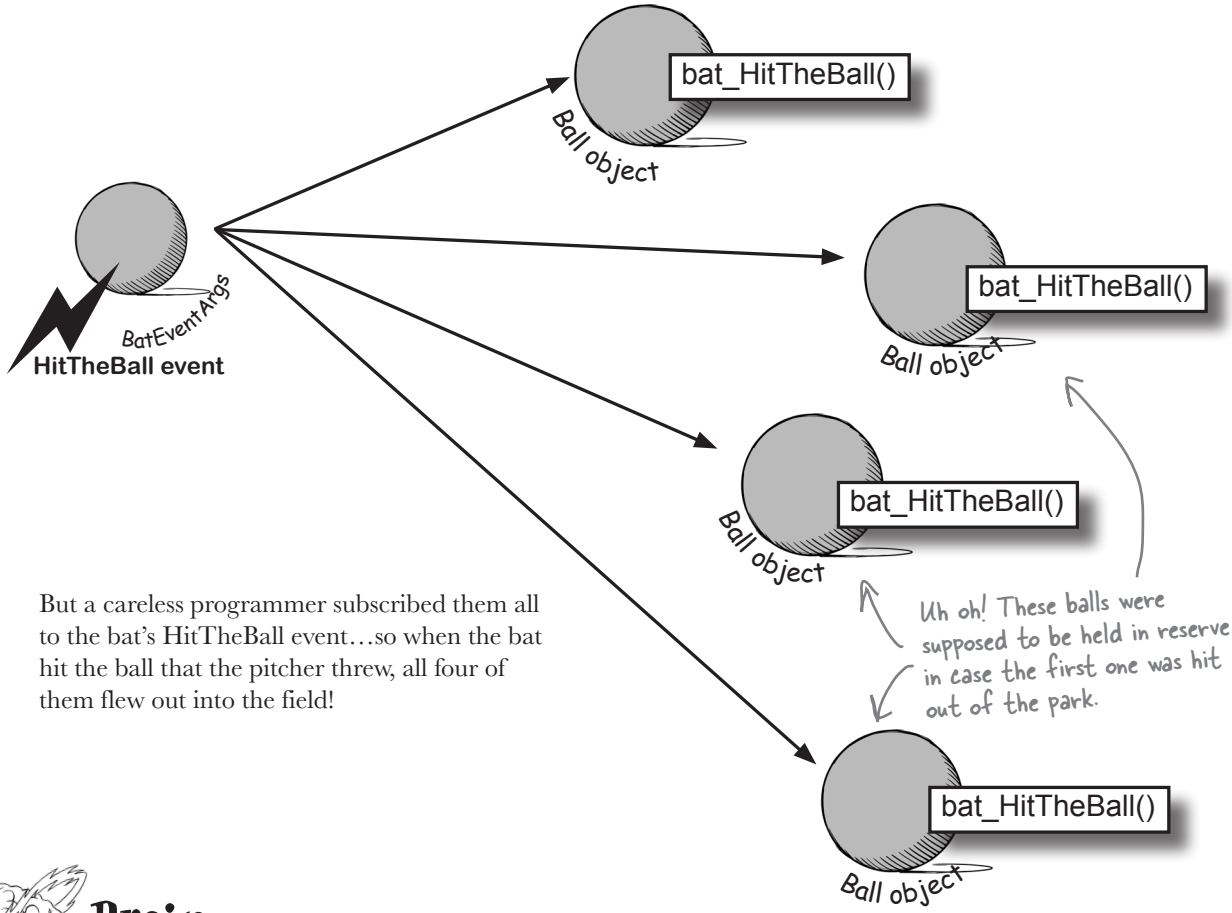
Suppose we add a new class to our simulator, a Bat class, and that class adds a HitTheBall event into the mix. Here's how it works: if the simulator detects that the player hit the ball, it calls the Bat object's HitTheBall method, which raises a HitTheBall event.

So now we can add a bat_HitTheBall method to the Ball class that subscribes to the Bat object's HitTheBall event. Then, when the ball gets hit, its own event handler calls its OnBallInPlay method to raise its own event, BallInPlay, and the chain reaction begins. Fielders field, fans scream, umpires yell…we've got a ball game.



...but that's not always a good thing!

There's only ever going to be one ball in play at any time. But if the Bat object uses an event to announce to the ball that it's been hit, then any Ball object can subscribe to it. That means we've set ourselves up for a nasty little bug—what happens if a programmer accidentally adds three more Ball objects? Then the batter will swing, hit, and **four different balls will fly** out into the field!



But a careless programmer subscribed them all to the bat's HitTheBall event…so when the bat hit the ball that the pitcher threw, all four of them flew out into the field!

Brain Power

This bug happened because a programmer used the HitTheBall event in a way that it wasn't intended. Earlier in the book, we learned how to use **encapsulation** to make it difficult to misuse our classes by making fields, properties, and methods private. Is there a way we could use encapsulation to make sure only one Ball object at a time is every hooked up to the bat?

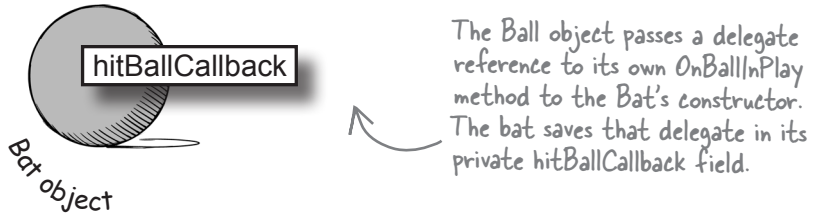
Use a callback to control who's listening

Our system of events only works if we've got one Ball and one Bat. If you've got several Ball objects, and they all subscribe to the public event HitTheBall, then they'll all go flying when the event is raised. But that doesn't make any sense…it's really only one Ball object that got hit. We need to let the one ball that's being pitched hook itself up to the bat, but we need to do it in a way that doesn't allow any other balls to hook themselves up.

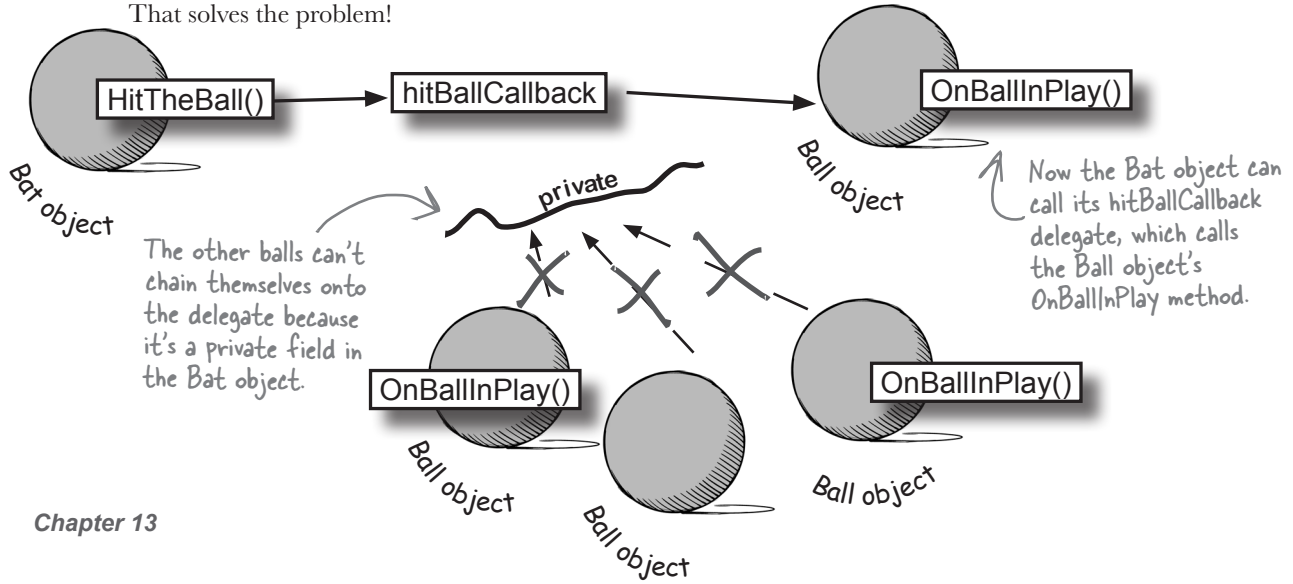
That's where a **callback** comes in handy. It's a technique that you can use with delegates. Instead of exposing an event that anyone can subscribe to, an object uses a method (often a constructor) that takes a delegate as an argument and holds onto that delegate in a private field. We'll use a callback to make sure that the Bat notifies exactly one Ball:

1 **The Bat will keep its delegate field private.**
The easiest way to keep the wrong Ball objects from chaining themselves onto the Bat's delegate is for the bat to make it private. That way, it has control over which Ball object's method gets called.

2 **The Bat's constructor takes a delegate that points to a method in the ball.**
When the ball is in play, it creates the new instance of the bat, and it passes the Bat object a pointer to its OnBallInPlay method. This is called a **callback method** because the Bat is using it to call back to the object that instantiated it.



3 **When the bat hits the ball, it calls the callback method.**
But since the bat kept its delegate private, it can be 100% sure that no other ball has been hit. That solves the problem!



The Case of the Golden Crustacean

Henry "Flatfoot" Hodgkins is a TreasureHunter. He's hot on the trail of one of the most prized possessions in the rare and unusual aquatic-themed jewelry markets: a jade-encrusted translucent gold crab… but so are lots of other TreasureHunters. They all got a reference to the same crab in their constructor, but Henry wants to claim the prize ***first***.

In a stolen set of class diagrams, Henry discovers that the GoldenCrab class raises a RunForCover event every time anyone gets close to it. Even better, the event includes NewLocationArgs, which detail where the crab is moving to. But none of the other treasure hunters know about the event, so Henry figures he can cash in.

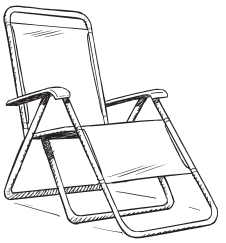
Henry adds code to his constructor to register his treasure_RunForCover method as an event handler for the RunForCover event on the crab reference he's got. Then, he sends a lowly underling after the crab, knowing it will run away, hide, and raise the RunForCover event—giving Henry's treasure_RunForCover method all the information he needs.

Everything goes according to plan, until Henry gets the new location and rushes to grab the crab. He's stunned to see three other TreasureHunters already there, fighting over the crab.

How did the other treasure hunters beat Henry to the crab?

→ solution on page 34

Pool Puzzle Solution



```
____Func____<__string__, __string__> evil = (string s) => $"{s}ming t{s}";

____Func____<__string__, string, __string__> kill = (string x, string y) => $"{y}{x}";

____Func____<__string__, string> slice = (string q) => " " + q;

___Action___<__string__> terrify = (string s) => Console.WriteLine(s);

_EventHandler_<__string__> laugh = (__object?__ sender, __string__ e) => terrify(e);

var laughter = new __RedNose__();
laughter.Honk ___+=___ laugh;
laughter.__OnHonk__(kill(evil("o"), "gers is c"), kill(slice("you"), "get"));

class RedNose
{
    public event __EventHandler__<string>? __Honk__;

    public void __OnHonk__(string noise, string fun) =>
                  Honk __?.__ __Invoke__ (this, $"Fin{noise} {fun}");
}
```

A callback is a way to use delegates

A callback is a **different way of using a delegate**. It's not a new keyword or operator. It just describes a **pattern**—a way that you use delegates with your classes so that one object can tell another object, "Notify me when this happens—if that's OK with you!"

① Define another delegate in your baseball project.

We're going to add a Bat class with a private delegate field that points to the Ball object's OnBallInPlay method. But first, add a delegate that matches that method's signature.

Create a Bat class, and **add this delegate** to the Bat.cs file ***outside*** the class but still ***inside*** its namespace:

```
delegate void BatCallback(BallEventArgs e);
```

The Bat object's callback will point to a Ball object's OnBallInPlay method, so the callback's delegate needs to match the signature of OnBallInPlay()—so it needs to take a BallEventArgs parameter and have a void return value.

② Implement the Bat class.

The Bat class is simple. It's got a HitTheBall method that the simulator will call every time a ball is hit. That HitTheBall method uses the hitBallCallback delegate to call the ball's OnBallInPlay method (or whatever method is passed into its constructor).

```
class Bat
{
    private BatCallback hitBallCallback;

    public Bat(BatCallback callbackDelegate) => this.hitBallCallback = callbackDelegate;

    public void HitTheBall(BallEventArgs e) => hitBallCallback?.Invoke(e);
}
```

The point of the callback is that the object doing the calling is in control of who's listening. In an event, other objects demand to be notified by adding event handlers. In a callback, other objects simply turn over their delegates and politely ask to be notified.

③ Hook the bat up to a ball.

So how does the Bat's constructor get a reference to a particular ball's OnBallInPlay method? Add this GetNewBat method to the Ball class, which creates a new bat with a callback that's hooked up to that ball instance's OnBallInPlay method:

```
public Bat GetNewBat() => new Bat(new BatCallback(OnBallInPlay));
```

The Ball's new GetNewBat method creates a new Bat object, and it uses the BatCallBack delegate to pass a reference to its own OnBallInPlay method to the new bat. That's the callback method the bat will use when it hits the ball.

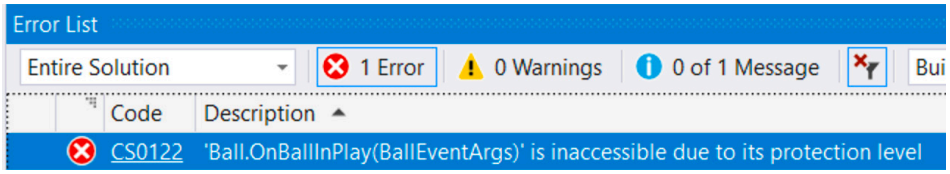
You'll set up the callback in the Bat object's constructor. But in some cases, it makes more sense to set up the callback method using a public method or property's set accessor.

4 Now we can encapsulate the Ball class a little better.

It's unusual for one of the On... methods that raise an event to be public. So let's follow that pattern with our ball, too, by making its OnBallInPlay method protected:

```
protected void OnBallInPlay(BallEventArgs e) => BallInPlay?.Invoke(this, e);
```

Now you'll see a compiler error because OnBallInPlay is inaccessible.



This is a really standard pattern that you'll see over and over again when you work with .NET classes. When a .NET class has an event that gets fired, you'll almost always find a protected method that starts with "On".

5 Fix the BaseballSimulator class.

BaseballSimulator can't call the Ball object's OnBallInPlay method anymore—which is exactly what we wanted (and why the IDE now shows an error). Instead, it needs to ask the Ball for a new bat in order to hit the ball. When it does, the Ball object will make sure that its OnBallInPlay method is hooked up to the bat's callback.

```
Console.Write("Enter a number for the distance (or anything else to quit): ");
if (int.TryParse(Console.ReadLine(), out int distance))
{
    BallEventArgs ballEventArgs = new BallEventArgs(angle, distance);
    var bat = ball.GetNewBat();
    bat.HitTheBall(ballEventArgs);
}
```

Now the way to hit the ball is to ask it for a Bat object with a callback that's already hooked up.

Now **run the program**—it should work exactly like it did before. But it's now **protected** from any problems that would be caused by more than one ball listening for the same event.

Check out *Head First Design Patterns*, another great book published by O'Reilly Media. It's a great way to learn about different patterns that you can apply to your own programs. The first one you'll learn about is called the Observer (or Publisher-Subscriber) pattern, and it'll look really familiar to you. One object publishes information, and other objects subscribe to it. Events are the C# way of implementing the Observer pattern.

The Case of the Golden Crustacean

How did the other treasure hunters beat Henry to the crab?

The crux of the mystery lies in how the treasure hunter seeks his quarry. First, we'll need to see exactly what Henry found in the stolen diagrams.

In a stolen set of class diagrams, Henry discovers that the GoldenCrab class raises a RunForCover event every time anyone gets close to it. Even better, the event includes NewLocationArgs, which detail where the crab is moving to. But none of the other treasure hunters know about the event, so Henry figures he can cash in.

```
class GoldenCrab {
    public delegate void Escape(object? sender, NewLocationArgs e);
    public event Escape? RunForCover;

    public void SomeonesNearby() =>
        Escape runForCover = RunForCover?.Invoke(this, new NewLocationArgs("Under the rock");
    }
}

class NewLocationArgs {
    public NewLocationArgs(HidingPlace newLocation) {
        this.newLocation = newLocation;
    }
    private HidingPlace newLocation;
    public HidingPlace NewLocation { get { return newLocation; } }
}
```

Any time someone comes close to the golden crab, its SomeonesNearby method fires off a RunForCover event, and it finds a place to hide.

So how did Henry take advantage of his newfound insider information?

Henry adds code to his constructor to register his treasure_RunForCover() method as an event handler for the RunForCover event on the crab reference he's got. Then, he sends a lowly underling after the crab, knowing it will run away, hide, and raise the RunForCover event—giving Henry's treasure_RunForCover() method all the information he needs.

```
class TreasureHunter {
    public TreasureHunter(GoldenCrab treasure) {
        treasure.RunForCover += treasure_RunForCover;
    }
    void treasure_RunForCover(object? sender, NewLocationArgs e) {
        MoveHere(e.NewLocation);
    }
    void MoveHere(HidingPlace location) {
        // ... code to move to a new location ...
    }
}
```

Henry thought he was being clever by altering his class's constructor to add an event handler that calls his MoveHere method every time the crab raises its RunForCover event. But he forgot that the other treasure hunters inherit from the same class, and his clever code adds their event handlers to the chain, too!

And that explains why Henry's plan backfired. When he added the event handler to the TreasureHunter constructor, he was inadvertently ***doing the same thing for all of the treasure hunters!*** And that meant that every treasure hunter's event handler got chained onto the same `RunForCover` event. So when the Golden Crustacean ran for cover, everyone was notified about the event. All of that would have been fine if Henry were the first one to get the message. But Henry had no way of knowing when the other treasure hunters would have been called—if they subscribed before he did, they'd get the event first.

there are no Dumb Questions

Q: How are callbacks different from events?

A: Events and delegates are part of C# and .NET. They're a way for one object to announce to other objects that something specific has happened. When one object publishes an event, any number of other objects can subscribe to it without the publishing object knowing or caring. When an object fires off an event, if anyone happens to have subscribed to it, then it calls their event handlers.

Callbacks are not part of .NET at all—instead, *callback* is just a name for the way we use delegates (or events—there's nothing stopping you from using a private event to build a callback). A callback is just a relationship between two classes where one object requests that it be notified. Compare this to an event, where one object demands that it be notified of that event.

Q: So a callback isn't an actual type in .NET?

A: No, it isn't. A callback is a pattern—it's just a novel way of using the existing types, keywords, and tools that C# comes with. Go back and take another look at the callback code you just wrote for the bat and ball. Did you see any new keywords that we haven't used before? Nope! But it does use a delegate, which is a .NET type.

It turns out that there are a lot of patterns that you can use. In fact, there's a whole area of programming called design patterns. A lot of problems that you'll run into have been solved before, and the ones that pop up over and over again have their own design patterns that you can benefit from.

Q: Does that mean callbacks are just private events?

A: No, not quite. It seems easy to think about it that way, but private events are a different beast altogether. Remember what the private access modifier really means? When you mark a class member private, only instances of that same class can access it. So if you mark an event private, then other instances of the same class can subscribe to it. That's different from a callback because it still involves one or more objects anonymously subscribing to an event.

Q: But it looks just like an event, except with the event keyword, right?

A: The reason a callback looks so much like an event is that they both use delegates. It makes sense that they both use delegates, because that's C#'s tool for letting one object pass another object a reference to one of its methods.

But the big difference between normal events and callbacks is that an event is a way for a class to publish to the world that some specific thing has happened. A callback, on the other hand, is never published. It's private, and the method that's doing the calling keeps tight control over who it's calling.

Bullet Points

- When you add a delegate to your project, you're **creating a new type** that stores references to methods.
- Events use delegates to notify objects that actions have occurred.
- Objects subscribe to an object's event if they need to react to something that happened in that object.
- An `EventHandler` is a kind of delegate you use to work with events.
- You can chain several event handlers onto one event. That's why you use += to assign a handler to an event.
- Always check that an event or delegate is not null before you use it to avoid a `NullReferenceException`.
- All of the controls in the toolbox use events to make things happen in your programs.
- When one object passes a reference to a method to another object so it—and only it—can return information, it's called a callback.
- Events let any method subscribe to your object's events anonymously, while callbacks let your objects exercise more control over which delegates they accept.
- Both callbacks and events use delegates to reference and call methods in other objects.
- The debugger is a really useful tool to help you understand how events, delegates, and callbacks work. Take advantage of it!