

O'REILLY®

Fourth
Edition

Head First

C#

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

Andrew Stellman
& Jennifer Greene



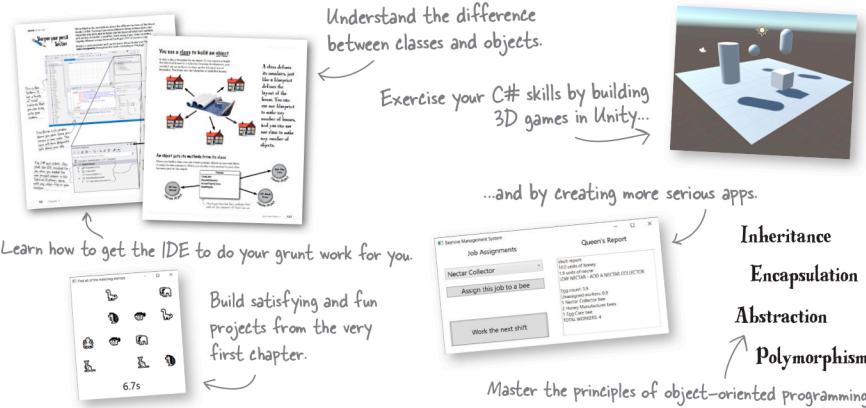
A Brain-Friendly Guide

Head First

C#

What will you learn from this book?

Dive into C# and create apps, user interfaces, games, and more using this fun and highly visual introduction to C#, .NET Core, and Visual Studio. With this completely updated guide, which covers C# 8.0 and Visual Studio 2019, beginning programmers like you will build a fully functional game in the opening chapter. Then 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, doing hands-on exercises, and building real-world applications. By the time you're done, you'll be a solid C# programmer—and you'll have a great time along the way!



What's so special about this book?

Based on the latest research in cognitive science and learning theory, *Head First C#* uses a visually rich format to engage your mind rather than a text-heavy approach that puts you to sleep. Why waste your time struggling with new concepts? This multisensory learning experience is designed for the way your brain really works.

.NET

US \$64.99

CAN \$85.99

ISBN: 978-1-491-97670-8



5 6 4 9 9
9 781491 976708

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

—Ryan White
Game Developer

"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

"If you want to learn
C# in depth and have
fun doing it, this is THE
book for you."

—Andy Parker
Fledgling C# programmer

O'REILLY®

Head First C#

Fourth Edition



Andrew Stellman
Jennifer Greene

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Head First C#

Fourth Edition

by Andrew Stellman and Jennifer Greene

Copyright © 2021 Jennifer Greene, Andrew Stellman. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Series Creators:

Kathy Sierra, Bert Bates

Cover Designer:

Ellie Volckhausen

Brain Image on Spine:

Eric Freeman

Editors:

Nicole Taché, Amanda Quinn

Proofreader:

Rachel Head

Indexer:

Potomac Indexing, LLC

Illustrator:

Jose Marzan

Page Viewers:

Greta the miniature bull terrier and Samosa the Pomeranian

Printing History:

November 2007: First Edition

May 2010: Second Edition

August 2013: Third Edition

December 2020: Fourth Edition



The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. The *Head First* series designations, *Head First C#*, and related trade dress are trademarks of O'Reilly Media, Inc.

Microsoft, Windows, Visual Studio, MSDN, the .NET logo, Visual Basic, and Visual C# are registered trademarks of Microsoft Corporation.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and the authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

No bees, space aliens, or comic book heroes were harmed in the making of this book.

ISBN: 978-1-491-97670-8

[LSI]

[2020-12-18]

Bonus Chapter events and delegates

What your code does when you're not looking



Your objects are starting to think for themselves.

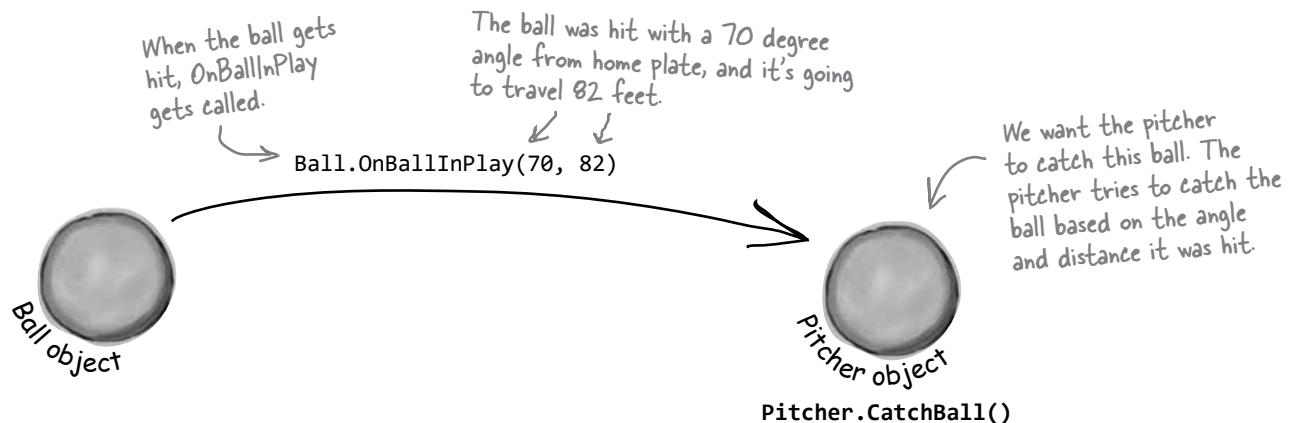
You can't always control what your objects are doing. Sometimes things...happen. When they do, you want your objects to be smart enough to **respond to anything** that pops up, and that's what events are all about. One object *publishes* an event, other objects *subscribe*, and everyone works together 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.

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:

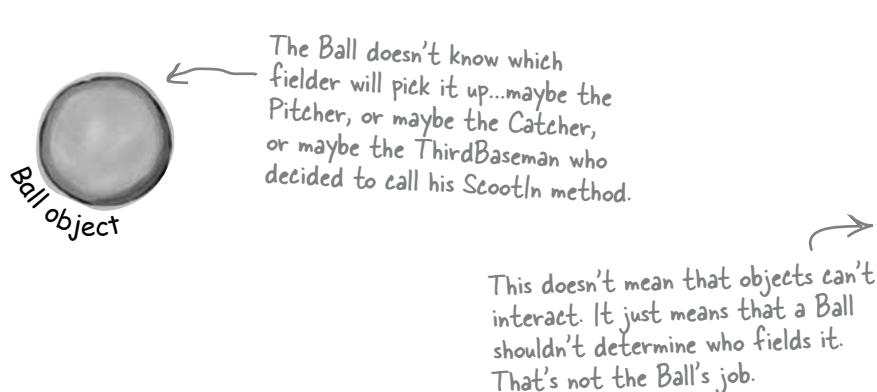
We named this method `OnBallInPlay` because it happens ON the occasion that the ball is in play.



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.



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.



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.

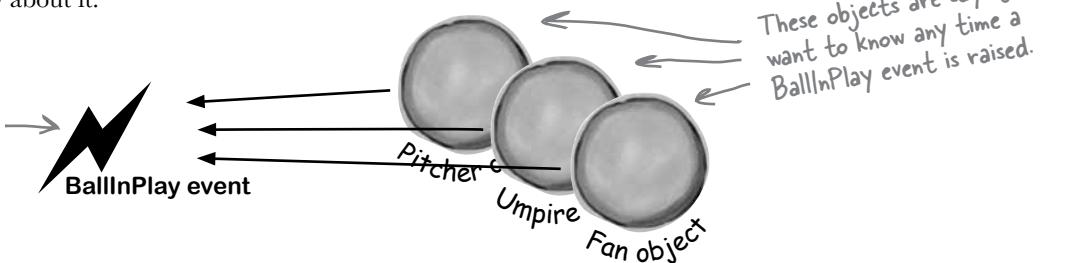
One object raises its event, others listen for it...

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.

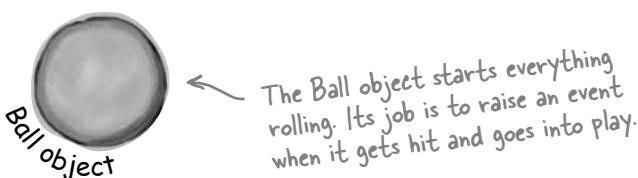
Before the Ball can raise its BallInPlay event, other objects need to subscribe to it. That's their way of saying that any time a BallInPlay event occurs, we want to know about it.

Every object adds its own event handler to listen for the event—just like you add button_Click to your programs to listen for Click events.



② Something triggers an event.

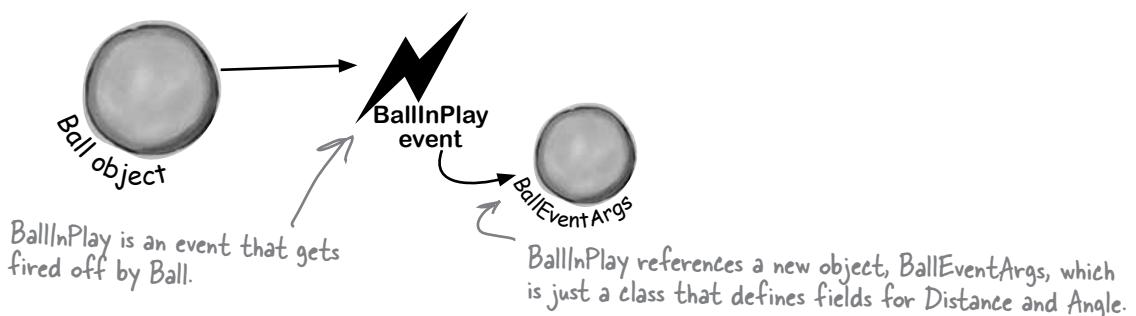
The ball gets hit. It's time for the Ball object to raise a new event.



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.

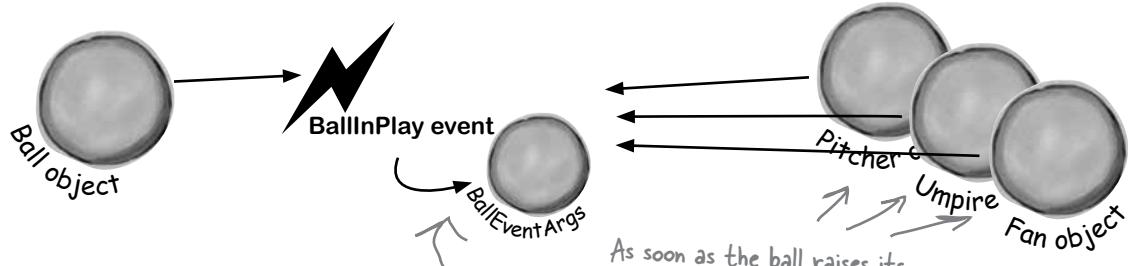


...then, the other objects handle the event

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.



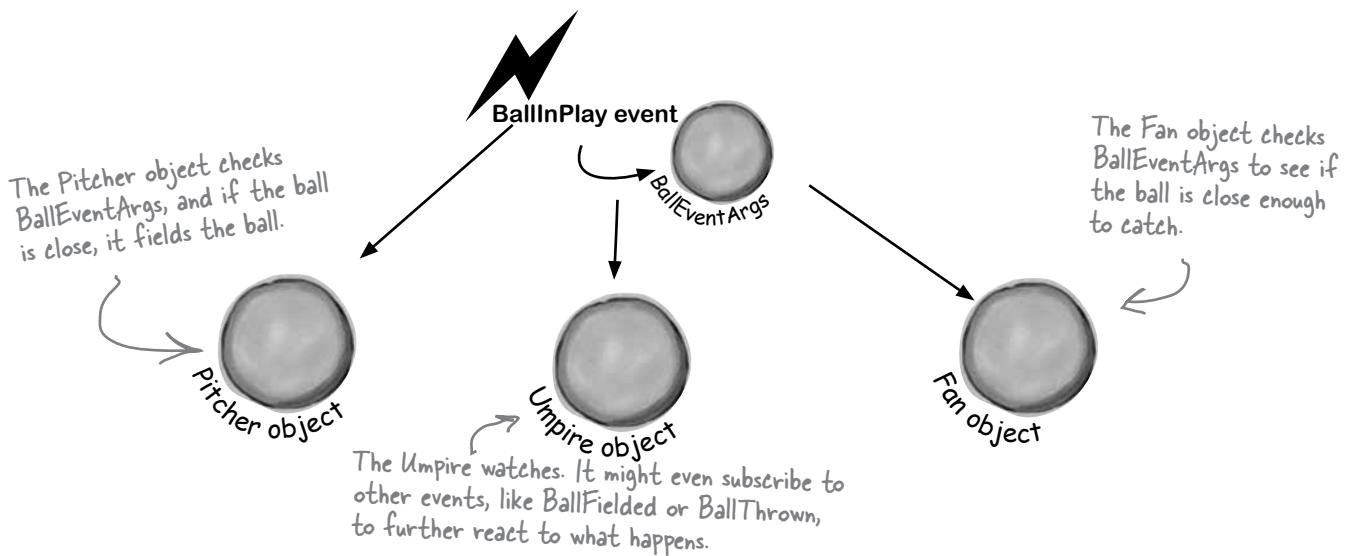
An event handler is a method in the subscriber object that gets run when the event is raised.

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.

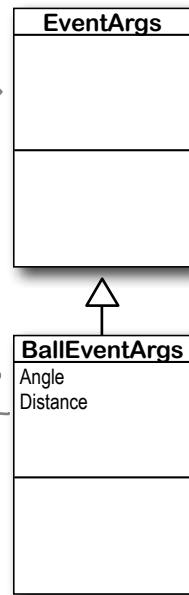
Now, Pitcher, Umpire, and Fan can all handle the BallInPlay event in their own way. But they don't all run at the same time—their event handlers get called one after another, with a reference to a BallEventArgs object as its parameter.



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.

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; }  
}
```

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

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 BallInPlayEventHandler(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 BallInPlayEventHandler 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(BallInPlayEventHandler);
```

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.

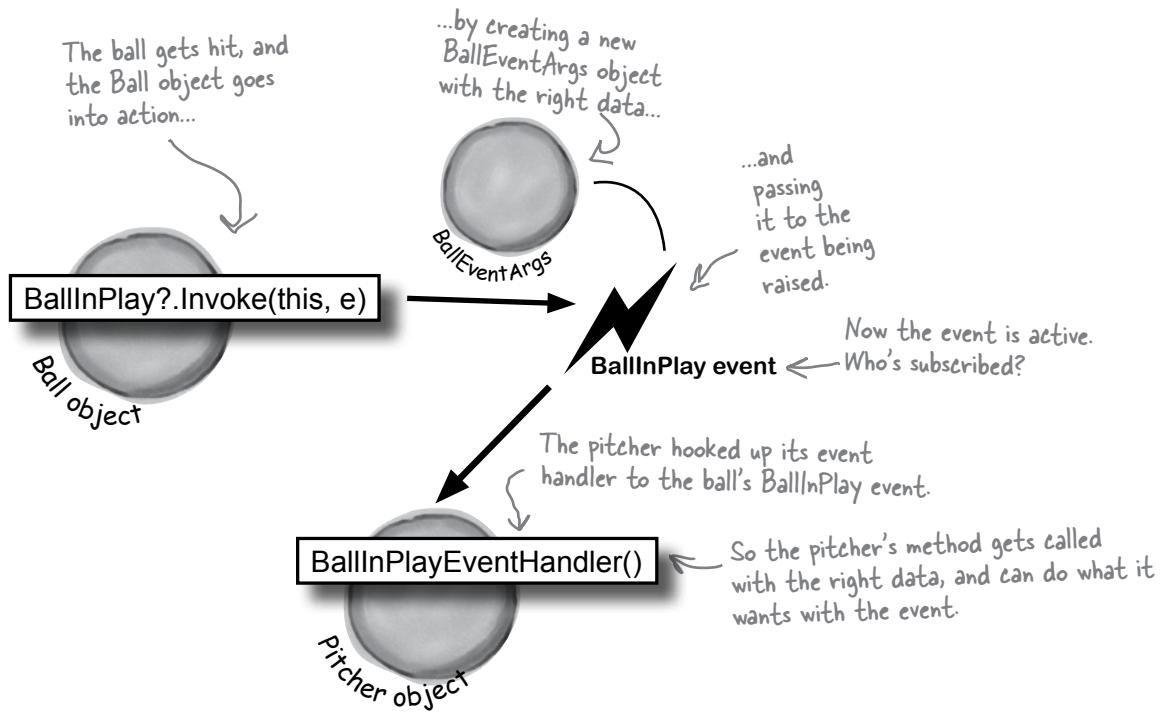
Turn the page; there's a little more... ➔

now your objects listen to each other

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 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 BallInPlayEventHandler. 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.)

Do this!

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

Start adding 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 += BallInPlayEventHandler;
```

When the IDE adds the event handler, it also pops up the Refactor Rename window so you can change its name. Name your new event handler method `BallInPlayEventHandler`.

**4**

The IDE will add your event handler, too.

You're not done—you still need to add a method to chain onto the event. Luckily, the IDE takes care of that for you, too. After the IDE finishes the statement, it shows you another box:

```
ball.BallInPlay += Ball_BallInPlay;
```

void Pitcher.Ball_BallInPlay(object sender, EventArgs e)

Hit the Tab key again to make the IDE add this event handler method to your `Pitcher` class. The IDE will always follow the `objectName_HandlerName` convention—but it also pops up its Rename window so you can rename your event handler. **Press Enter** to keep the default name:

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

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 BallInPlayEventHandler(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 classes:

- ★ The Program class with the Main method.
- ★ 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 += BallInPlayEventHandler;
```

2 The Main method 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 Main method just needs to call it:

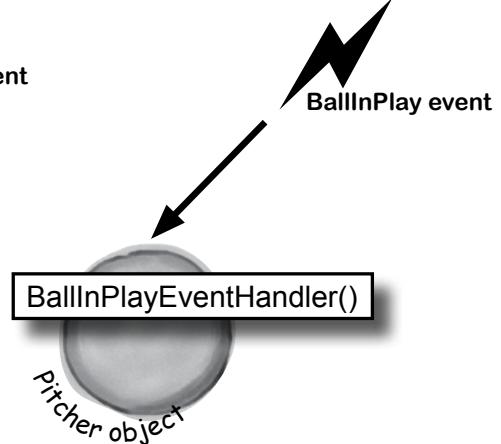
```
BallEventArgs ballEventArgs = new BallEventArgs(angle, distance);  
ball.OnBallInPlay(ballEventArgs);
```



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 BallInPlayEventHandler to the Ball.BallInPlay event, when the Ball raises its event the Pitcher's BallInPlayEventHandler method gets called **automatically**.



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 Main method.

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 input 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!
```

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

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

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

    void BallInPlayEventHandler(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 the Program class with the Main method. It uses static fields for references to the Ball, Pitcher, and Fan objects, and calls the Ball object's OnBallInPlay method to raise its BallInPlay event, which the Pitcher and Fan handle:

```
class Program
{
    static readonly Ball ball = new Ball();
    static readonly Pitcher pitcher = new Pitcher(ball);
    static readonly Fan fan = new Fan(ball);

    static void Main(string[] args)
    {
        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 Main method just has 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 Main method doesn't even need to know that's happening.

Generic EventHandlers let you define your own event types

We've seen EventArgs throughout the book. Go back to Chapter 1 and have a look at your Timer_Tick method declaration. Here's the WPF version:

```
private void Timer_Tick(object sender, EventArgs e)
```

And here's the ASP.NET Blazor version:

```
private void Timer_Tick(Object source, ElapsedEventArgs e)
```

And in Chapter 2, when you modified the text box to only accept numbers, your WPF app used an event handler method with a TextChangedEventArgs parameter, while your Blazor app used one with a ChangeEventArgs parameter.

Now take a look at the event declaration in your Ball class:

```
public event EventHandler BallInPlay;
```

That definitely works. 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. 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 BallInPlayEventHandler(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");
}
```

Run your app again. It should still work.

The generic argument to EventHandler has to be a subclass of EventArgs.



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.

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));
}
```

3 Add static methods to the Program class.

These are the methods we'll chain onto the event:

```
static int count;

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

static 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.

4 Update the Main method.

The Main method prompts the user to chain additional methods or raise the event.

```
static void Main(string[] args)
{
    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");
                myEvent.OnTalkToMe(line);
                break;
        }
    }
}
```

Now run your app. Try adding SaySomething 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.

XAML controls use routed events

This section only applies to WPF projects—but if you’re only using Blazor projects on macOS, it’s still worth reading to get the concepts.

Go back to the WPF app you built in Chapter 5 to calculate sword damage and look at its event handlers:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    RollDice();
}
```

You’ll see the same RoutedEventArgs parameter type in the radio button event handler in the WPF app in Chapter 2. These are event handlers for **routed events**. These are like normal events, except for one difference: when a control object responds to a routed event, first it fires off the event handler method as usual. Then it does something else: if the event hasn’t been handled, it **sends the routed event up to its container**. The container fires the event, and then if it isn’t handled, it sends the routed event up to its container. The event keeps **bubbling up** until it’s either handled or it hits the **root**, or the container at the very top. Here’s a typical routed event handler method signature.

private void EventHandler(object sender, RoutedEventArgs e)

The RoutedEventArgs object has a property called **Handled** that the event handler can use to indicate that it’s handled the event. Setting this property to true **stops the event from bubbling up**.

In both routed and standard events, the sender parameter always contains a reference to the object that called the event handler. So if an event is bubbled up from a control to a container like a Grid, then when the Grid calls its event handler, sender will be a reference to the Grid control. But what if you want to find out which control fired the original event? No problem. The RoutedEventArgs object has a property called **OriginalSource** that contains a reference to the control that initially fired the event. If **OriginalSource** and **sender** point to the same object, then the control that called the event handler is the same control that originated the event and started it bubbling up.

IsHitTestVisible determines if an element is “visible” to the pointer or mouse

Typically, any element on the page can be “hit” by the pointer or mouse—as long as it meets certain criteria. It needs to be visible (which you can change with the **Visibility** property), it has to have a **Background** or **Fill** property that’s not null (but can be **Transparent**), it must be enabled (with the **Enabled** property), and it has to have a height and width greater than zero. If all of these things are true, then the **IsHitTestVisible property will return True**, and that will cause it to respond to pointer or mouse events.

This property is especially useful if you want to make your events “invisible” to the mouse. If you set **IsHitTestVisible** to **False**, then any pointer taps or mouse clicks will **pass right through the control**. If there’s another control below it, that control will get the event instead.

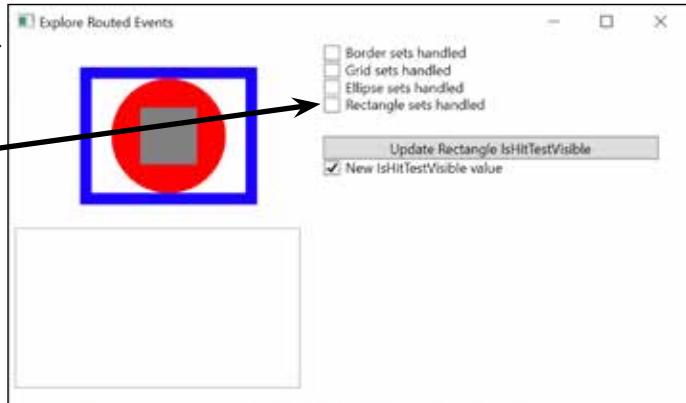
You can see a list of input events that are routed events here:
<http://msdn.microsoft.com/en-us/library/windows/apps/Hh758286.aspx>

The structure of controls that contain other controls that in turn contain yet more controls is called an object tree, and routed events bubble up the tree from child to parent until they hit the root element at the top.

Create an app to explore routed events

Here's a WPF application that you can use to experiment with routed events. It's got a StackPanel that contains a Border, which contains a Grid, and inside that grid are an Ellipse and a Rectangle. Have a look at the screenshot. See how the Rectangle is on top of the Ellipse? If you put two controls into the same cell, they'll stack on top of each other. But both of those controls have the same parent: the Grid, whose parent is the Border, and the Border's parent is the StackPanel. Routed events from the Rectangle or Ellipse bubble up through the parents to the root of the **object tree**.

You've already seen the CheckBox control, which you can use to toggle a value on and off. The Content property sets the label for the control. The IsChecked property is a Nullable<bool> because in addition to on and off, it can also have a third indeterminate state



```

<Grid Margin="5">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <StackPanel x:Name="panel" MouseDown="StackPanel_MouseDown">
        <Border BorderThickness="10" BorderBrush="Blue" Width="155" x:Name="border">
            Margin="20" MouseDown="Border_MouseDown">
                <Grid x:Name="grid" MouseDown="Grid_MouseDown">
                    <Ellipse Fill="Red" Width="100" Height="100"
                        MouseDown="Ellipse_MouseDown"/>
                    <Rectangle Fill="Gray" Width="50" Height="50"
                        MouseDown="Rectangle_MouseDown" x:Name="grayRectangle"/>
                </Grid>
            </Border>
            <ListBox BorderThickness="1" Width="250" Height="140" x:Name="output" Margin="0,0,20,0"/>
        </StackPanel>
        <StackPanel Grid.Column="1">
            <CheckBox Content="Border sets handled" x:Name="borderSetsHandled"/>
            <CheckBox Content="Grid sets handled" x:Name="gridSetsHandled" />
            <CheckBox Content="Ellipse sets handled" x:Name="ellipseSetsHandled"/>
            <CheckBox Content="Rectangle sets handled" x:Name="rectangleSetsHandled"/>
            <Button Content="Update Rectangle IsHitTestVisible"
                Click="UpdateHitTestButton" Margin="0,20,20,0"/>
            <CheckBox IsChecked="True" Content="New IsHitTestVisible value"
                x:Name="newHitTestVisibleValue" />
        </StackPanel>
    </Grid>

```

Routed events bubble up the object tree.

IsChecked defaults to False. This CheckBox has its set to True because controls always have IsHitTestVisible set to true by default.

The Ellipse and Rectangle controls draw shapes in your WPF window. Use the Width and Height properties to specify their size. The Border control draws a border, with properties to set its total width, thickness, and color.

Flip the page to finish the app →

you are here ▶

19

You'll need this ObservableCollection to display output in the ListBox.

Make a field called outputItems and set the ListBox.ItemsSource property in the page constructor. Don't forget to add the using System.Collections.ObjectModel; statement for ObservableCollection<T>.

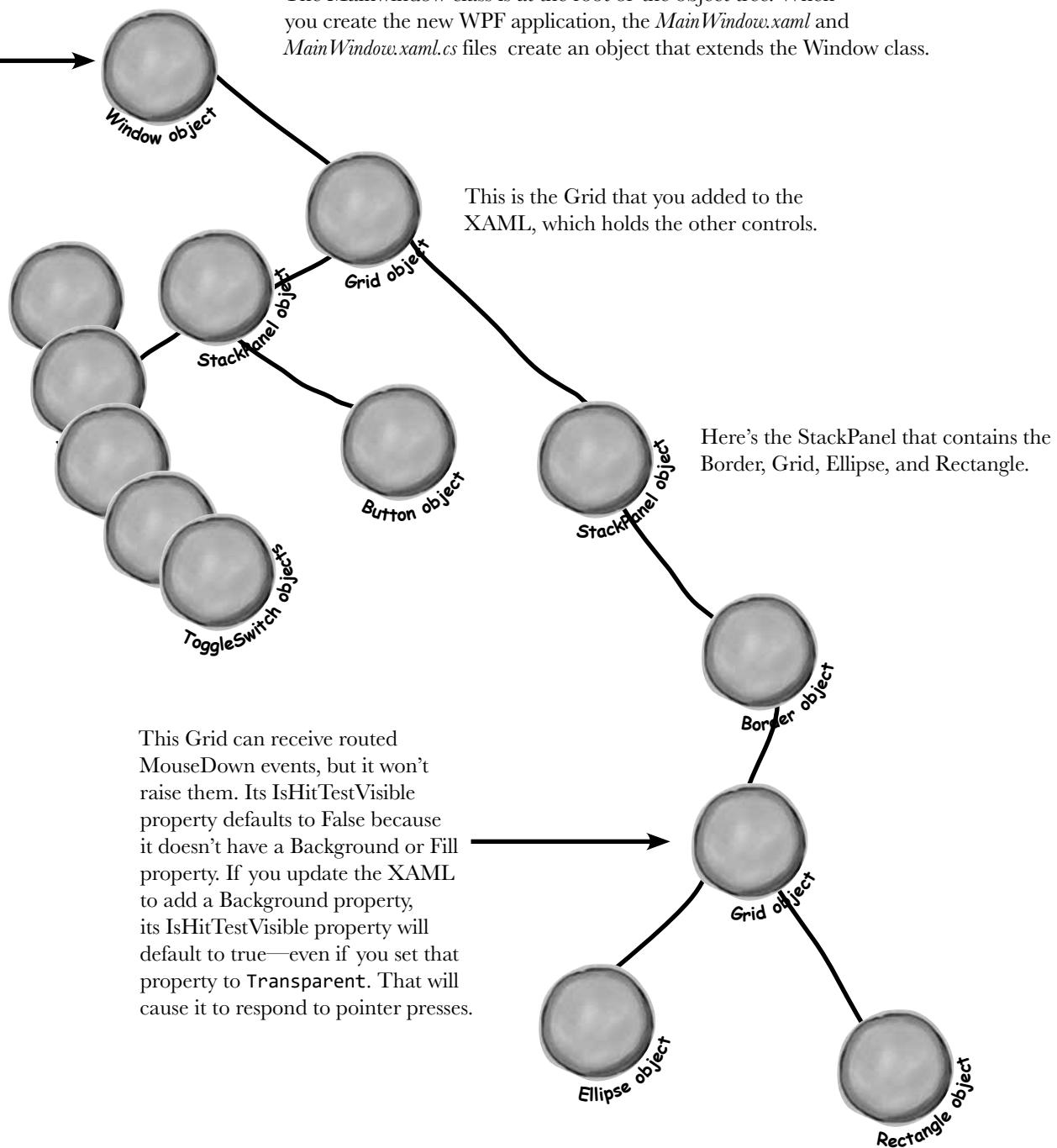
```
public partial class MainWindow : Window { ←  
    ObservableCollection<string> outputItems = new ObservableCollection<string>();  
  
    public MainWindow() {  
        InitializeComponent();  
  
        output.ItemsSource = outputItems;  
    }
```

Here's the code-behind. Each control's MouseDown event handler clears the output if it's the original source, and then it adds a string to the output. If its "handled" toggle switch is on, it uses e.Handled to handle the event.

```
private void Ellipse_MouseDown(object sender, MouseButtonEventArgs e) {  
    if (sender == e.OriginalSource) outputItems.Clear();  
    outputItems.Add("The ellipse was pressed");  
    if (ellipseSetsHandled.IsChecked == true) e.Handled = true;  
}  
  
private void Rectangle_MouseDown(object sender, MouseButtonEventArgs e) {  
    if (sender == e.OriginalSource) outputItems.Clear();  
    outputItems.Add("The rectangle was pressed");  
    if (rectangleSetsHandled.IsChecked == true) e.Handled = true;  
}  
  
private void Grid_MouseDown(object sender, MouseButtonEventArgs e) {  
    if (sender == e.OriginalSource) outputItems.Clear();  
    outputItems.Add("The grid was pressed");  
    if (gridSetsHandled.IsChecked == true) e.Handled = true;  
}  
  
private void Border_MouseDown(object sender, MouseButtonEventArgs e) {  
    if (sender == e.OriginalSource) outputItems.Clear();  
    outputItems.Add("The border was pressed");  
    if (borderSetsHandled.IsChecked == true) e.Handled = true;  
}  
  
private void StackPanel_MouseDown(object sender, MouseButtonEventArgs e) {  
    if (sender == e.OriginalSource) outputItems.Clear();  
    outputItems.Add("The panel was pressed");  
}  
  
private void UpdateHitTestButton(object sender, RoutedEventArgs e) {  
    grayRectangle.IsHitTestVisible = (bool)newHitTestVisibleValue.IsChecked;  
}  
  
The Click event handler for the button uses the IsOn  
property of the toggle switch to turn IsHitTestVisible  
on or off for the Rectangle control. →
```

Here's the object graph for your main window.

The Mainwindow class is at the root of the object tree. When you create the new WPF application, the *MainWindow.xaml* and *MainWindow.xaml.cs* files create an object that extends the Window class.



Run the app and click or tap the gray Rectangle.

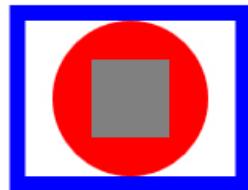
You should see the output in the screenshot to the right. →

You can see exactly what's going on by putting a breakpoint on the first line of `Rectangle_MouseDown`, the Rectangle control's `MouseDown` event handler:

```
private void Rectangle_MouseDown(object sender, MouseButtonEventArgs e)
{
    if (sender == e.OriginalSource) outputItems.Clear();
    outputItems.Add("The rectangle was pressed");
    if (rectangleSetsHandled.IsChecked == true) e.Handled = true;
}
```

Click the gray rectangle again—this time the breakpoint should fire. Use Step Over (F10) to **step through the code line by line**. First you'll see the if block execute to clear the `outputItems` `ObservableCollection` that's bound to the `ListBox`. This happens because `sender` and `e.OriginalSource` reference the same Rectangle control, which is true only inside the event handler method for the control that originated the event (in this case, the control that you clicked or tapped), so `sender == e.OriginalSource` is true.

When you get to the end of the method, **keep stepping through the program**. The event will bubble up through the object tree, first running the Rectangle's event handler, then the Grid's event handler, then the Border's, then the Panel's, and finally it runs an event handler method that's part of `LayoutAwarePage`—this is outside of your code and not part of the routed event, so it will always run. Since none of those controls is the original source for the event, none of their senders will be the same as `e.OriginalSource`, so none of them clear the output.



The rectangle was pressed
The grid was pressed
The border was pressed
The panel was pressed

Turn `IsHitTestVisible` off, press the “Update” button, and then click or tap the rectangle.

← You should see this output.
 Update Rectangle `IsHitTestVisible`
 New `IsHitTestVisible` value

The ellipse was pressed
The grid was pressed
The border was pressed
The panel was pressed

Wait a minute! You pressed the Rectangle, but the Ellipse control's `MouseDown` event handler fired. What's going on?

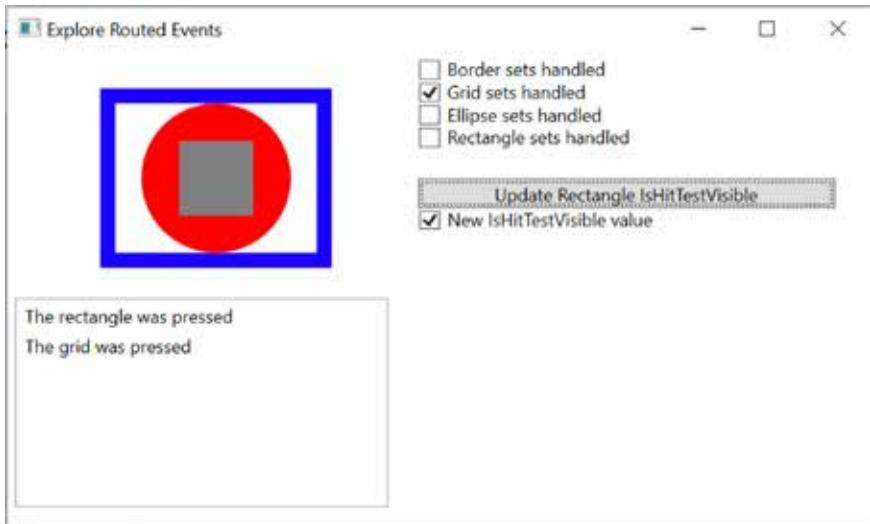
When you pressed the button, its Click event handler updated the Rectangle control's `IsHitTestVisible` property to false, which made it “invisible” to pointer presses, clicks, and other pointer events. So when you tapped the Rectangle, your tap passed right through it to the topmost control underneath it on the page that has `IsHitTestVisible` set to true and has a `Background` property that's set to a color or `Transparent`. In this case, it finds the Ellipse control and fires its `MouseDown` event.

Check the “Grid sets handled” box and click or tap the gray Rectangle.

You should see this output. →

So why did only two lines get added to the output ListBox?

Step through the code again to see what's going on. This time, gridSetsHandled.IsOn was true because you toggled the gridSetsHandled to On, so the last line in **the Grid's event handler set e.Handled to true**. As soon as a routed event handler method does that, **the event stops bubbling up**. As soon as the Grid's event handler completes, the app sees that the event has been handled, so it doesn't call the Border or Panel's event handler method, and instead skips to the event handler method in LayoutAwarePage that's outside of the code you added.



Use the app to experiment with routed events.

Here are a few things to try:

- ★ Click on the gray Rectangle and the red Ellipse and watch the output to see how the events bubble up.
- ★ Turn on each of the toggle switches, starting at the top, to cause the event handlers to set e.Handled to true. Watch the events stop bubbling when they're handled.
- ★ Set breakpoints and debug through all of the event handler methods.
- ★ Try setting a breakpoint in the Ellipse's event handler method, and then turn the gray Rectangle's IsHitTestVisible property on and off by toggling the bottom switch and pressing the button. Step through the code for the Rectangle when IsHitTestVisible is set to false.
- ★ Stop the program and add a Background property to the Grid to make it visible to pointer hits.

A routed event first fires the event handler for the control that originated the event, and then bubbles up through the control hierarchy until it hits the top—or an event handler sets e.Handled to true.

eventhandler is a delegate



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.

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)** on Windows or **Go to Declaration (⌘D)** on macOS 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 that has no event data.</summary>
/// <param name="sender">The source of the event.</param>
/// <param name="e">An object that contains no event data.</param>
public delegate void EventHandler (object? sender, EventArgs e);
```

Here's how a delegate works:

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.

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 = AddNumberSign;
```

AddNumberSign



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.** Go to the Program class and add this line:

```
class Program
{
    delegate string IntToString(int i);

    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```

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 to reference 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:

```
class Program
{
    delegate string IntToString(int i);

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

AddNumberSign

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

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

methodRef

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

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

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));
}
```

PlusOne

methodRef

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

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.

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

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

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 a Main method that prompts the user for a chef or an amount.

Here's the code for the Main method:

```
static void Main(string[] args)
{
    Adrian adrian = new Adrian();
    Harper harper = new Harper();

    GetSecretIngredient addIngredientMethod = null;

    while (true)
    {
        Console.WriteLine("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 MySecretIngredientMethod 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 AddAdriensSecretIngredient(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. The Main method is calling 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:

```
using System;
using System.Linq;
using System.Collections.Generic;

namespace ExploreFuncAndAction
{
    class Program
    {
        static void Main(string[] args)
        {
            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 body of your Main** method with these lines:

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

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

Then **click on Func** and use Go to Definition (F12) on Windows or Go to Declaration (⌘D) on macOS:

```
/// <summary>Encapsulates a method that has one parameter and returns a value of the type specified
/// by the <typeparamref name="TResult" /> parameter.</summary>
/// <param name="arg">The parameter of the method that this delegate encapsulates.</param>
/// <typeparam name="T">The type of the parameter of the method that this delegate encapsulates.</typeparam>
/// <typeparam name="TResult">The type of the return value of the method that this delegate encapsulates.</typeparam>
/// <returns>The return value of the method that this delegate encapsulates.</returns>
public delegate TResult Func<in T, out TResult>(T arg);
```

Func is a delegate with one 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.

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

```
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);
```

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

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.

Poo! 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

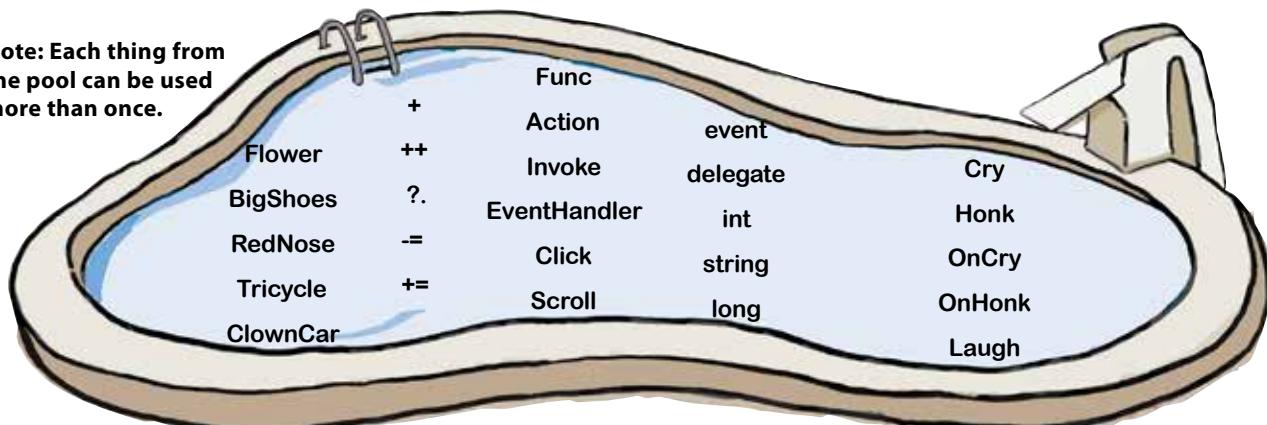
Fingers is coming to get you!

```
class RedNose
{
    public event _____ <string> _____;
    public void _____(string noise, string fun) =>
        Honk _____ _____ (this, $"Fin{noise} {fun}");
}

class Program
{
    static void Main(string[] args)
    {
        _____ <_____, ____> 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"));
    }
}
```

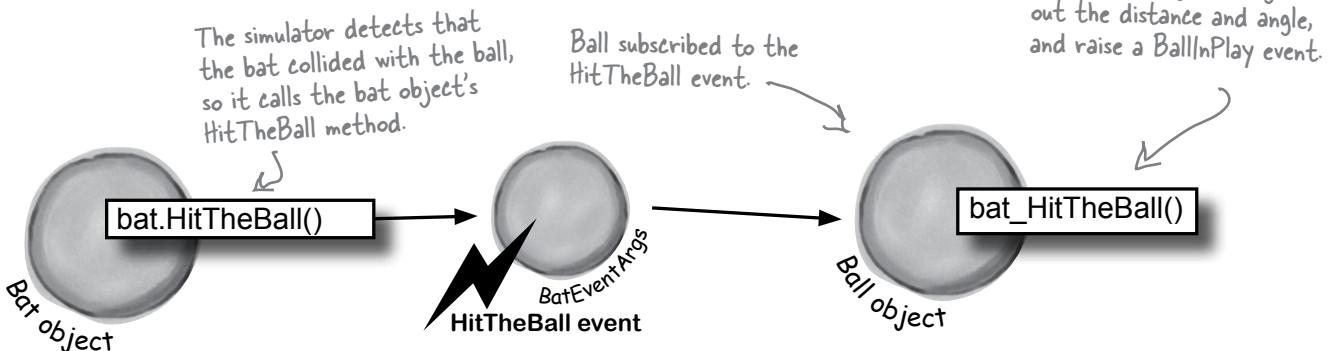
Note: Each thing from the pool can be used more than once.



An object can subscribe to an event...

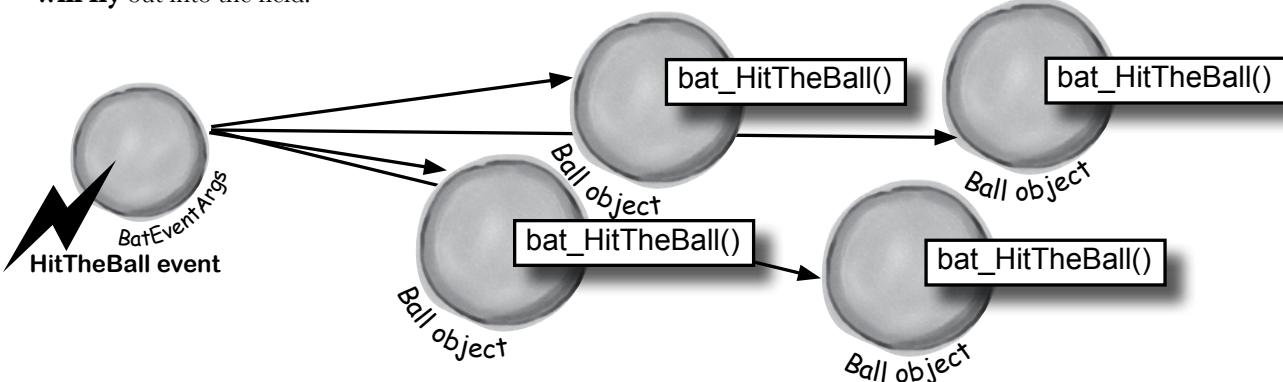
Suppose we add a new class to our simulator, a Bat class, and that class adds a HitTheBall event into the mix. Here's how it works: if the simulator detects that the player hit the ball, it calls the Bat object's HitTheBall method, which raises a HitTheBall event.

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!



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.

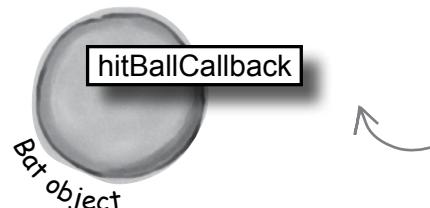
That's where a **callback** comes in handy. It's a technique that you can use with delegates. Instead of exposing an event that anyone can subscribe to, an object uses a method (often a constructor) that takes a delegate as an argument and holds onto that delegate in a private field. We'll use a callback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

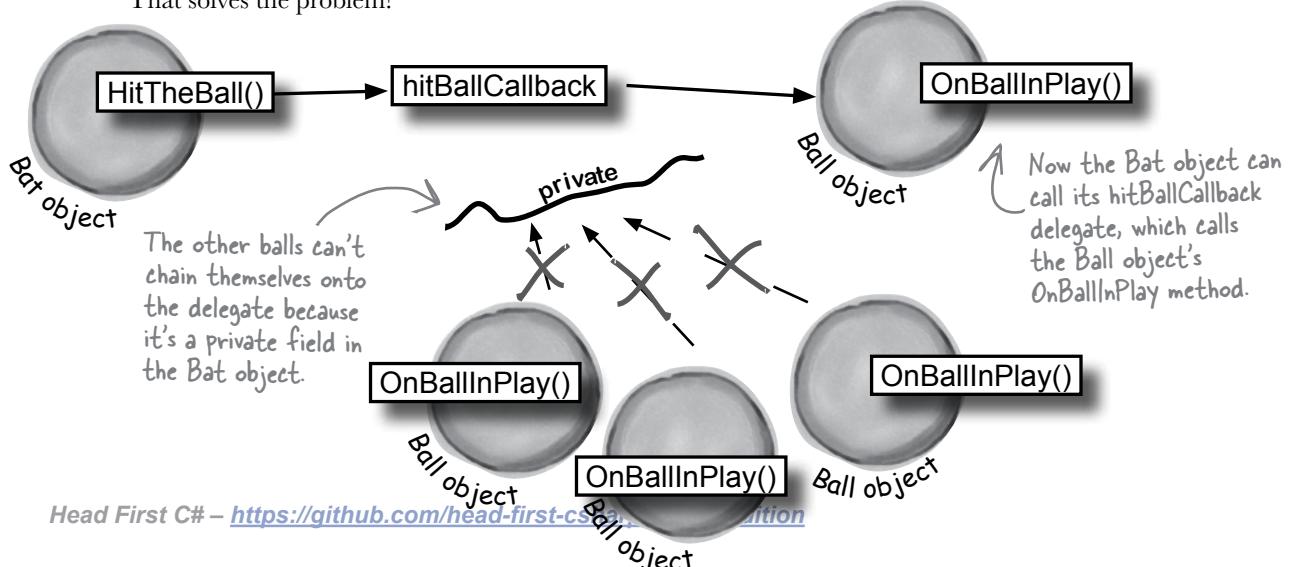
When the ball is in play, it creates the new instance of the bat, and it passes the Bat object a pointer to its OnBallInPlay method. This is called a **callback method** because the Bat is using it to call back to the object that instantiated it.



The Bat object passes a delegate reference to its own OnBallInPlay method to the Bat's constructor. The bat saves that delegate in its private hitBallCallback field.

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!



Five Minute Mystery



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 36

```
class RedNose
{
    public event EventHandler<string> Honk;
    public void OnHonk(string noise, string fun) =>
        Honk?.Invoke(this, $"Fin{noise} {fun}");
}

class Program
{
    static void Main(string[] args)
    {
        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"));
    }
}
```

Pool Puzzle Solution



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!”

Do this

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.WriteLine("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.

There's another design pattern that's really useful for building apps: it's called the MVVM pattern. You can learn more about it in the MVVM PDF, which you can download from the Head First C# GitHub page: <https://github.com/head-first-csharp/fourth-edition>

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.

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.

*there are no
Dumb Questions*

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!