O'REILLY®

~~Fourth~~ Fifth Edition

Head First C#

This is an early release preview of the first four chapters of Head First C# (5th edition) by Andrew Stellman and Jenny Greene. We'll release the final version of this PDF when the book is published in late 2023.

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

Andrew Stellman
& Jennifer Greene

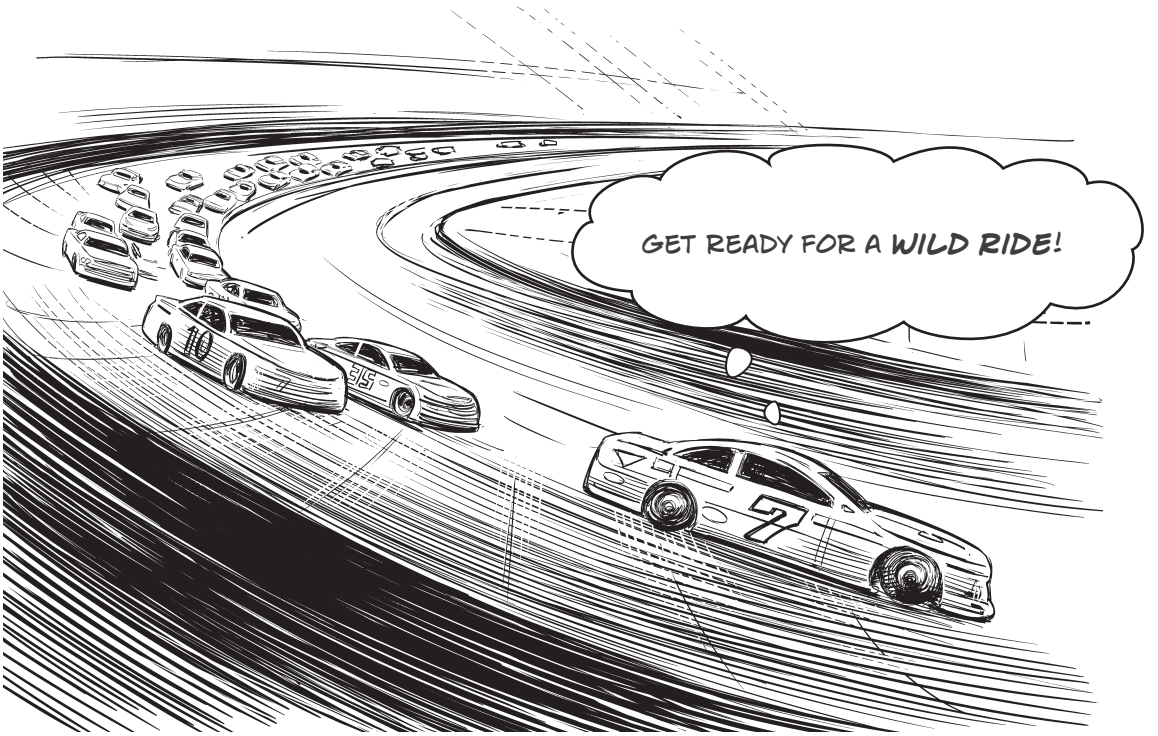
You can download this PDF, as well as early release versions of the .NET MAUI projects and Unity Labs from the 5th edition from our GitHub page.

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

A Brain-Friendly Guide

1 start building apps with C#

Build something great...fast!



Want to build great apps…right now?

With C#, you've got a modern programming language and a **valuable tool** at your fingertips. And with **Visual Studio**, you've got an amazing development environment with highly intuitive features that make coding as easy as possible. Not only is Visual Studio a great tool for writing code, it's also a **really effective learning tool** for exploring C#. Sound appealing? ***Let's get coding!***

Why you should learn C#

C# is a simple, modern language that lets you do incredible things. When you learn C#, you're learning more than just a language. C# unlocks the whole world of .NET, an incredibly powerful open source platform for building all sorts of applications.

So why should you learn C#? There are ***so many reasons***! Here are just a few of them:

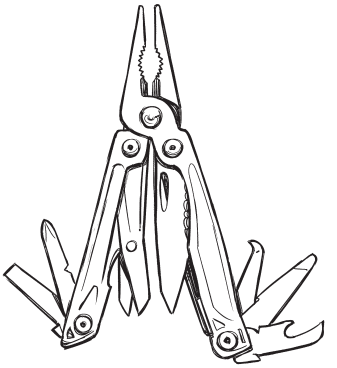
- **C# is really popular… and for good reason.** Microsoft released the first version of C# over 20 years ago, and the C# team been working on it since then—and it shows! They've kept it up to date and added lots of new features, which is why it's one of the most flexible and powerful programming languages in the world.
- **You can build many, many, *many* kinds of apps in C#.** C# is really versatile. You can use it to build everything from games to financial and accounting systems to websites to… well, you name it, C# can do it.
- **.NET—the framework that powers most C# apps—is cross-platform and open source.** In fact, all of the apps that you'll build in this book will run on Windows and macOS, and many of the apps can even be deployed to Android and iOS mobile devices.
- **C# skills are *in demand*.** Are you looking to land a programming job? C# is one of the most in-demand programming languages around, because companies all over the world use C# to build their desktop applications and websites.

Write code and explore C# with Visual Studio

The best way to get started with C# is to **write lots of code**.

This book uses **pictures, puzzles, quizzes, stories, and games** to help you learn C# in a way that suits your brain. Every one of those elements is built to help you with a single goal: to keep things interesting while we help you get C# concepts, ideas, and skills into your brain.

This book is also **full of C# projects** that are ***specifically designed*** to give you lots of different ways to explore C# and learn about important ideas and concepts that will help you become a great developer. We designed those projects to be engaging, fun, and interactive to give you lots of opportunities to put those concepts, ideas, and skills into practice.



Visual Studio is your gateway to C#

Learning C# is all about exploring and growing your skills ***at your own pace***—and that's where **Visual Studio** comes in. It's amazing tool built by Microsoft. At its heart, it's an editor for your C# code and projects, but it's much more than that. It's a creative tool that helps you with every aspect of C# development. We'll use Visual Studio throughout this book as an important tool to to help you learn and explore C#.

Visual Studio is an **IDE**—that's short for ***integrated development environment***—is a text editor, visual designer, file manager, debugger…it's like a multitool for everything you need to write code.

Here are just a few of the things that Visual Studio helps you do:

- **It's a file and project manager.** C# projects are often made up of a lot of files. Visual Studio makes it easy to see exactly where they are, and integrates with version control systems like Git to make sure you never lose a line of code.
- **It helps you edit and manage your code.** Visual Studio has many intuitive features to help you edit your code and C# projects, including powerful AI-driven tools like IntelliSense pop-ups and IntellICode code completion that gives you great suggestions to help keep you in the flow.
- **It's a debugger that lets you see your code in action.** When you debug your apps in Visual Studio, you can see exactly what your code is doing while it runs—which is a great way to ***really understand*** how C# code works.

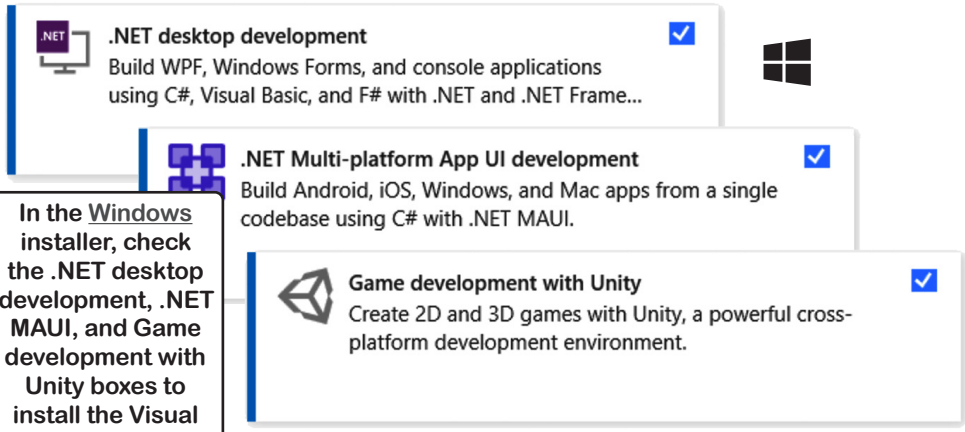
Visual Studio is an powerful development environment, and it's an amazing learning tool to help you explore C#.

IDE Tips

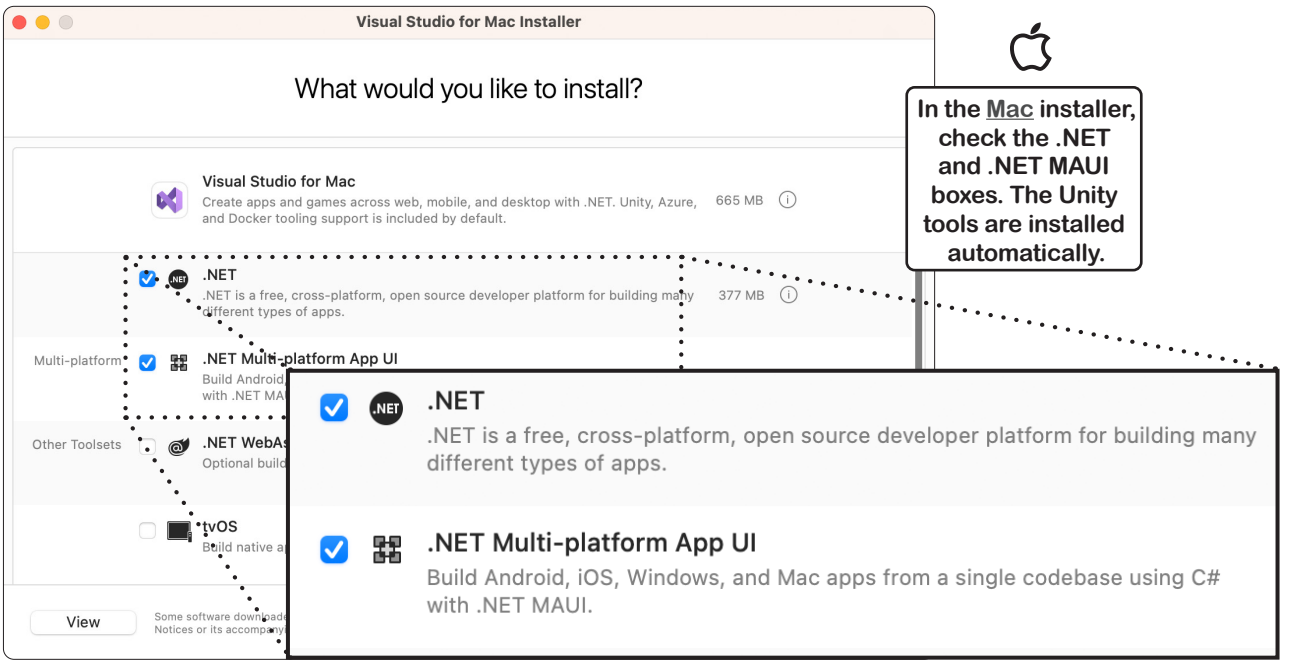
We'll often refer to Visual Studio as **"the IDE"** throughout this book. Keep an eye out for handy IDE tips that help you become a more efficient coder.

Install Visual Studio Community Edition

Open https://visualstudio.microsoft.com/ and **download Visual Studio Community Edition**. It's available for both Windows and macOS. The installers look a little different depending on which platform you're using. Make sue you install the .NET desktop development tools and .NET Multi-platform App UI (or .NET MAUI) development toolols. We'll be doing 3D game development with Unity, so make sure you check that option on Windows (it's installed automatically for Mac).



In the Windows installer, check the .NET desktop development, .NET MAUI, and Game development with Unity boxes to install the Visual Studio tools you'll use in this book.



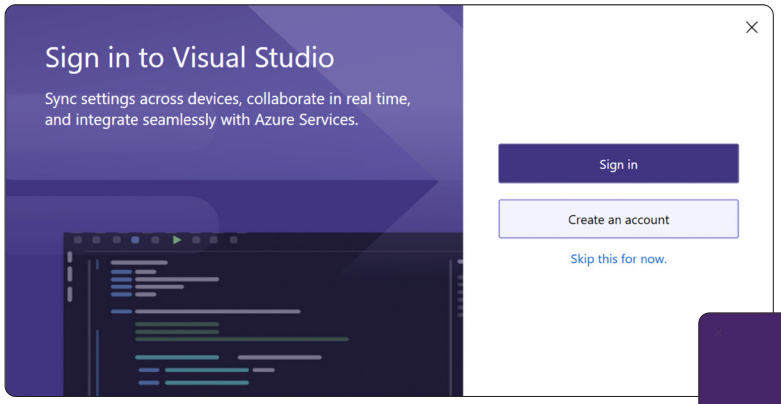
In the Mac installer, check the .NET and .NET MAUI boxes. The Unity tools are installed automatically.

Make sure you're installing Visual Studio, not Visual Studio Code.

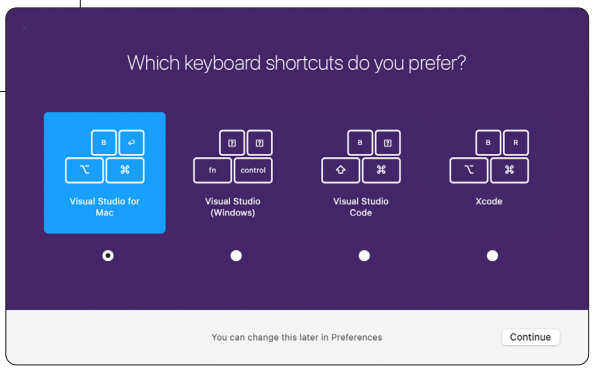
Visual Studio Code is an amazing open source, cross-platform code editor, but it's not tailored to .NET development the way Visual Studio is. That's why we can use Visual Studio throughout this book as a tool for learning and exploration.

Run Visual Studio

We're going to jump right into code! So once the installer finishes, **run Visual Studio**.

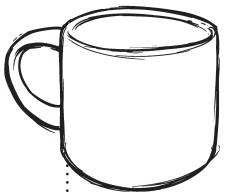


The first time you start Visual Studio, it may ask you to sign into your Microsoft account—make sure you do that, and if you don't have an account, you can create one. This is how Microsoft activates you your free license for the Community edition of Visual Studio.



Visual Studio for Mac will ask you which keyboard shortcuts you prefer. We recommend choosing the Mac shortcuts, but give you both the Windows and Mac shortcuts throughout this book. You can change this option later if you want.

Keep an eye out for these "relax" boxes—they point out some common issues that a lot of readers run into, so you know they're coming and don't have to worry about them.



Relax

Grab a cup of coffee—it can take some time for Visual Studio to install.

Don't worry if it takes a few minutes (or more!) to finish installing Visual Studio. And while we're on the subject, here's something else that you don't have to worry about.

All of the screenshots in this book were taken with **Visual Studio 2022 Community Edition**, *the latest version available while we're writing it. If Microsoft released a newer version of Visual Studio since we took these screenshots, feel free to try it! The code and ideas that we teach should still work just fine. But if you want the screenshots to match, Microsoft makes older versions of Visual Studio available for download—and you can always install different versions of Visual Studio on the same computer: https://visualstudio.microsoft.com/vs/older-downloads/*

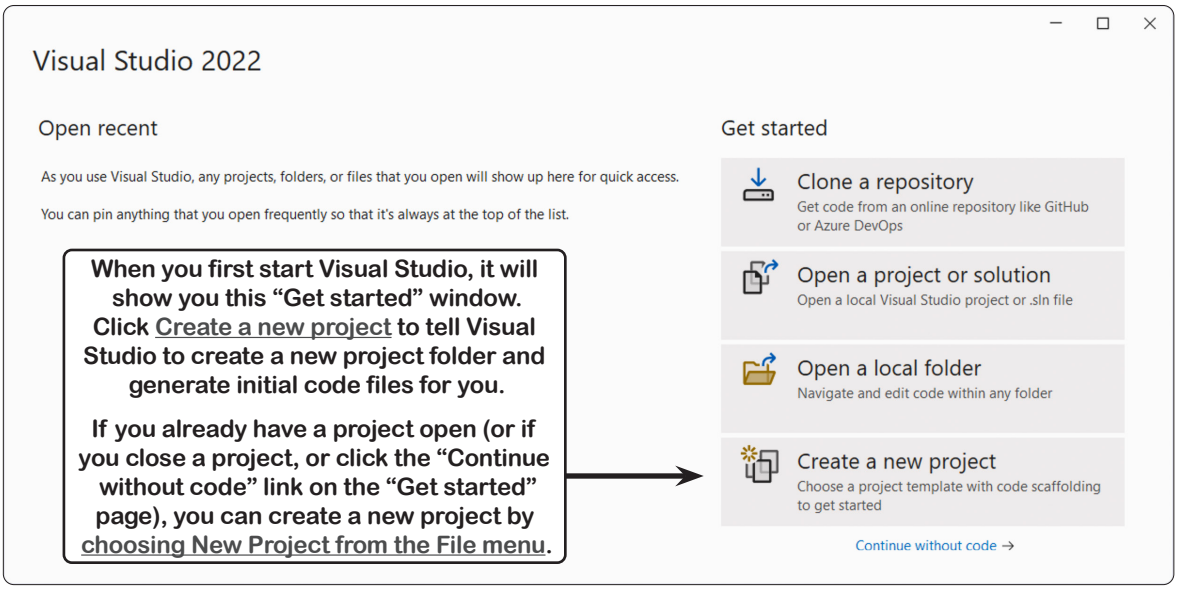
If you run into trouble installing Visual Studio, head to our YouTube channel to see videos of the entire installation process: https://www.youtube.com/@headfirstcsharp

Create your first project in Visual Studio – Windows edition

The best way to learn C# is to start writing code, so you're going to write a lot of code—and create a lot of apps!—throughout this book. Each app will get its own **project**, or a folder that Visual Studio creates with special files to organize all of the code.

1 Tell Visual Studio to create a new project.

When you launch Visual Studio, the first thing you'll see is a "Get started" window with four options. **Click "Create a new project"** to create a new project.



When you create a new project, Visual Studio will ask you which of its **project templates** you want to use. Every C# project consists of a set of folders and files. Visual Studio has many built-in templates that it can use to generate different kinds of projects. In this book, you'll use Visual Studio's templates to create two kinds of projects, Console App projects and .NET MAUI projects. (You'll also create Unity projects, but you won't use Visual Studio to create them.)

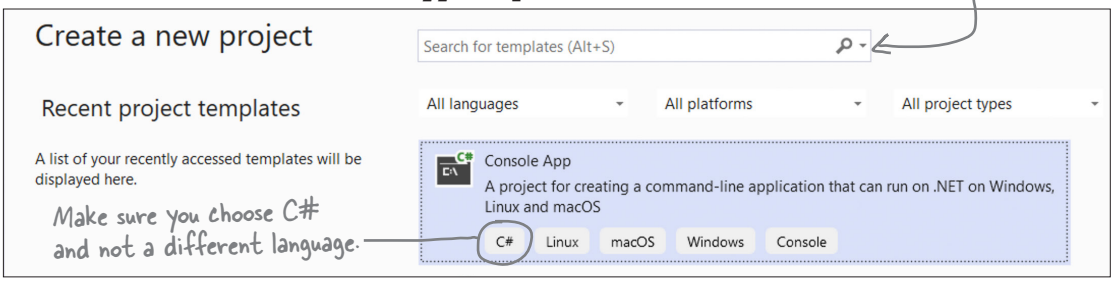
This applies to both Windows and Mac projects! We'll show you how to create and run an app in Visual Studio for Mac, too.

Geek Note

You'll be writing a lot of code throughout this book, which means you'll be **creating a lot of projects**. Most of those projects will be **Console App projects**, just like the one you're creating now—so you can follow these directions any time you need to create a new Console App project. Just make sure you choose a different project name each time, so that Visual Studio creates the project in a new folder (don't worry—it will warn you if that name already exists).

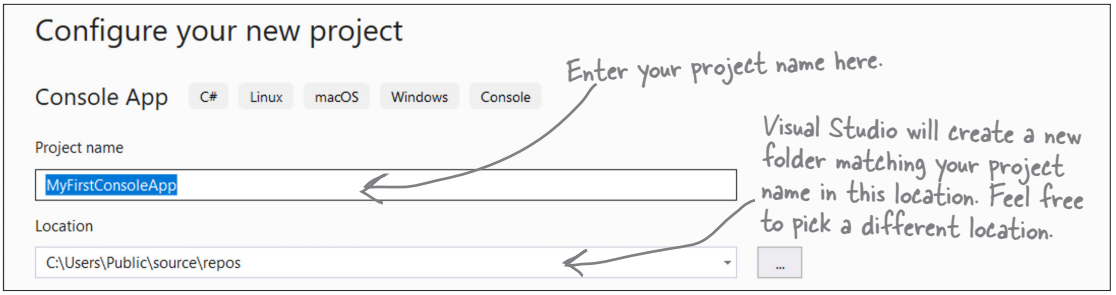
2 Choose a project template for Visual Studio to use.

Visual Studio creates new projects using a *template* that determines what files to create. **Choose the *Console App* template** and click Next.



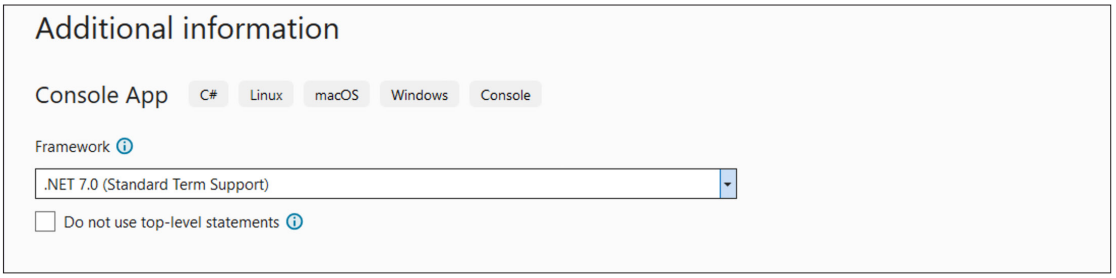
3 Enter a name for your project and click Next.

Your project's name is important—it determines file and folder names, and you'll see it inside some of the code that Visual Studio generates for you. If we ask you to pick a specific name, make sure you do, otherwise the code in your project may not match screenshots in the book.

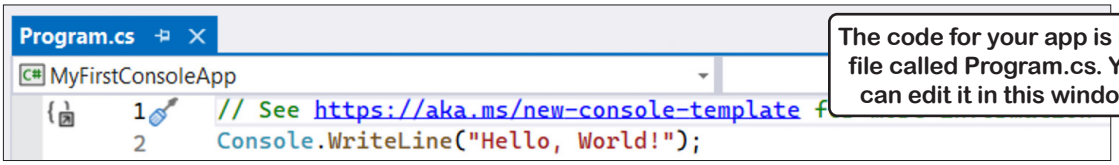


4 Make sure you're using the current version of .NET.

The current version of .NET at the time we're writing this is 7.0 – make sure the version that you're using is 7.0 or later. Then **click the Create button** to create your project.



Once Visual Studio creates your project, it will open a file called Program.cs with this code:

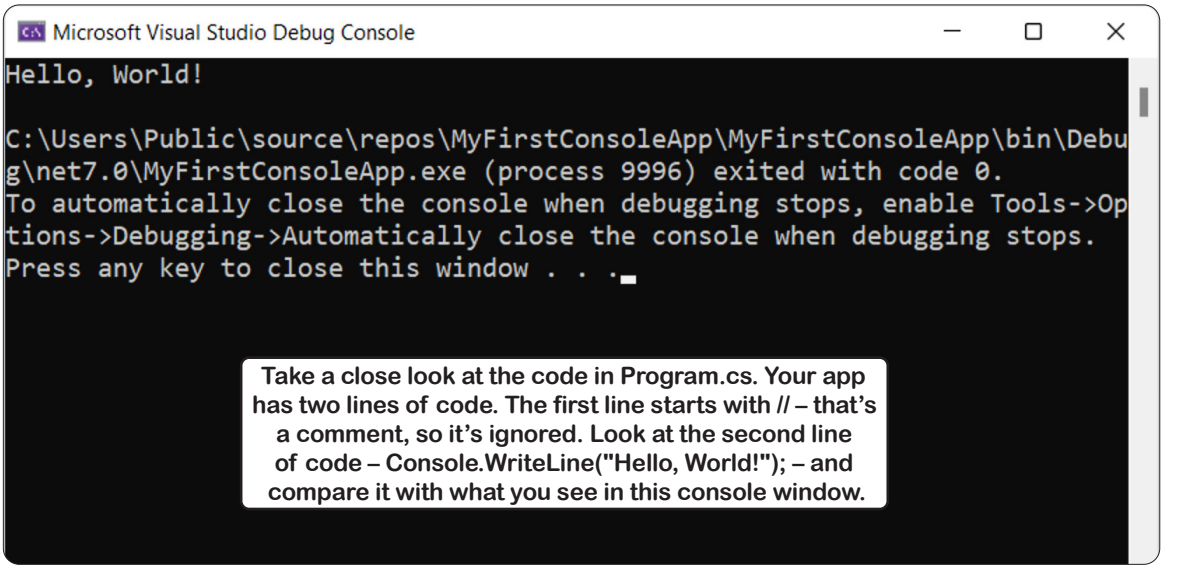


5 Run your app.

The app Visual Studio created for you is ready to run. At the top of the Visual Studio IDE, find the button with a green triangle and your app's name and click it:

6 Look at your app's output.

When you run your program, the **Microsoft Visual Studio Debug Console window** will pop up and show you the output of the program:



At the top of the window is the **output of the program**:

```
Hello World!
```

Then there's a line break, followed by some additional text:

```
C:\path-to-your-project-folder\MyFirstConsoleApp\MyFirstConsoleApp\bin\Debug\
net7.0\MyFirstConsoleApp.exe (process ####) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options-
>Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

You'll see the same message at the bottom of every Debug Console window. Your program printed a single line of text (`Hello World!`) and then exited. Visual Studio is keeping the output window open until you press a key to close it so you can see the output before the window disappears.

Press a key to close the window. Then run your program again. This is how you'll run all of the .NET Core Console App projects that you'll build throughout the book.

Keep an eye out for these Q&A sections. They often answer your most pressing questions, and point out questions other readers are thinking of. In fact, a lot of them are real questions from readers of previous editions of this book!

there are no Dumb Questions

Q: So if Visual Studio writes all this code for me, is learning C# just a matter of learning how to use it?

A: No. The IDE is great at automatically generating some code for you, but it can only do so much. There are some things it's really good at, like setting up good starting points for you, and automatically changing properties of controls in your UI. The most important part of programming—figuring out what your program needs to do and making it do it—is something that no IDE can do for you. Even though the Visual Studio IDE is one of the most advanced development environments out there, it can only go so far. It's ***you***—not the IDE—who writes the code that actually does the work.

Q: What if the IDE creates code I don't want in my project?

A: You can change or delete it. The IDE is set up to create code based on the way the element you dragged or added is most commonly used, but sometimes that's not exactly what you wanted. Everything the IDE does for you—every line of code it creates, every file it adds—can be changed, either manually by editing the files directly or through an easy-to-use interface in the IDE.

Q: Why did you ask me to install Visual Studio Community edition? Are you sure that I don't need to use one of the versions of Visual Studio that isn't free in order to do everything in this book?

A: There's nothing in this book that you can't do with the free version of Visual Studio (which you can download from Microsoft's website). The main differences between Community and the other editions aren't going to stop you from writing C# and creating fully functional, complete applications.

Q: My screen doesn't look like yours! It's missing some of the windows, and others are in the wrong place. Did I do something wrong? How can I reset it?

A: If you click on the **Reset Window Layout** command under the Window menu, the IDE will restore the default window layout for you. Then use the **View>>Other Windows** menu to open the any windows that are missing. That will make your screen look like the ones in this chapter and throughout the book.

Some windows collapse by default. Use the pushpin button in the upper-right corner of the window to make it stay open.

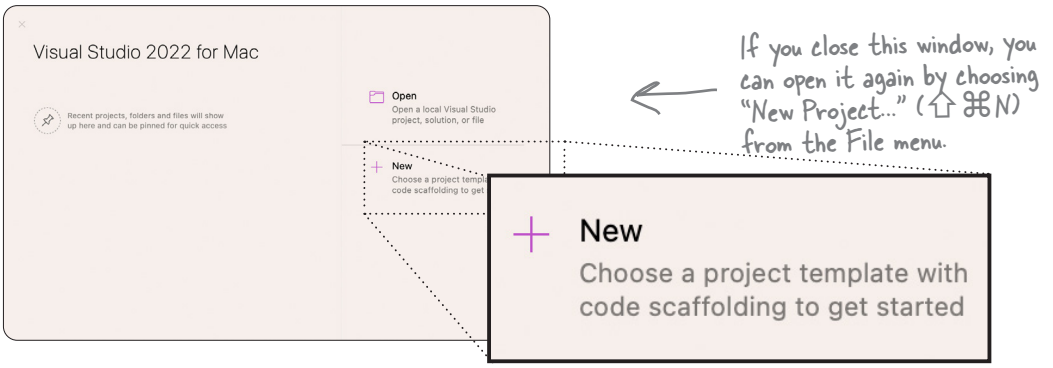
Visual Studio will generate code you can use as a starting point for your applications. Making sure the app does what it's supposed to do is entirely up to you.

Create your first project in Visual Studio – Mac edition

The best way to learn C# is to start writing code, so you're going to write a lot of code—and create a lot of apps!—throughout this book. Each app will get its own **project**, or a folder that Visual Studio creates with special files to organize all of the code. Follow these steps to create the Console App projects in this book.

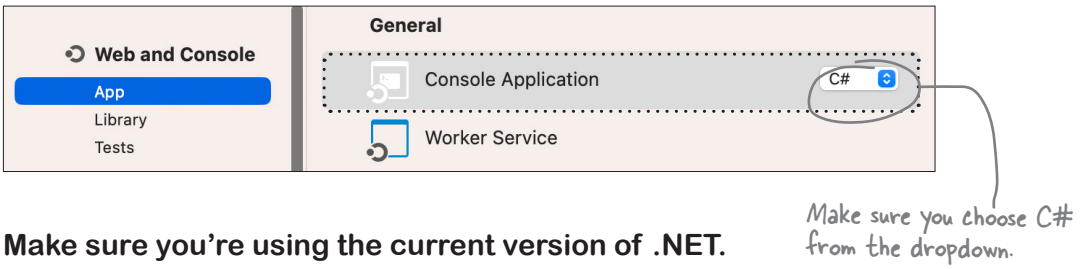
1 Start creating a new project a new project.

Start up Visual Studio for Mac. When it first starts up, it shows you a window with big buttons to open an existing project or create a new one. **Click + New** to start creating a new project.



2 Choose a project template for Visual Studio to use.

Visual Studio creates new projects using a *template* that determines what files to create. Click "App" in the "Web and Console" list and **choose the *Console App* template** and click Continue.



3 Make sure you're using the current version of .NET.

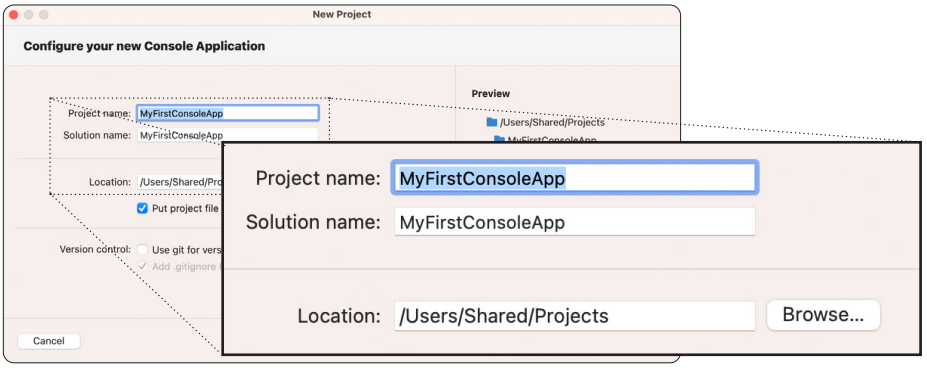
The current version of .NET at the time we're writing this is 7.0 – make sure the version that you're using is 7.0 or later. Click the Continue button.

If Visual Studio gives you an error about Xcode when you try to run your MAUI app, open the Xcode app on your Mac, choose Settings (⌘,) from the Xcode menu, click Locations, and select the highest option from the Command Line Tools dropdown. If it's blank, open Terminal and run this command: `xcode-select --install`

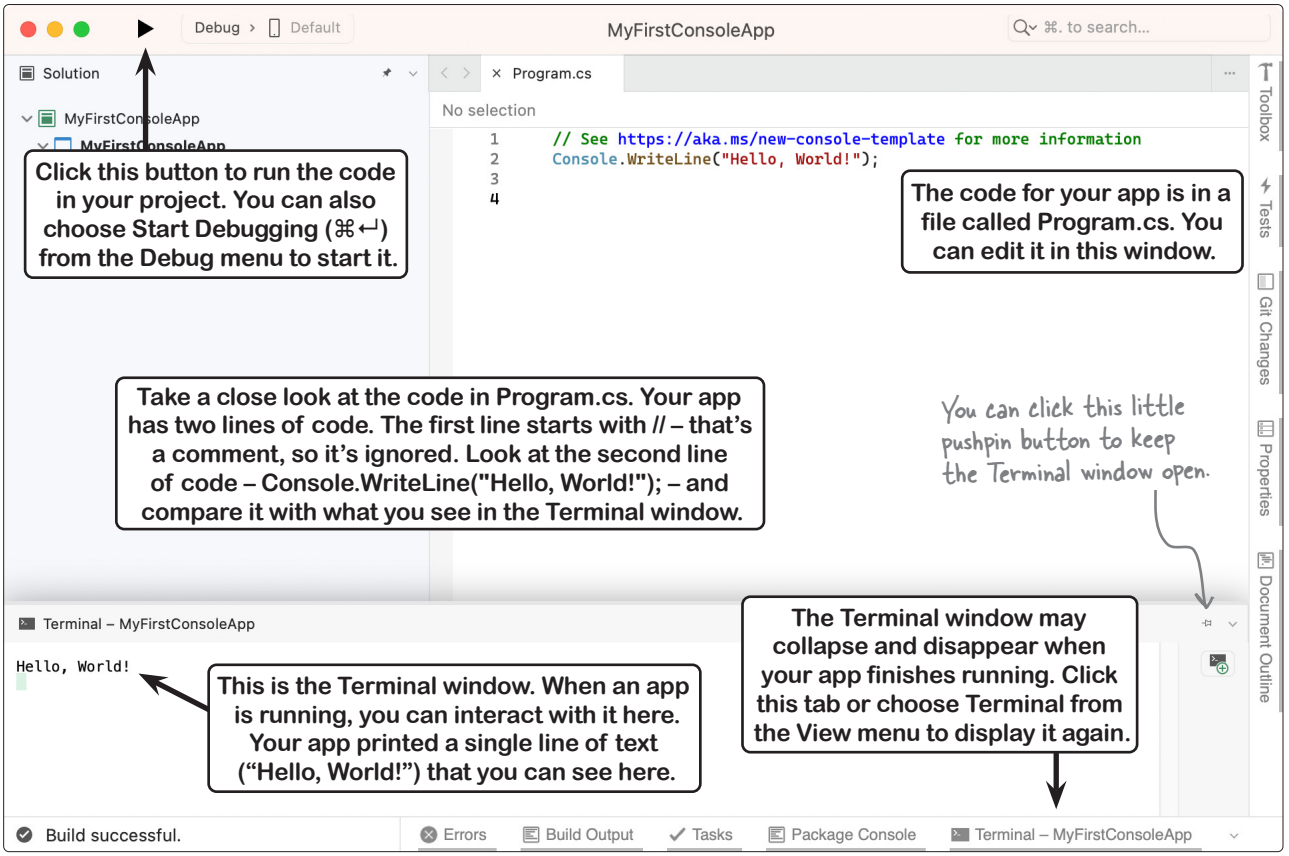
4

Enter a name for your project and click Create.

Your project's name is important—it determines file and folder names, and you'll see it inside some of the code that Visual Studio generates for you. If we ask you to pick a specific name, make sure you do, otherwise the code in your project may not match screenshots in the book.

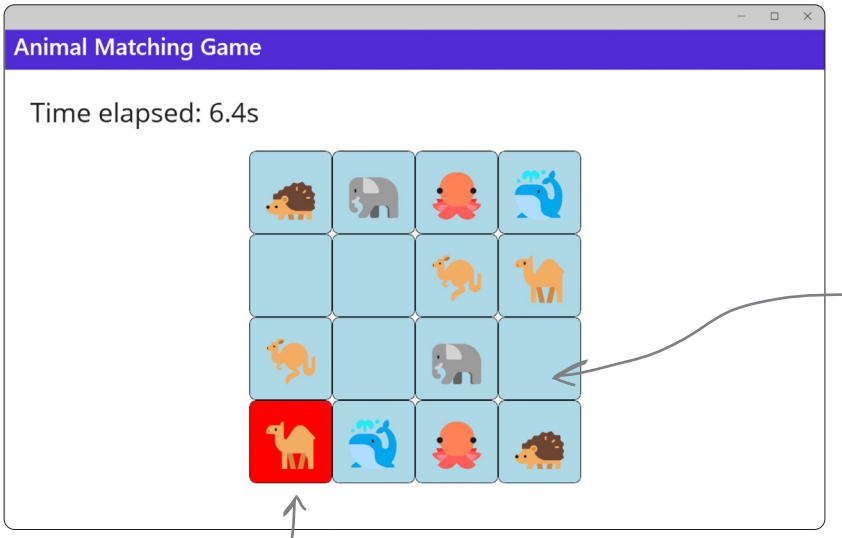


Once Visual Studio creates your project, open a file called Program.cs that has the code for your new app. You can **press the run button ▶ to run the app** and view the results in the Terminal window.



Let's build a game!

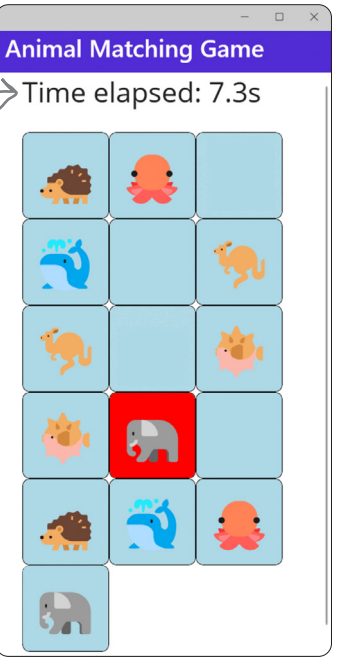
You've built your first C# app, and that's great! Now that you've done that, let's build something a little more complex. We're going to build an **animal matching game**, where a player is shown 8 pairs of animals and needs to click on them in pairs to make them disappear.



The game displays sixteen buttons with eight pairs of matching animals in a random order. You play by clicking animals in pairs: first click one animal, then click its match. Match all eight animals and you win the game!

When you click a matched pair of animals, they both disappear.

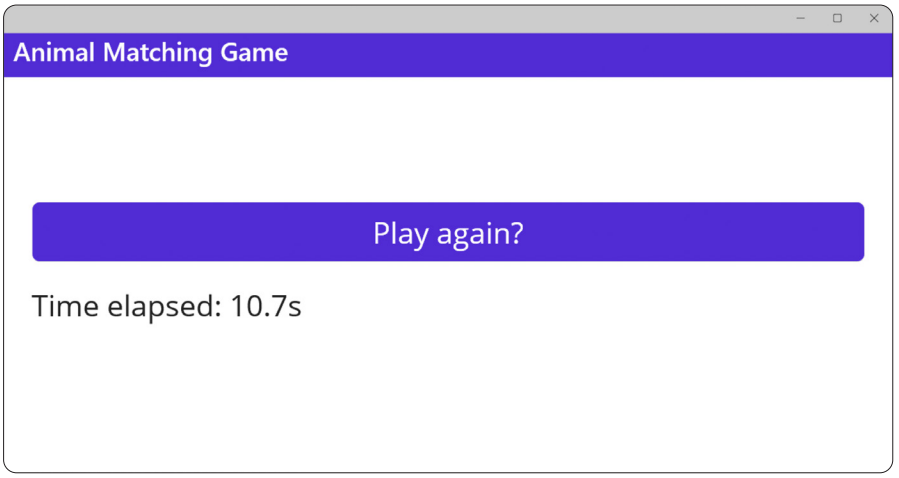
When you click the first button, it changes color. If you click on its match, then both animals disappear. If you click any other animal, the color of the first button changes back and you have to start over with a new pair.



To make things more exciting, the game starts a timer as soon as you start the game. Can you beat your best time?

You can change the size of the window and the animal buttons will rearrange themselves out to fill up the new width.

When you've found all eight pairs of animals, the game displays a big "Play again?" button, with your final time underneath it. Click the button to reset the game and start over again!



Keep an eye out for these "Game design...and beyond" elements scattered throughout the book. We'll use game design principles as a way to learn and explore important programming concepts and ideas that apply to any kind of project, not just video games.

Game design... and beyond

What is a game?

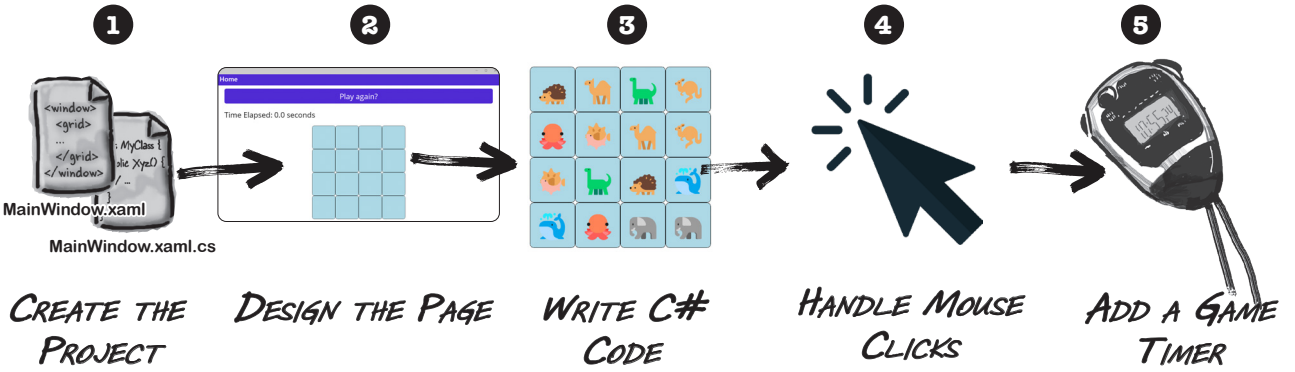
It may seem obvious what a game is. But think about it for a minute—it's not as simple as it seems.

- Do all games have a **winner**? Do they always end? Not necessarily. What about a flight simulator? A game where you design an amusement park? What about a game like The Sims?
- Are games always **fun**? Not for everyone. Some players like a "grind" where they do the same thing over and over again; others find that miserable.
- Is there always **decision making, conflict, or problem solving**? Not in all games. Walking simulators are games where the player just explores an environment, and there are often no puzzles or conflict at all.
- It's actually pretty hard to pin down exactly what a game is. If you read textbooks on game design, you'll find all sorts of competing definitions. So for our purposes, let's define the **meaning of "game"** like this:

A game is a program that lets you play with it in a way that (hopefully) is as entertaining to play as it is to make.

Break up large projects into smaller parts

Our goal in this book is to help you to learn C#, but we also help you **become a great developer**, and one of the most important skills great developers work on is tackling large projects. You'll build a lot of projects throughout this book. They'll be smaller starting with the next chapter, but they'll get bigger as you go further. As the projects get bigger, we'll show you how to break them up into smaller parts that you can work on one after another. This project is no exception—it's a larger project, like the ones you'll do later in the book—so you'll do it in five parts.



If you run into any trouble with this project, you can watch a full video walkthrough on our YouTube channel. https://www.youtube.com/@headfirstcsharp

You can download all of the code and a PDF of this chapter from our GitHub page: https://github.com/head-first-csharp/fifth-edition

Relax

This chapter is all about learning the basics, getting used to creating projects, editing code, and building your game.

Don't worry if there are things that you don't understand yet. By the end of the book, you'll understand everything that's going on in this game. For now, just follow the step-by-step instructions to get your game up and running. This will give you a solid foundation to build on later.

Here's how you'll build your game

The rest of this chapter will walk you through building your animal matching game. You'll build it using **.NET MAUI** (which stands for .NET Multi-platform App UI, or just **MAUI**). MAUI is a technology that you can use to create apps in C# that run natively as desktop apps on Windows and macOS, or as mobile apps on your Android or iOS mobile devices.

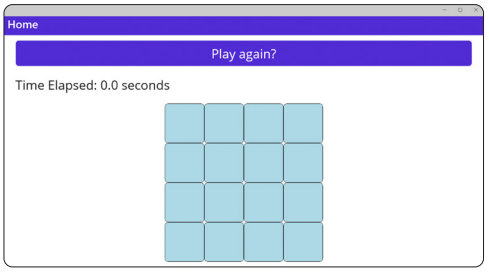
The rest of this chapter will walk you through building your animal matching game, and you'll be doing it in a series of separate parts:

This project can take anywhere from 20 minutes to over an hour, depending on how quickly you type. We learn better when we don't feel rushed, so give yourself plenty of time.

MainWindow.xaml

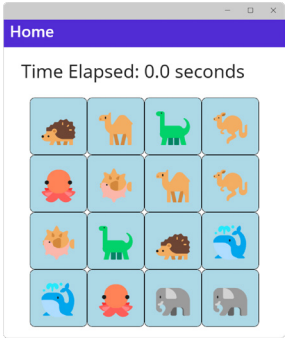
MainWindow.xaml.cs

1. **First you'll create a new .NET MAUI project in Visual Studio.**
   You just created a new console aplication. Now you'll create a new MAUI app.



2. **Then you'll use XAML to design the page.**
   Individual screens in MAUI apps are called **pages**. You'll design them using XAML, a design language you'll use to define how those pages work.

3. **You'll write C# code to add random animal emoji to the page.**
   When your app first loads, it will run that code to display sixteen buttons with eight pairs of animal emoji in a random order.

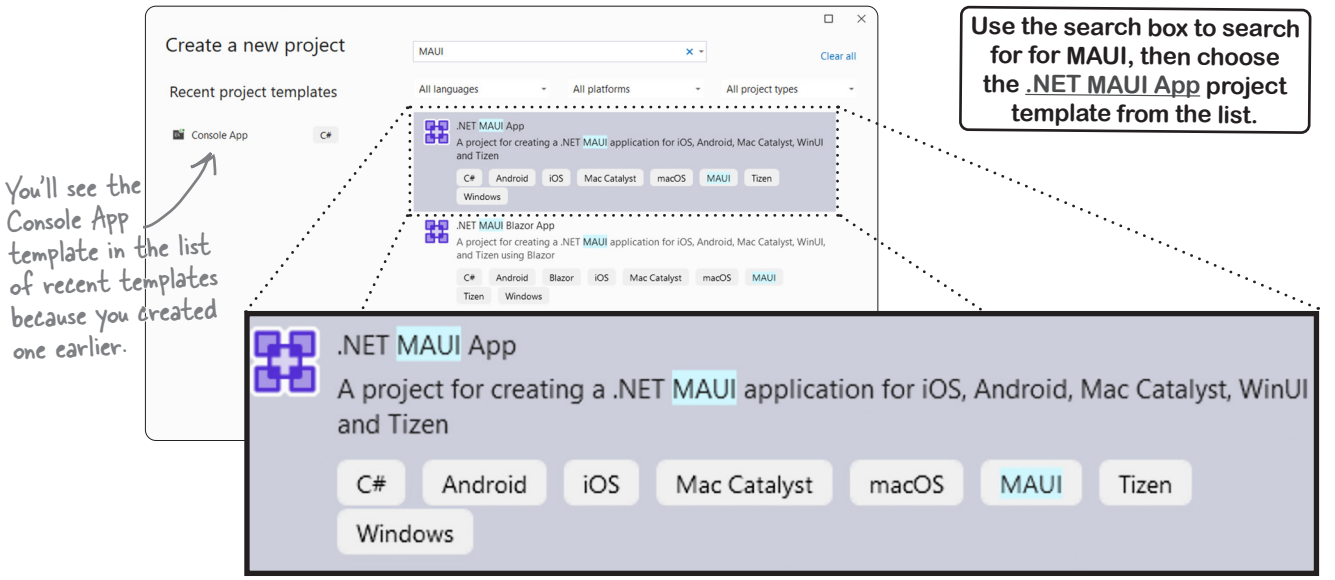


4. **You'll make the gameplay work.**
   The game needs to detect when the user clicks on pairs of emoji, keep track of the pairs, and end the game when they've found all of the matches. You'll write that code too.

5. **Finally, you'll make the game more exciting by adding a timer.**
   Your timer will start when the player starts the game, and keep track of how long it takes the player to find all eight pairs of animals.

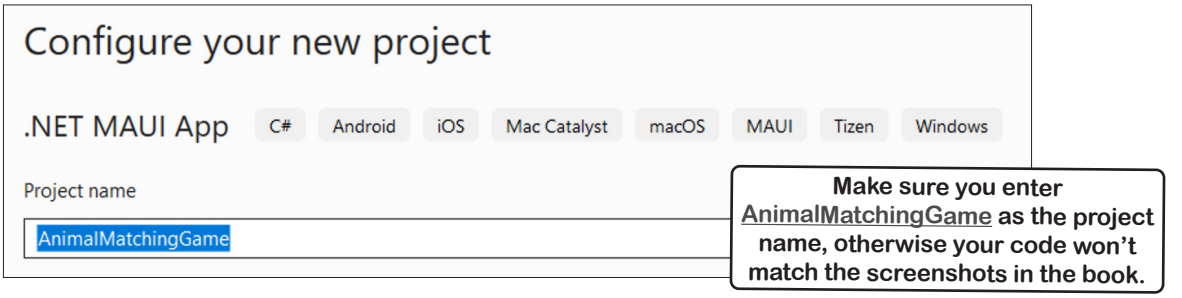
Create a .NET MAUI project in Visual Studio – Windows edition

You can create a .NET MAUI app in Visual Studio just like you did with the console app at the beginning of the chapter. Open up Visual Studio or choose New >> Project (Ctrl+Shift+N) from the File menu to bring up the "Create a new project" window.



Choose the **.NET MAUI App project template** and click Next. Visual Studio will prompt you for a project name, just like it did when you created a Console App project.

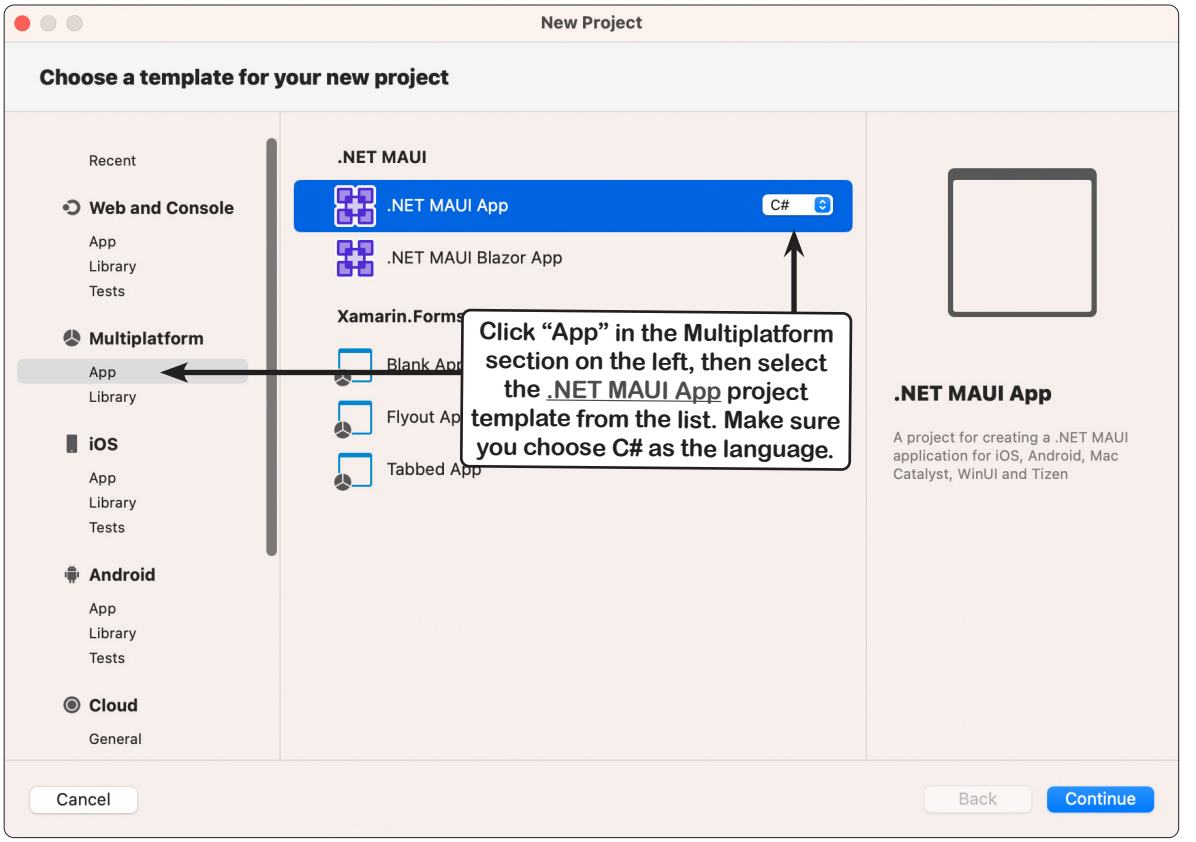
Enter **AnimalMatchingGame** as the project name and click Next.



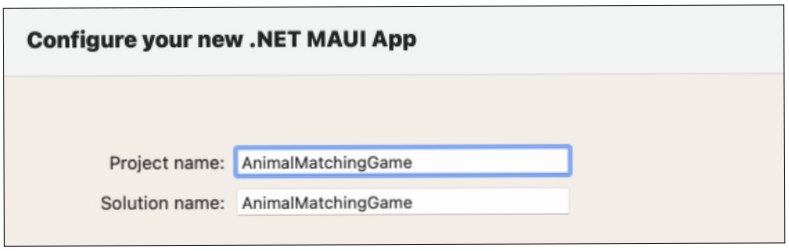
Finally, Visual Studio will ask you to choose a version of .NET – choose the latest version, just like you did when you created the Console App project. Then click the Create button to create your new .NET MAUI project.

Create a .NET MAUI project in Visual Studio – Mac edition

If you're using Visual Studio for Mac, creating a .NET MAUI App project is really similar to creating the Console App project, just like you did at the beginning of the chapter. Start Visual Studio and click + New – or choose New Project... (⇧⌘N) from the File menu – to bring up the New Project window.



Visual Studio will ask you for the version of .NET, like it did when you created a console app. Choose the latest version and click Continue.

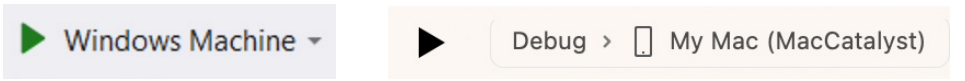


Make sure you enter AnimalMatchingGame as the project name, otherwise your code won't match the screenshots in the book.

Finally, Visual Studio will ask you for a project name. **Enter AnimalMatchingGame**, then click the Create button to create your new .NET MAUI project.

Run your new .NET MAUI app

Go ahead and **run your new .NET MAUI app**. You can click the Run button in the toolbar:

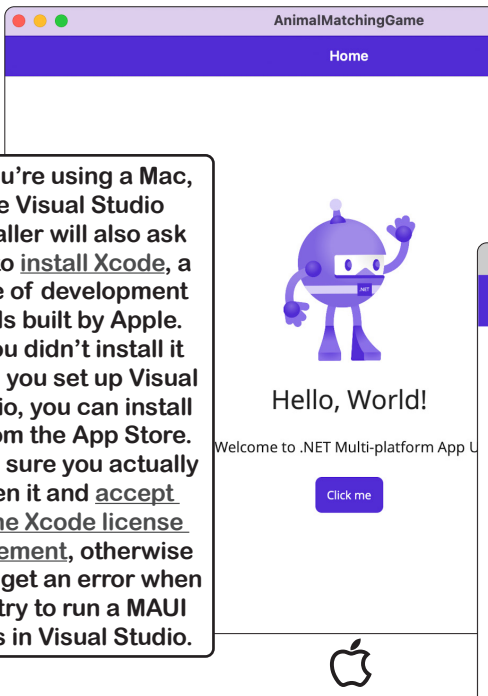


Do this!

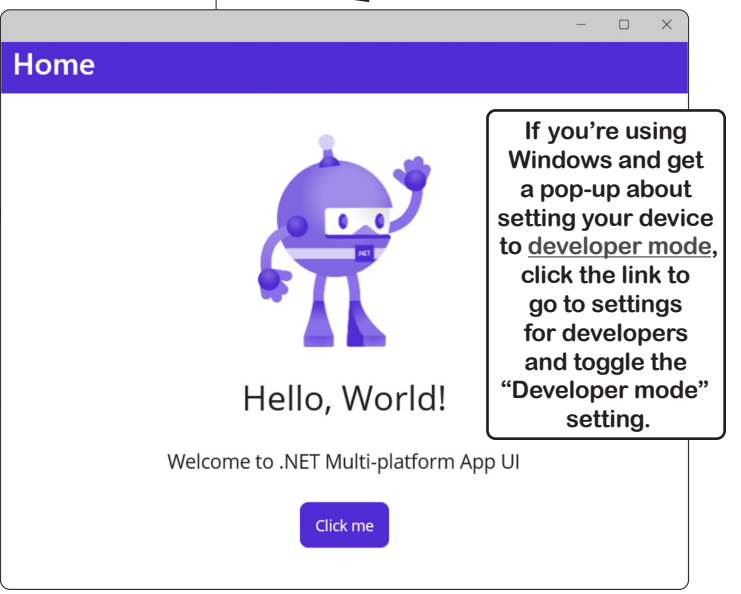
Or you can choose Start Debugging (F5 or ⌘↵) from the Debug menu.

Visual Studio will build your code, which means converting it to an executable program that your operating system can run. Then it will start your app:

When you see Do this! (or Now do this!, or Debug this!, etc.), go to Visual Studio and follow along. We'll tell you exactly what to do, and point out what to look for to get the most out of the example we show you.



If you're using a Mac, the Visual Studio installer will also ask you to install Xcode, a suite of development tools built by Apple. If you didn't install it when you set up Visual Studio, you can install it from the App Store. Make sure you actually open it and accept to the Xcode license agreement, otherwise you'll get an error when you try to run a MAUI apps in Visual Studio.



If you're using Windows and get a pop-up about setting your device to developer mode, click the link to go to settings for developers and toggle the "Developer mode" setting.

Stop your MAUI app

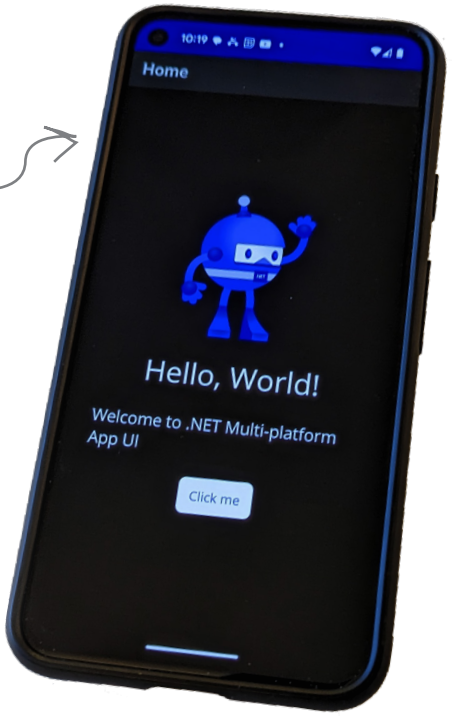
You can stop your app by closing the app window. You can also choose Stop Debugging (Shift+F5 or ⇧⌘↵) from the Debug menu, or click the square Stop button in the Visual Studio toolbar.

You can start or stop your app at any time. If there are syntax errors (like typos or invalid keywords) in the C# or XAML code, Visual Studio won't be able to run the app.

MAUI apps work on all of your devices

MAUI is a **cross-platform framework** for building visual apps, which means the apps that you build can run on your Android and iOS devices. If you have an Android phone or tablet, for example, you can set it to developer mode, plug it into your computer, and tell Visual Studio to deploy the app straight to your phone. You can do the same thing with your iPhone or iPad, but Apple requires you to join the Apple Developer Program before you can do that—at the time we're writing this costs USD $99 (although nonprofits, government agencies, and students can get a fee waiver).

We just plugged in an Android phone in and deployed the .NET MAUI app to it... and it worked!



You can run MAUI apps on an Android device right from Visual Studio. This page shows you how to set up an Android device so you can connect it to your computer run your MAUI apps on it: https://learn.microsoft.com/en-us/dotnet/maui/android/device/setup

You can also run MAUI apps on your iOS device, but it requires a little more setup—and it costs money, because you need to join the Apple Developer Program. This page walks you through the whole process: https://learn.microsoft.com/en-us/dotnet/maui/ios/device-provisioning/

MAUI apps are designed with XAML

XAML (the X is pronounced like Z, and it rhymes with "camel") is a **markup language** that you'll use to build the user interfaces for your MAUI apps. XAML is based on XML (so if will generate you've ever worked with HTML you have a head start). Here's an example of a XAML tag for a button:

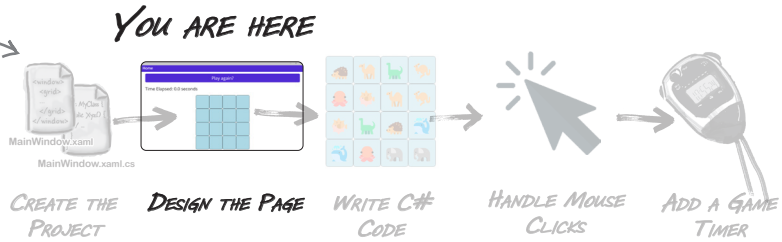
```
<Button Text="Click" Clicked="Button_Click" />
```

This book is about learning C#, so we'll give you **just enough** XAML so you can build great-looking MAUI apps—and we'll make sure that you have a solid foundation for learning more.

A lot of C# developers consider XAML a core skill, and many C# jobs require you to know at least some XAML, so we wanted to make sure to include enough of it in this book to give you a good grounding in it.

Many of the chapters in this book include .NET MAUI projects, so you can learn to build interactive apps that can run on computers running Windows or macOS, and mobile devices running Android or iOS.

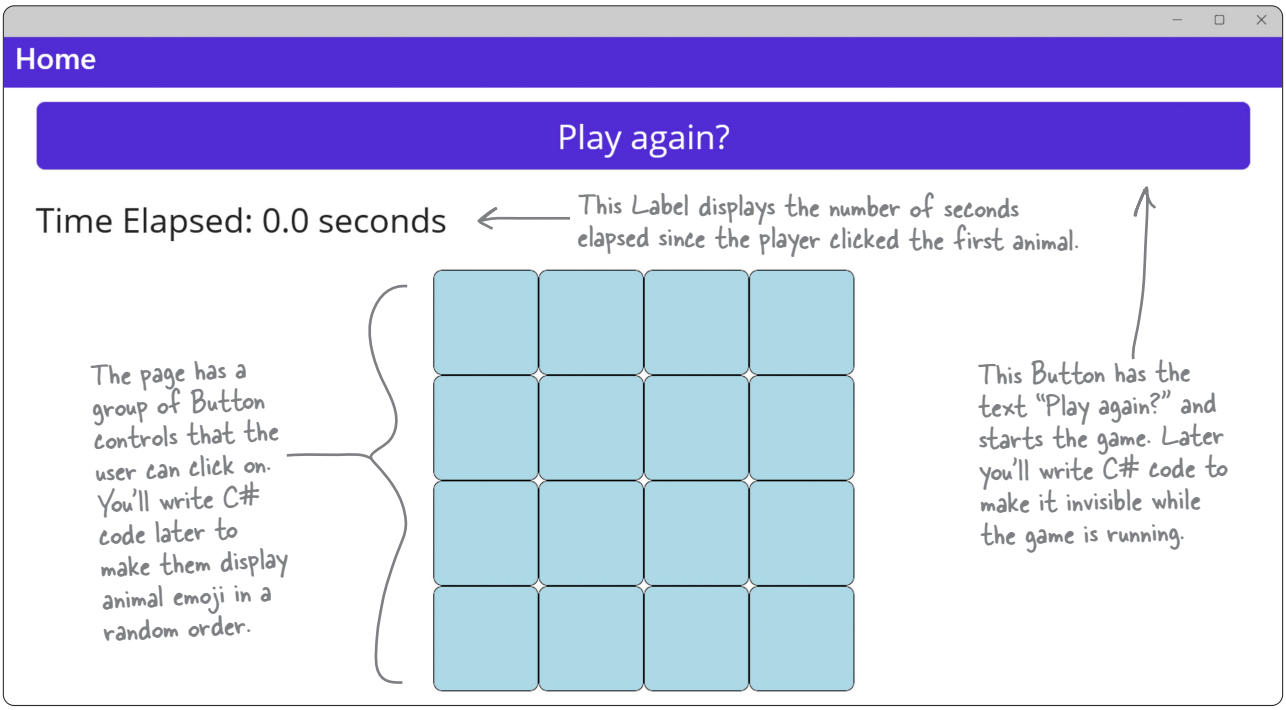
We'll include a "map" like this at the start of each of the sections of this project to help you keep track of the big picture.



Here's the page that you'll build

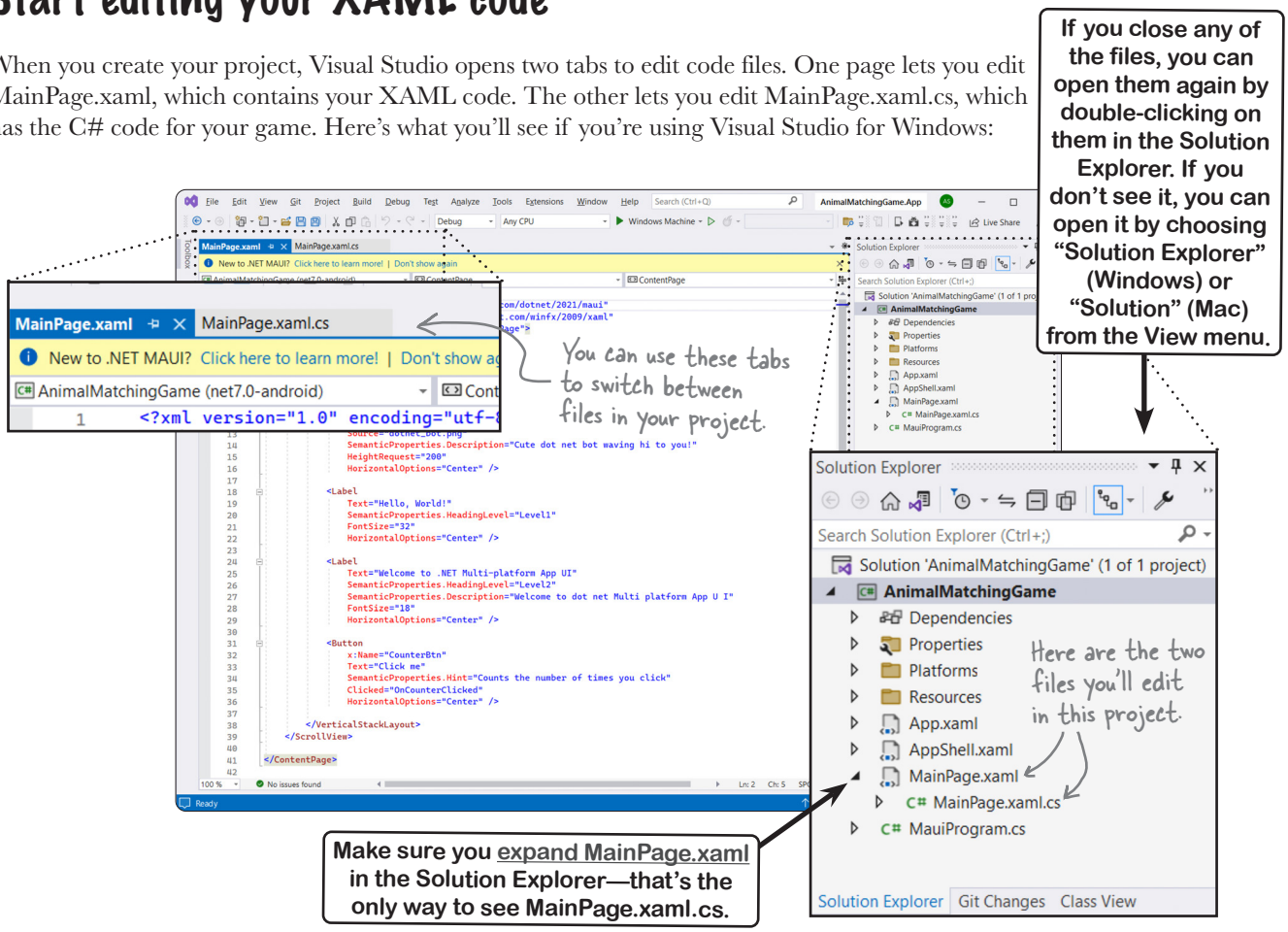
When you start a project, the first thing you always want to do is take a few minutes to understand the big picture. What are you going to create? How will it work? Let's take a look at the page you're about to build.

When you open an app built with .NET MAUI, the first thing it shows you is a **page** that you interact with. That page uses **controls**, or visual widgets like buttons and labels, to create a user interface (or UI) that you can interact with. Here's the page that you're going to design:

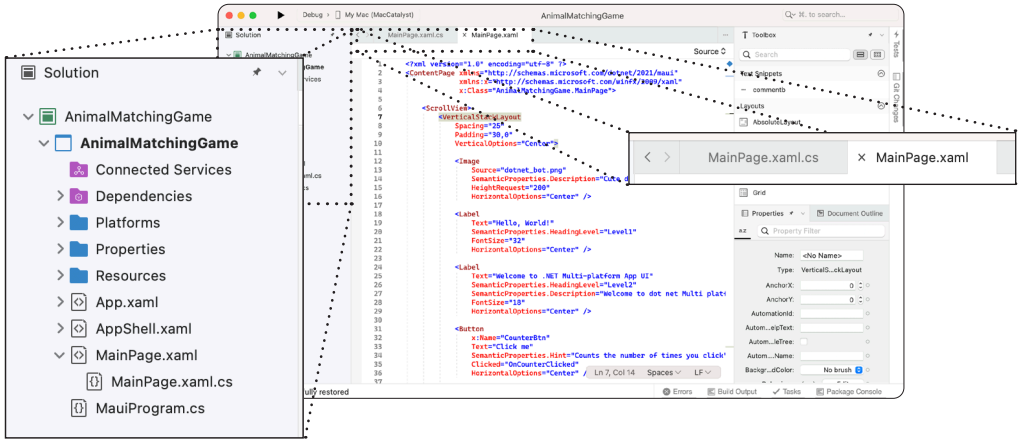


Start editing your XAML code

When you create your project, Visual Studio opens two tabs to edit code files. One page lets you edit MainPage.xaml, which contains your XAML code. The other lets you edit MainPage.xaml.cs, which has the C# code for your game. Here's what you'll see if you're using Visual Studio for Windows:



Visual Studio for Mac looks a little different, but works exactly the same way.



Add the XAML for a Button and a Label

The first thing we'll do is Design the page for the game. It will have sixteen buttons to display the animal emoji, plus a "Play again?" button to restart the game when the player wins.

1 **Delete everything between the opening and closing VerticalStackLayout tags.**
XAML is a **tag-based markup language**. That means your XAML code uses **tags** to define everything that appears in your app. Here's an example of a tag—you can find it on line 6 of MainPage.xaml:

```
<ScrollView>
```

That's an **opening tag**. You can find its matching **closing tag** on line 39: `</ScrollView>`

This is a ScrollView. If your app is in a window that's smaller than its contents, everything between the opening and closing tag can be scrolled up and down.

Find the opening VerticalStackLayout tag. It's on lines 7 through 10, and it looks like this:

```
        <VerticalStackLayout
            Spacing="25"
            Padding="30,0"
            VerticalOptions="Center">
```

Next, find the closing VerticalStackLayout tag on line 38:

```
        </VerticalStackLayout>
```

Now **carefully delete all of the lines between those two two tags**. The XAML code in your MainPage.xaml file should now look like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="AnimalMatchingGame.MainPage">

    <ScrollView>

        <VerticalStackLayout
            Spacing="25"
            Padding="30,0"
            VerticalOptions="Center">


        </VerticalStackLayout>
    </ScrollView>

</ContentPage>
```

In the next step, you'll put your new XAML code right here, where you deleted the old code.

② Delete the C# code that goes with the XAML that you just deleted.

If you try to run your app right now, Visual Studio will give you an error message and refuse to run it, because the C# code depends on some things that you just deleted. First, find this code on line 12:

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

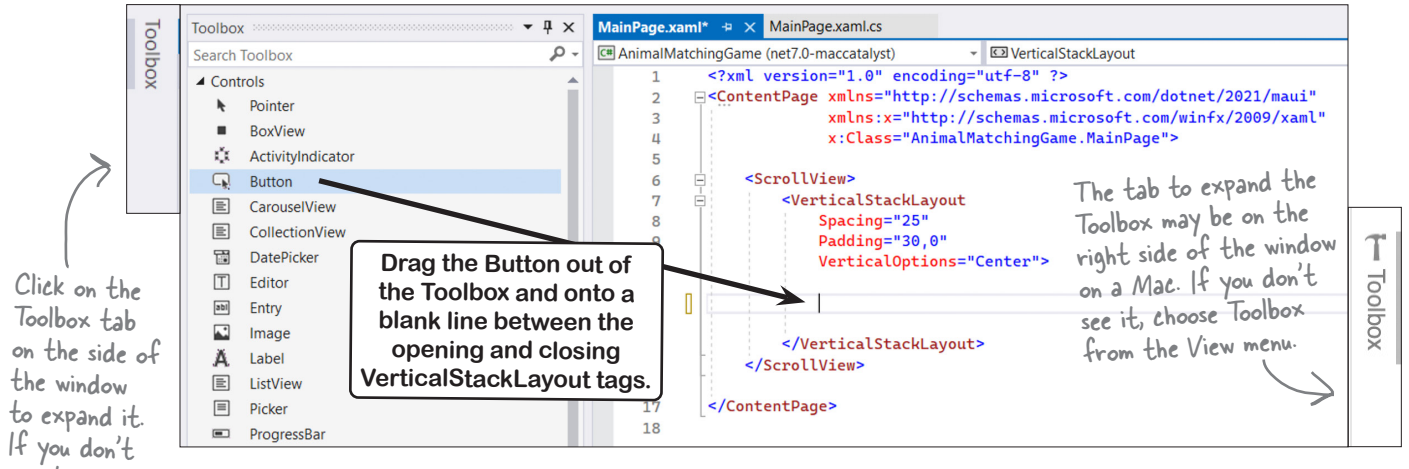
Delete it, and the next 10 lines of code, up to and including the closing curly brace `}` on line 22. Be careful not to delete the final closing `}` on line 24. Then delete the line of code on line 5: `int count = 0;`

Your C# code should now look like this:

```
namespace AnimalMatchingGame;

public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();
    }
}
```

③ Use the Toolbox to add the "Play Again?" button.

You'll be editing the XAML code again, so switch back to the MainPage.xaml tab. If you don't see the Toolbox panel, expand it by clicking the tab on the side of the window. **Add a few extra blank lines** where you deleted the code between the opening and closing VerticalStackLayout tags. Then **drag the Button out of the Toolbox** and drop it onto tone of the lines that you added.



You should now see a new Button tag between the VerticalStackLayout tags—it's okay if the spacing or indenting is a little different, because extra spaces or lines don't matter in XAML:

```
<VerticalStackLayout
    Spacing="25"
    Padding="30,0"
    VerticalOptions="Center">

    <Button Text="" />

</VerticalStackLayout>
```

When you dragged the Button out of the toolbox and into your code, Visual Studio added this Button tag.

④ Add Properties to the XAML tag for the "Play again?" button.

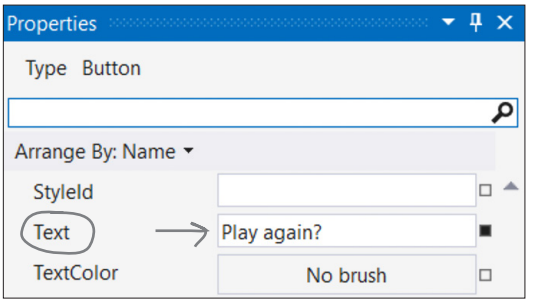
XAML tags have **properties** that let you set options to customize how they're displayed on the page. The **Properties window** in Visual Studio makes it easier to edit them.

Click on the code for the Button tag in MainPage.xaml, so your cursor is somewhere between the opening **<** and closing **>** angle brackets. Then look at the **Properties window**—it's usually docked in the lower right corner of Visual Studio. If you don't see it, choose Properties or Properties Window from the View menu. Make sure it says "Type Button" at the top, so you know that you're editing the Button.

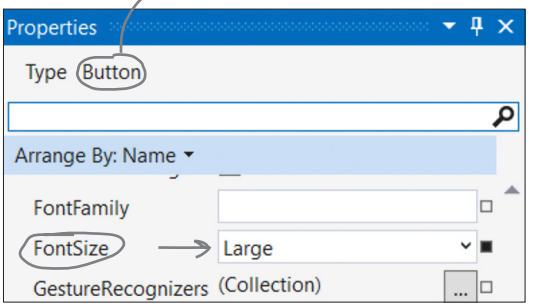
Find the ***Text*** property and **set it to `Play again?`**

Then find the ***FontSize*** property and **set it to `Large`**

The Mac properties window looks a little different, but works exactly the same way.



Make sure you're editing the Button.



When you're done editing the button, the XAML for it should look like this:

```
<Button Text="Play again?" FontSize="Large" />
```

The Button tag now has **Text and FontSize properties**.

⑤ Edit the XAML code by hand for your button to give it a name.

You can also edit XAML code by hand—for example, if you ran into trouble with the Properties window, you could type the XAML directly into the editor. ***You need to make sure that you copy all of the brackets, quotes, etc., exactly, otherwise your code won't run!***

In the next part of the project, you'll write C# code to make your "Play again?" button visible when the game is over, and invisible while the game is running. You'll give it a **name** that the C# code can use to tell it to show or hide itself.

Use the editor to **add an x:Name property** to give your Button a name. It should look like this:

```
<Button x:Name="PlayAgainButton" Text="Play again?" FontSize="Large" />
```

XAML tags have properties that let you set options to customize how they're displayed on the page.

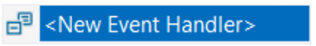
⑥ Add an event handler so your button does something.

When you click a button, it executes C# code called an **event handler**. Visual Studio makes it easy to add one. Place your mouse cursor just before the **/>** at the end of the Button tag and start typing **Clicked**. Visual Studio will pop up an IntelliSense window:

```
FontSize="Large" Cli />
```

Choose Clicked from the list and either click on it or press Enter. Visual Studio will then show you this:

```
FontSize="Large" Clicked="" />
```



Your Button tag can be on a single line or split across multiple lines.

Press Enter to add a new event handler. Your XAML tag should now look like this:

```
<Button x:Name="PlayAgainButton" Text="Play again?" FontSize="Large"
        Clicked="PlayAgainButton_Clicked" />
```

Switch to the MainPage.xaml.cs tab. You can see the code that Visual Studio added, which looks like this:

```
private void PlayAgainButton_Clicked(object sender, EventArgs e)
{

}
```

When you see an Exercise, that's your chance to get some practice on your own. Make sure you do every exercise—they're an important part of the book. If an exercise is part of a project, then the project won't work until you get it right. But don't worry—we'll always give you the solution. And if you get stuck, it's always okay to peek at the solution!

Exercise

Add a Label control to your XAML page.

Go back to the screenshot of the game that shows the "Play again?" button. Notice how it also has text above the button that displays the time elapsed? That's a Label. It's up to you to add a tag for it. Here's what you'll do:

1. Switch to the MainPage.xaml tab.
2. Open the Toolbox and **drag a Label** into your XAML code. Make sure it gets added directly below the Button, just like you did in Step 3 when you were adding the Button.
3. Use the Properties window to set the **Text button to "Time Elapsed: 0.0 seconds"** and the **FontSize to "Large"** just like you did in Step 4 when you were adding the Button.
4. Edit the XAML code by hand and **set the x:Name to "TimeElapsed"** just like you did in Step 5 when you were adding the Button.

Exercise Solution

Add a Label control to your XAML page.

If you followed the steps in the Exercise correctly, your XAML code in MainPage.xaml should now look like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="AnimalMatchingGame.MainPage">

    <ScrollView>
        <VerticalStackLayout
            Spacing="25"
            Padding="30,0"
            VerticalOptions="Center">

            <Button x:Name="PlayAgainButton" Text="Play again?" FontSize="Large"
                    Clicked="PlayAgainButton_Clicked" />

       <Label x:Name="TimeElapsed" Text="Time Elapsed: 0.0 seconds" FontSize="Large" />

        </VerticalStackLayout>
    </ScrollView>

</ContentPage>
```

It's okay if there are line breaks between the properties in a tag, or if the properties are in a different order.

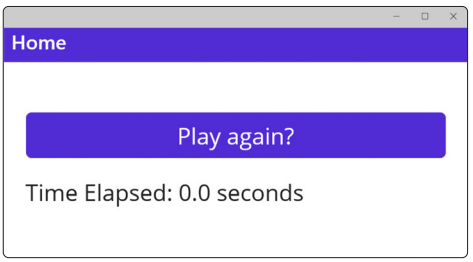
Here's the Label that you added in the Exercise.

The C# code in MainPage.xaml.cs didn't get modified as part of the Exercise, so it should still look like this:

```
namespace AnimalMatchingGame;

public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();
    }

    private void PlayAgainButton_Clicked(object sender, EventArgs e)
    {

    }
}
```

Make sure that your XAML and C# code matches and your app looks like this screenshot before moving on.

If you run your app now, it should look like this



there are no Dumb Questions

Q: What exactly is a "page" in a MAUI app?

A: A .NET MAUI app is usually built out of one or more **pages**, or individual screens that different layouts and contain **controls** like labels and buttons. Some MAUI apps have mutliple pages that let you navigate between them. Your Animal Matching Game app will just have a single page with sixteen animal buttons, a "Play again?" button, and a label to show the elapsed time.

Q: So those buttons and labels are controls?

A: Yes. Everything you see on a MAUI page is a control—including the page itself, which is a ContentPage control. Some controls are dedicated to making your page look a certain way, like the VerticalStackLayout control that causes other controls to be stacked one on top of another. Others, like the Button and Label controls, are there to display some kind of widget that the user can see and interact with. We'll talk more about controls in the next chapter.

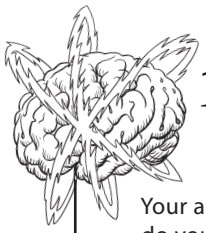
Q: It looks like some controls contain others, like the VerticalStackLayout in my app contains Button and a Label. What's going on there?

A: When you include layout control like VerticalStackLayout on your page, you can't actually see it. Its whole purpose is to cause the other controls on the page to be displayed a certain way—in this case, to be stacked on top of each other. You need a way to tell MAUI which other controls on the page you want it to stack. To do that, you **nest** those other controls inside the VerticalStackLayout by including their tags between its opening <VerticalStackLayout> tag and its closing </VerticalStackLayout> tag.

Q: Why do some tags like <ScrollView> have a closing </ScrollView> tag, but others like <Button> don't have one?

A: A Button control doesn't need to have any other controls nested inside of it, so there's no need for it to have a closing tag—instead, you can just end the tag with **/>** to make it **self-closing**.

These Brain Power boxes are here to give you something to think about. When you see one, don't just go on to the next section. Take a few minutes and actually think about what you're being asked. That will really help you get this material into your brain faster!



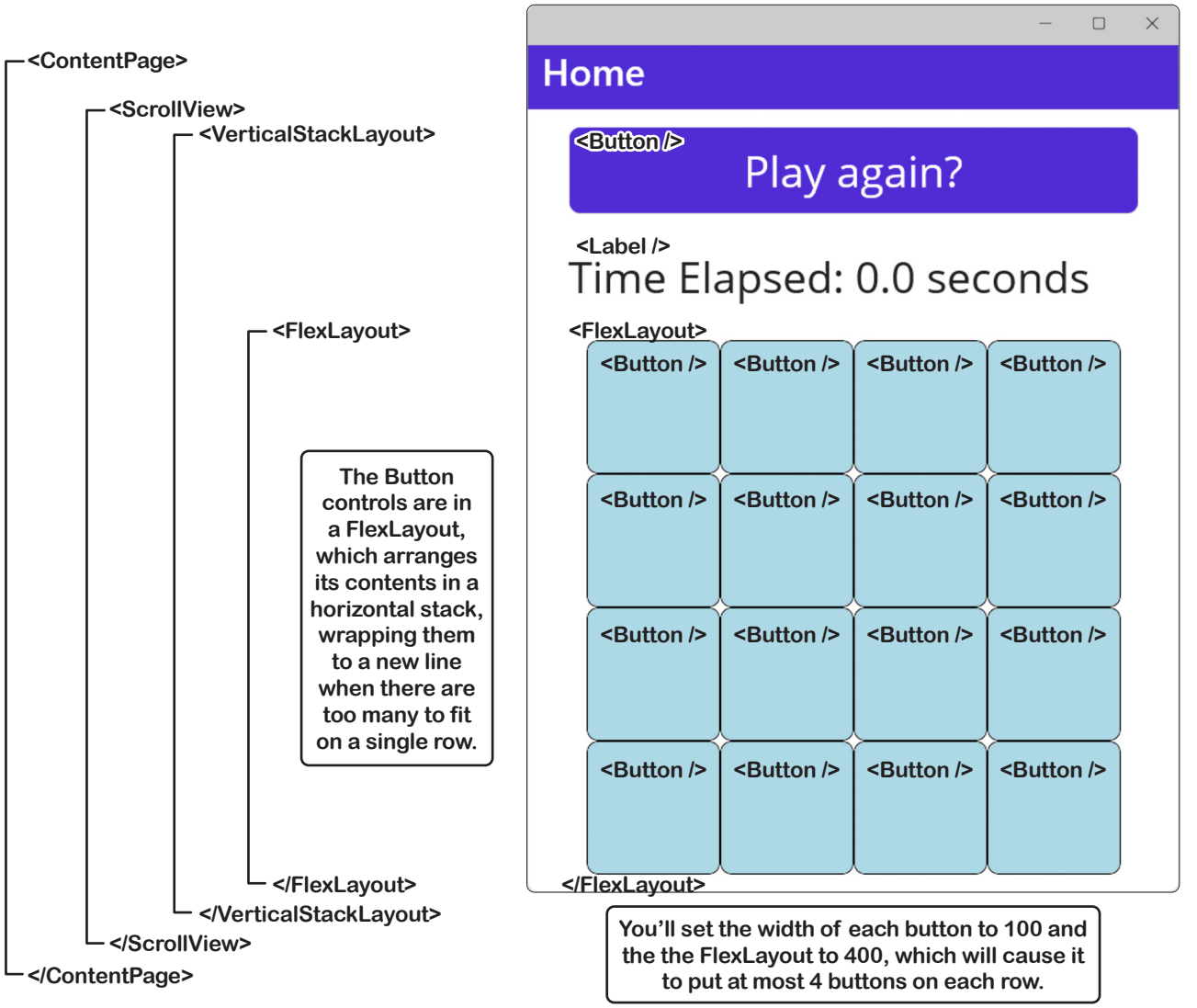
Brain Power

Your app is looking good so far, but now you need to add some buttons. How do you think you'll do that? What do you think you'll have to add to the XAML to get sixteen buttons to be displayed in a layout with four rows of four buttons?

Use a FlexLayout to make a grid of animal buttons

The XAML for your page currently has three tags that determine its layout: there's a ContentPage tag on the outside that displays the whole view. It contains a ScrollView—everything nested between its start and end tags will scroll if it goes off the bottom of the page. Inside it is a VerticalStackLayout, which causes everything between its start and end tags to be stacked on top of each other in the order that they appear. Inside all of those tags are self-closing Buton and Label tags.

Now you add a **FlexLayout**, which arranges anything inside of it in rows, wrapping them to the next row so they all fit inside its total width. You'll add sixteen Button tags inside the FlexLayout. You'll get them to display in a 4x4 grid by setting the width of each button to 100 and the width of the FlexLayout to 400, so exactly four buttons will fit on each row.



WARNING! At the time we're writing this, there's a bug in the macOS version of .NET MAUI where the FlexLayout's MaximumWidthRequest property does not work, so you won't see a nice 4x4 grid like this. 😿

Exercise

This looks like a big exercise! But don't worry, just take it step by step. We know you can do it! And remember, it's not cheating to look at the solution… in fact, seeing the solution is a great way to help you learn.

It's time to finish designing your page. In this exercise, you'll add a FlexLayout underneath the Label that you added in the last Exercise. Next, you'll set its properties. Then you'll add a button. And finally, you'll copy the XAML for that button and paste it fifteen more times, so you have a total of sixteen buttons on your page.

Add extra space for your FlexLayout control

Take a careful look at the screenshot that we just showed you. It shows you how the whole page works. Now go back to Visual Studio and look at the XAML for your page, and figure out exactly where the FlexLayout should go—just below the <Label ... /> tag.

Now put your cursor at that location and press enter a few times to give yourself space to drag the FlexLayout.

Add the FlexLayout control just below the Label

1. Open the Toolbox and **drag a FlexLayout** into your XAML code. Make sure it gets added directly below the Label, into the extra space you just added. It will look like this: `<FlexLayout></FlexLayout>`
2. Position your cursor between the **>** and **<** in the middle of the XAML you just added and **add several extra spaces** between the opening and closing tags (you'll drag a button into that space later in the exercise).
3. Place your cursor directly on the opening `<FlexLayout>` tag. Make sure the Properties window shows that the type is FlexLayout.
4. Use the Properties window to set the **Wrap property** to ***Wrap*** and the **MaximumWidthRequest property** to ***400***.
5. Edit the XAML code by hand and **set the x:Name to "AnimalButtons"** just like you did in the last Exercise.

Add the first Button inside the FlexLayout

1. Open the Toolbox and **drag a Buton** into your XAML code. Make sure it gets added in the space that you addefd between the opening and closing tags of the FlexLayout. It will look like this: `<Button Text="" />`
2. Place your cursor inside the Button tag. Make sure the Properties window shows that the type is Button.
3. Use the Properties window to set the Button's **HeightRequest property** to ***100***, the **WidthRequest property** to ***100***, and the **FontSize property** to ***60***. The dropdown in the Properties window won't have numbers—you can either type 60 into the window, or choose Caption from the dropdown list to set the font size.
4. Edit the XAML for the button and **delete the Text property** by selecting it in the code editor and pressing delete. Your cursor should now be inside the Button control.
5. Keep the cursor where it is and edit the XAML code by hand to **set the BackgroundColor property** to ***LightBlue*** and the **the BorderColor property** to ***Black***. Visual Studio's IntelliSense pop-up will help you match the colors.
6. Add a **Clicked event handler**, just like you did with PlayAgainButton. **Choose <New Event Handler>** from the dropdown, so it creates a new event handler method in the C# code. Use the default name Button_Clicked.

Add the rest of the Buttons

Copy the <Button ... /> tag that you just added. Then **paste 15 identical tags below it**. You should now have a total of 16 identical Button tags inside a FlexLayout just below the Label. Run your app—it should match our screenshot.

Exercise Solution

It's time to finish designing your page. In this exercise, you'll add a FlexLayout underneath the Label that you added in the last Exercise. Next, you'll set its properties. Then you'll add a button. And finally, you'll copy the XAML for that button and paste it fifteen more times, so you have a total of sixteen buttons on your page.

If you followed the steps in the Exercise correctly, your XAML code in MainPage.xaml should now look like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="AnimalMatchingGame.MainPage">

  <ScrollView>
     <VerticalStackLayout
            Spacing="25"
            Padding="30,0"
            VerticalOptions="Center">

     <Button x:Name="PlayAgainButton" Text="Play again?" FontSize="Large"
                    Clicked="PlayAgainButton_Clicked" />

     <Label x:Name="TimeElapsed" Text="Time Elapsed: 0.0 seconds" FontSize="Large" />

     <FlexLayout x:Name="AnimalButtons" Wrap="Wrap" MaximumWidthRequest="400">

       <Button BackgroundColor="LightBlue" BorderColor="Black" Clicked="Button_Clicked"
                        HeightRequest="100" WidthRequest="100" FontSize="60" />
       <Button BackgroundColor="LightBlue" BorderColor="Black" Clicked="Button_Clicked"
                        HeightRequest="100" WidthRequest="100" FontSize="60" />
       <Button BackgroundColor="LightBlue" BorderColor="Black" Clicked="Button_Clicked"
                        HeightRequest="100" WidthRequest="100" FontSize="60" />
       <Button BackgroundColor="LightBlue" BorderColor="Black" Clicked="Button_Clicked"
                        HeightRequest="100" WidthRequest="100" FontSize="60" />
       <Button BackgroundColor="LightBlue" BorderColor="Black" Clicked="Button_Clicked"
                        HeightRequest="100" WidthRequest="100" FontSize="60" />
       <Button BackgroundColor="LightBlue" BorderColor="Black" Clicked="Button_Clicked"
                        HeightRequest="100" WidthRequest="100" FontSize="60" />
       <Button BackgroundColor="LightBlue" BorderColor="Black" Clicked="Button_Clicked"
                        HeightRequest="100" WidthRequest="100" FontSize="60" />
       <Button BackgroundColor="LightBlue" BorderColor="Black" Clicked="Button_Clicked"
                        HeightRequest="100" WidthRequest="100" FontSize="60" />
       <Button BackgroundColor="LightBlue" BorderColor="Black" Clicked="Button_Clicked"
                        HeightRequest="100" WidthRequest="100" FontSize="60" />
       <Button BackgroundColor="LightBlue" BorderColor="Black" Clicked="Button_Clicked"
                        HeightRequest="100" WidthRequest="100" FontSize="60" />
```

If you chose a different name for your project, you'll see that name here.

This looks like a lot of XAML code, but most of it is just the sixteen identical Button tags that you copied and pasted.

Make sure the opening tag of your FlexLayout is right below the Label, and that its properties match ours. Be careful to add a Maximum and not Minumum width request or your button's won't be in a 4x4 grid.

It looks like there is a lot of XAML code here! But most of the XAML that you added is the same <Button ... /> tag copied and pasted sixteen times.

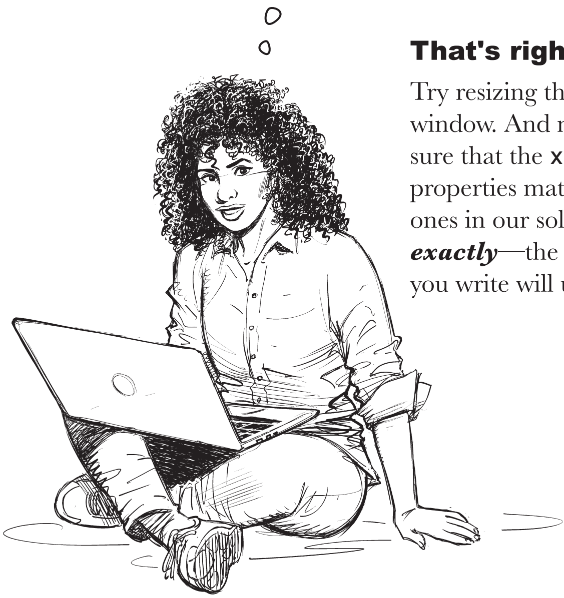
Exercise Solution

The sixteen Button tags should be identical. It's okay if the properties are in a different order.

```
        <Button BackgroundColor="LightBlue" BorderColor="Black" Clicked="Button_Clicked"
                 HeightRequest="100" WidthRequest="100" FontSize="60" />
        <Button BackgroundColor="LightBlue" BorderColor="Black" Clicked="Button_Clicked"
                 HeightRequest="100" WidthRequest="100" FontSize="60" />
        <Button BackgroundColor="LightBlue" BorderColor="Black" Clicked="Button_Clicked"
                 HeightRequest="100" WidthRequest="100" FontSize="60" />
        <Button BackgroundColor="LightBlue" BorderColor="Black" Clicked="Button_Clicked"
                 HeightRequest="100" WidthRequest="100" FontSize="60" />
        <Button BackgroundColor="LightBlue" BorderColor="Black" Clicked="Button_Clicked"
                 HeightRequest="100" WidthRequest="100" FontSize="60" />
        <Button BackgroundColor="LightBlue" BorderColor="Black" Clicked="Button_Clicked"
                 HeightRequest="100" WidthRequest="100" FontSize="60" />
      </FlexLayout>

    </VerticalStackLayout>
  </ScrollView>

</ContentPage>
```

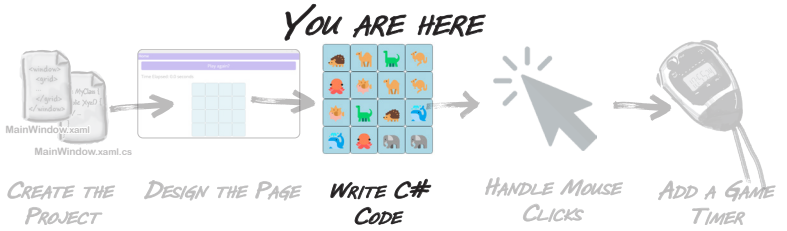
Make sure **every button** has the `Clicked="Button_Clicked"` property. If the Clicked event handler has a different name, your C# code won't match ours. You can delete the Clicked property from all of the buttons, then re-add it with the correct name. Once you add the event handler, it will show up in the dropdown when you change the other buttons.

I CAN CHECK IF MY SOLUTION IS RIGHT BY *COMPARING IT WITH THE SCREENSHOT*. THAT MAKES IT EASIER!



That's right!

Try resizing the window. And make sure that the `x:Name` properties match the ones in our solution ***exactly***—the C# code you write will use them.



Write C# code to add the animals to the buttons

You started this book to learn C#. You've done all the preparation: creating the project, designing the page for your app. Now it's time to **get started writing C# code**.

We'll give you all of the code for this project, and show you exactly where it goes. But the goal is to ***get you started learning C#***, so we'll also work with you to help you understand how it all works—and that will provide you with a solid foundation to start writing code on your own.

You'll add code that's run every time the "Play again?" button is clicked. Here's what it will do:

```
Make the animal buttons visible
                ↓
Make the "Play again?" button invisible
                ↓
Create a list of 16 pairs of animal emoji
                ↓
For each of the 16 buttons:  ←──────────┐
    Pick a random animal from the list  │
                ↓                       │
    Add that random animal to the button│
                ↓                       │
    Remove the animal from the list     │
                ↓                       │
    Keep going until it runs out of buttons ┘
```

Start editing the PlayAgainButton event handler method

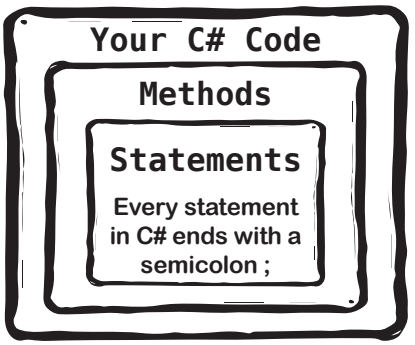
When you were writing the XAML code for the "Play again?" button, you added an event handler:

```
FontSize="Large" Clicked="" />
```

<New Event Handler>

When you did this, Visaul Studio added `Clicked="PlayAgainButton_Clicked"` to the XAML tag for the button. It also added this C# code to MainPage.xaml.cs:

```
private void PlayAgainButton_Clicked(object sender, EventArgs e)
{

}
```



That's a **method**. C# code is made up of statements, or specific tasks that you're telling your app to execute. Those statements are bundled into methods. Methods have a name—this method is named `PlayAgainButton_Clicked`.

Visual Studio generated that method for you automatically when you added the Clicked event handler to your XAML code to give you a place to add the statements that will tell it what to do when the "Play again?" button is clicked.

Add a C# statement to the event handler method

Place your cursor on the line between the opening { curly bracket and closing } curly bracket of the method. Then start typing the following line of code to make the animal buttons visible:

Do this!

```
AnimalButtons.IsVisible = true;
```

As you're typing, you'll see some of Visual Studio's really powerful tools that help you write code:

```
0 references
private void PlayAgainButton_Clicked(object sender, EventArgs e
{
    AnimalButtons.IsVisible=true;
```

Tab Tab to accept | Alt+. next suggestion

- IsLoaded
- IsReadOnly
- IsSet
- IsVisible
- Join<>
- JustifyContent
- Last<>
- LastOrDefault<>
- Layout

This is an IntelliSense window. Visual Studio displays it as soon as you pressed the period key. It shows you options that you can use to add to the code you just wrote. If you choose "IsVisible" it will auto-complete that part of the statement for you.

When you're typing code, you may see Visual Studio giving you suggestions to complete the entire statement. This is a really powerful feature called IntelliCode. It uses an artificial intelligence system trained on millions of lines code to give you suggestions—and more often than not, it seems to be able to read your mind!

Add more statements to your event handler

When the player clicks the "Play again?" button, the app will display the animal buttons, hide the "Play again?" button, and then fill the animal buttons eight pairs of animal emoji in a random order. You're going to add statements to the PlayAgainButton_Clicked event handler method to do all that.

Do this!

1 Add a statement to make the "Play again?" button invisible.

Do you remember how you used the x:Name property in your XAML code to give names to the "Play again?" button and the FlexLayout that contains the sixteen animal buttons? Take a minute and go back to that XAML code—you'll you gave the FlexLayout the name *AnimalButtons*, and you just added a line of code which used that name.

You also used an x:Name to give the "Play again?" button the name *PlayAgainButton*. Now add a second line of code to your event handler method:

```
private void PlayAgainButton_Clicked(object sender, EventArgs e)
{
    AnimalButtons.IsVisible = true;
    PlayAgainButton.IsVisible = false;
}
```

Add this line of code right below the one you just added.

That statement turns the "Play again?" button invisible.

2 Make the animal buttons invisible when the app starts.

Take a closer look at the first statement that you added to your event handler method. It turns the FlexLayout that contains the animal buttons visible. But wait a minute—it's already visible! You saw it when you ran your app. Let's do something about that.

Go to top of your MainPage.xaml.cs file and find a method that starts `public MainPage()`. This is a special method that gets executed once while your page is loading. It has a statement in it – `InitializeComponent();` – that initializes the page.

Add a statement right after that one to make the animal buttons invisible:

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

    AnimalButtons.IsVisible = false;
}
```

We made the code that's already in your MainPage.xaml.cs file gray to make it easier for you to see what to add.

Add this statement to make the FlexLayout with the animal buttons invisible after the page is initialized.

Now your app will make the animal buttons invisible when it starts up. As soon as the player clicks the "Play again?" button to start the game, it will show the animal buttons and hide the "Play again?" button.

3 Run your app and make sure it works so far.

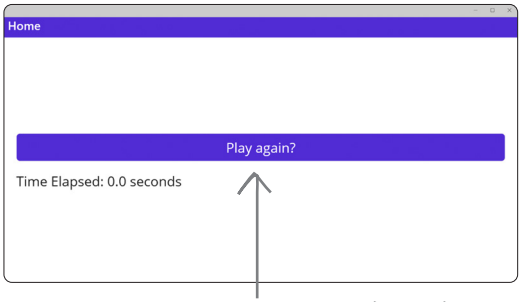
When you're writing code, you don't just write a complete app from beginning to end, then run it to see if it works. That's not how it works at all! ***Writing code is a creative process.*** There are many, many ways to make your code do a specific thing, and in a lot of cases, the only way you can really be sure you're happy with it is to try writing it one way—and if you don't like it, change it.

Plus, it's easy to make **syntax errors** in your code. A syntax error means that you wrote something that isn't valid C# code, like using a keyword or symbol incorrectly, or using a name that doesn't exist. For example, if you enter an extra **}** closing curly brace at the end of a method and then try to run it, Visual Studio will give you an error telling you that it can't **build** your code (which is what it does to turn your C# code into something that your computer can actually execute).

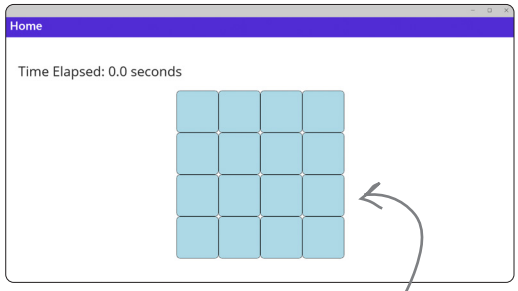
What does all that mean?

It means that you'll ***run your apps all the time, over and over again***. And that's perfectly fine! It's absolutely okay to run your app after even a tiny change, just to see what that change did. The more comfortable you are running your app, the more you'll feel like you can experiment and make changes—and the more fun you'll have with it.

So go ahead and **run your app now**. Make sure it starts out with the "Play again?" button visible and the animal buttons invisible. Click the "Play again?" button and make sure it hides itself and shows the animal buttons. When you're done, close the app (or stop it from inside Visual Studio).



When you start the app, you'll see the "Play again" button but no animal buttons.



Click the "Play again?" button to cause it to disappear and the animal buttons to appear.

When you enter your C# code, even tiny errors can make a big difference.

Some people say that you truly become a developer after the first time you've spent hours tracking down a misplaced period. Case matters: `AnimalButtons` is different from `animalButtons`. Extra commas, semicolons, parentheses, etc. can break your code—or, worse, change your code so that it still builds but does something different than what you want it to do. The IDE's ***AI-assisted IntelliSense and IntelliCode features*** *can help you avoid those problems…but it can't do everything for you. It's up to you to make sure your code is right—and that it does what* ***you*** *expect it to do.*

Add animals to your buttons

This game won't be much fun without animals to click on. Let's update the "Play again?" button's event handler method to set up the buttons with eight pairs of emojis positioned randomly on the buttons.

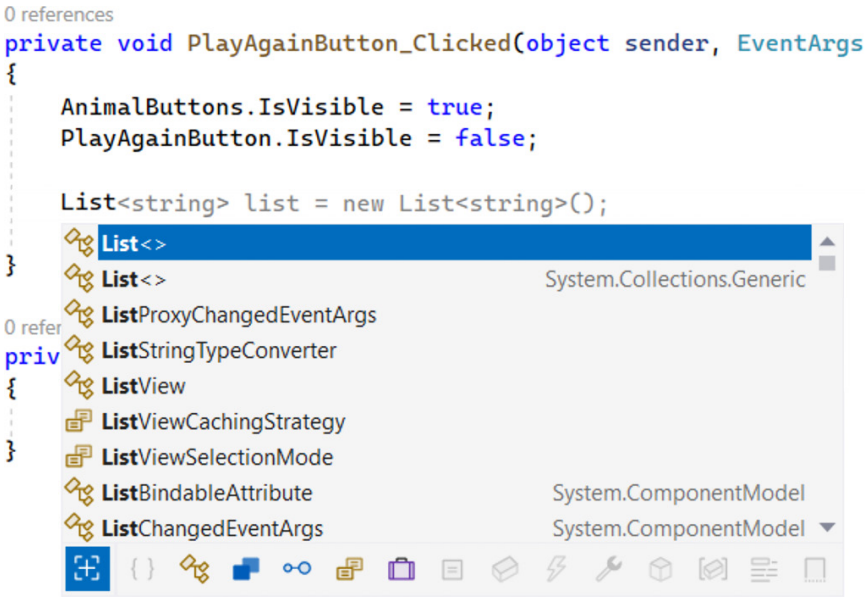
1 Start creating a List of animal emoji.

Your event handler method needs to start with eight pairs of emoji, so you're going to write a statement that creates them and stores them in something called a List (you'll learn a lot more about that in Chapter 8).

Start typing this line of code right after the statements that you just added—but ***don't*** end it with a semicolon, because that's not the end of the statement yet:

```
List<string> animalEmoji = new List<string>()
```

While you're typing, you'll see IntelliSense windows pop up to help you enter that code. You might even get an IntelliCode suggestion that matches that code exactly, except with a semicolon at the end like this:



You should use the IntelliSense and IntelliCode suggestions if they make sense. In this case, if you get do get a matching suggestion and take it, make sure you **delete the semicolon** at the end of the line.

Your PlayAgainButton_Clicked event handler method should now look like this:

```
private void PlayAgainButton_Clicked(object sender, EventArgs e)
{
    AnimalButtons.IsVisible = true;
    PlayAgainButton.IsVisible = false;

    List<string> animalEmoji = new List<string>()
```

Your statement isn't done yet! So don't add a colon to the end.

Some people think the plural emoji is emoji, others think it's emojis. We went with emoji—but both ways are fine!

2 **Add a pair of animal emoji to your list.**

Your C# statement isn't done yet. Make sure your cursor is placed just after the) at the end of the line, then type an opening curly bracket {—the IDE will add the closing one for you, and your cursor will be positioned between the two brackets. **Press Enter**—the IDE will add line breaks for you automatically:

```
List<string> animalEmoji = new List<string>()
{
    |
}
```

Now let's add **8 pairs of animal emoji**. You can find emoji by going to your favorite emoji website (for example, https://emojipedia.org/nature) and copying a individual emoji characters. Alternately…

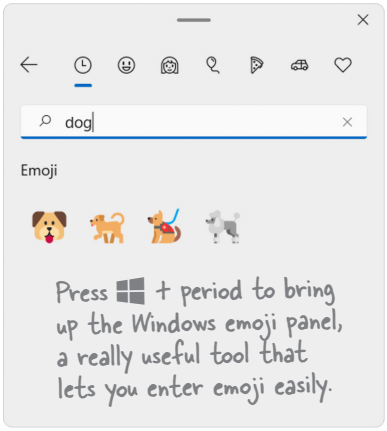
If you're using Windows, use the **Windows emoji panel** (press Windows logo key + period). If you're using a Mac, use the Character Viewer panel (choose "Show Emoji & Symbols" from the Input menu).

Go back to your code add a double quote " then paste the character—we used an octopus—followed by another " and a comma, a space, another ", the same character again, and one more " and comma:

```
List<string> animalEmoji = new List<string>()
{
    "🐙", "🐙",
```

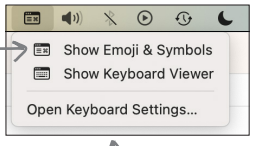
How to enter emoji

If you're using Windows, use the **emoji panel** by pressing Windows logo key + period. Use the search box to search for a specific animal. When you find the emoji you want to enter, click on it to enter it as if you'd typed it.



Press + period to bring up the Windows emoji panel, a really useful tool that lets you enter emoji easily.

If you're using a Mac, use the **Character Viewer panel**, which you can view by choosing "Show Emoji & Symbols" from the Input menu in the menu bar . Use the search box to search for a specific animal. When you find the emoji you want to enter, click on it to enter it as if you'd typed it.



If you don't see the Input menu in your menu bar, open System Settings and search for "input menu" – there's an option that you can turn on to show the input menu in the menu bar.

3 Add the rest of the animal emoji pairs to your list.

Then do the same thing for seven more emoji so you end up with **eight pairs of animal emoji between the brackets**. We added a blowfish, elephant, whale, camel, brontosaurus, kangaroo, and porcupine—but you can add whatever animals (or other emoji!) that you want.

Add a `;` after the closing curly bracket. This is what your statement should look like now:

A List is a collection that stores a set of values in order. You'll learn all about collections in Chapters 8 and 9.

You're using the "new" keyword to create your List, and you'll learn about that in Chapter 3.

```
List<string> animalEmoji = new List<string>()
{
    "🐡", "🐡",
    "🐘", "🐘",
    "🐋", "🐋",
    "🐪", "🐪",
    "🦕", "🦕",
    "🦘", "🦘",
    "🦔", "🦔",
};
```

Be really careful with the quotes and commas. If you miss one, your code won't build.

4 Finish the method.

Add the rest of the code to add random animal emoji to the buttons:

You'll learn more about loops in the next chapter.

```
foreach (var button in AnimalButtons.Children.OfType<Button>())
{
    int index = Random.Shared.Next(animalEmoji.Count);
    string nextEmoji = animalEmoji[index];
    button.Text = nextEmoji;
    animalEmoji.RemoveAt(index);
}
```

This is a foreach loop. It goes through a collection (like your list of emoji) and executes a set of statements for each of item it finds.

Before you run your app, read through the code that you just added. It's okay if you don't understand everything that's going on with it yet. An important part of learning C# is starting to make the code make sense, and reading through it is a great way to do that.

Reading through C# code – even if you don't understand all of it yet – is a great way to make it all start to make sense.

5 **Make sure your code matches ours.**

Here's all of the C# code that you've added so far. We gave the parts that Visual Studio generated for you automatically a lighter color so you can see the code that you entered yourself.

```
namespace AnimalMatchingGame;
```

If you chose a different name for your project, this line will match that name.

```
public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();

        AnimalButtons.IsVisible = false;
    }

    private void PlayAgainButton_Clicked(object sender, EventArgs e)
    {
        AnimalButtons.IsVisible = true;
        PlayAgainButton.IsVisible = false;

        List<string> animalEmoji = new List<string>()
        {
            "🐙", "🐙",
            "🐷", "🐷",
            "🐵", "🐵",
            "🐸", "🐸",
            "🐪", "🐪",
            "🐘", "🐘",
            "🐓", "🐓",
            "🦔", "🦔",
        };

        foreach (var button in AnimalButtons.Children.OfType<Button>())
        {
            int index = Random.Shared.Next(animalEmoji.Count);
            string nextEmoji = animalEmoji[index];
            button.Text = nextEmoji;
            animalEmoji.RemoveAt(index);
        }
    }

    private void Button_Clicked(object sender, EventArgs e)
    {

    }
}
```

You added this line to make the animal buttons invisible when the app first starts up.

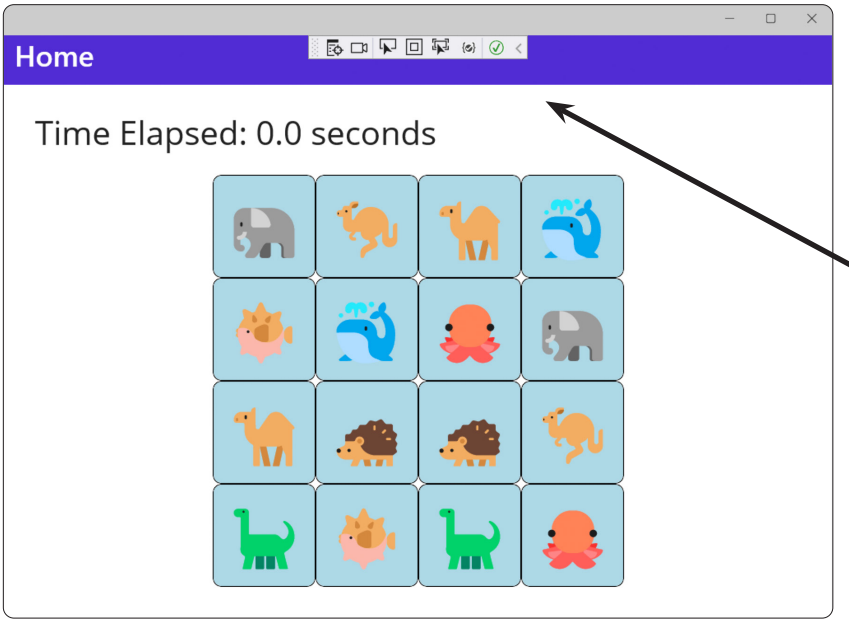
Make sure there are exactly eight matching pairs of emoji. That's part of what makes the game work.

You just added this code to add the emoji to the buttons.

Visual Studio added this empty event handler method when you added a Clicked property to the button that you copied and pasted. Make sure it's there!

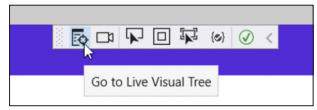
Run your app!

Run your app again. The first thing you'll see is the "Play again?" button. Click the button—you should now see eight pairs of animals in random positions:

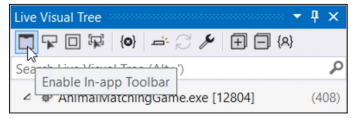


Stop it and run it again a few times. The animals should get reshuffled in a different order every time you click the "Play again?" button.

If you're using Windows, you might see the in-app toolbar hovering at the top of the window:



Click the first button in the runtime tools to bring up the Live Visual Tree panel in the IDE:



Then click the first button in the Live Visual Tree to disable the in-app toolbar.

You've set the stage for the next part that you'll add.

When you build a new game, you're not just writing code. You're also running a project. A really effective way to run a project is to build it in small increments, taking stock along the way to make sure things are going in a good direction. That way you have plenty of opportunities to change course.

This is a pencil-and-paper exercise. We included a lot of of games and puzzles like this throughout the book. You should do all of them, because there's neuroscience evidence that writing things down is an effective way to get important concepts into your brain faster.

Who Does What?

C# statement

What it does

```
List<string> animalEmoji = new List<string>()
{
    "🐙","🐙",
    "🐵","🐵",
    "🐘","🐘",
    "🐡","🐡",
    "🐫","🐫",
    "🦕","🦕",
    "🐇","🐇",
    "🐌","🐌",
};
```

Make the button display the selected emoji

Find every button in the FlexLayout and repeat the statements between the { curly brackets } for each of them

Make the "Play again?" button invisible

```
foreach (var button in AnimalButtons.Children.OfType<Button>())
{
```

Create a list of eight pairs of emoji

```
animalEmoji.RemoveAt(index);
```

Make the FlexLayout with the emoji buttons visible

```
button.Text = nextEmoji;
```

```
string nextEmoji = animalEmoji[index];
```

Pick a random number between 0 and the number of emoji left in the list and call it "index"

```
AnimalButtons.IsVisible = true;
```

Remove the chosen emoji from the list

```
int index = Random.Shared.Next(animalEmoji.Count);
```

```
PlayAgainButton.IsVisible = false;
```

Use the random number called "index" to get a random emoji from the list

Who Does What? Solution

C# statement

```
List<string> animalEmoji = new List<string>()
{
    "🐙","🐙",
    "🐵","🐵",
    "🐘","🐘",
    "🐡","🐡",
    "🦙","🦙",
    "🦒","🦒",
    "🐰","🐰",
    "🐌","🐌",
};

foreach (var button in AnimalButtons.Children.OfType<Button>())
{

animalEmoji.RemoveAt(index);

button.Text = nextEmoji;

string nextEmoji = animalEmoji[index];

AnimalButtons.IsVisible = true;

int index = Random.Shared.Next(animalEmoji.Count);

PlayAgainButton.IsVisible = false;
```

What it does

Make the button display the selected emoji

Find every button in the FlexLayout and repeat the statements between the { curly brackets } for each of them

Make the "Play again?" button invisible

Create a list of eight pairs of emoji

Make the FlexLayout with the emoji buttons visible

Pick a random number between 0 and the number of emoji left in the list and call it "index"

Remove the chosen emoji from the list

Use the random number called "index" to get a random emoji from the list

Here's another pencil–and paper exercise. Take a few minutes to do it!

MINI Sharpen your pencil

Here's a pencil-and-paper exercise that will help you **really start to understand** your C# code.

1. Take a piece of paper and turn it on its side so it's in landscape orientation, and draw a vertical line down the middle.
2. Write out the entire SetUpGame method by hand on the left side of the paper, leaving space between each statement. (You don't need to be accurate with the emoji.)
3. On the right side of the paper, write each of the "what it does" answers above next to the statement that it's connected to. Read down both sides—it should all start to make sense.

I'M NOT SURE ABOUT THESE "SHARPEN YOUR PENCIL" AND MATCHING EXERCISES. ISN'T IT BETTER TO ***JUST GIVE ME THE CODE*** TO TYPE INTO THE IDE?

Working on your code comprehension skills will make you a better developer.

The pencil-and-paper exercises are **not optional**. They give your brain a different way to absorb the information. But they do something even more important: they give you opportunities to ***make mistakes***. Making mistakes is a part of learning, and we've all made plenty of mistkaes (you may even find one or two typos in this book!). Nobody writes perfect code the first time—really good programmers always assume that the code that they write today will probably need to change tomorrow. In fact, later in the book you'll learn about *refactoring*, or programming techniques that are all about improving your code after you've written it.

We're serious—take the time to do the pencil–and–paper exercises. They're carefully designed to reinforce important concepts, and they're the fastest way to get the ideas in this book into your brain.

We'll add bullet points like this to give a quick summary of many of the ideas and tools that you've seen so far.

Bullet Points

- Visual Studio is Microsoft's **IDE**—or *integrated development environment*—that simplifies and assists in editing and managing your C# code files.
- .**Console apps** are cross-platform apps that use text for input and output.
- .NET **MAUI** (or .NET Multi-platform App UI) is a cross-platform framework for building visual apps in C#.
- MAUI user interfaces are designed in **XAML** (eXtensible Application Markup Language), an XML-based markup language that uses tags and properties to define controls in a user interface.
- MAUI apps are made up of **pages** that show **controls**.
- The FlexLayout control contains other controls and wraps them so they display on the page.
- The IDE's Properties window makes it easy to edit the properties of your controls like the text or font size.
- C# is made up of **statements** grouped into **methods**.
- An **event handler method** gets executed when specific events—like button clicks—happen.
- Visual Studio's AI-assisted **IntelliSense** and **IntelliCode** help you enter code more quickly.

You can use Git to save all of your code, and Visual Studio will help make it easy.

You're going to write a lot of code in this book! Wouldn't it be great if there was a convenient place to put that code so you can always go back to it?

We bet that you'll write some apps that you really like, and you'll want to share them with your friends so they can see the great things you've built.

Do you have a desktop and a laptop? A computer at home and at an office? Wouldn't it be great if you could start a project on one computer, then finish it on another one?

Imagine you're working on a project. You've spent hours getting the code right, and you're really happy with it. Then you make a few changes, and... oh no! Something went completely wrong, your code is broken, and you don't remember exactly what you changed. It would be great if you see a history of all the changes you made, right?

Git can help you do all of those things!

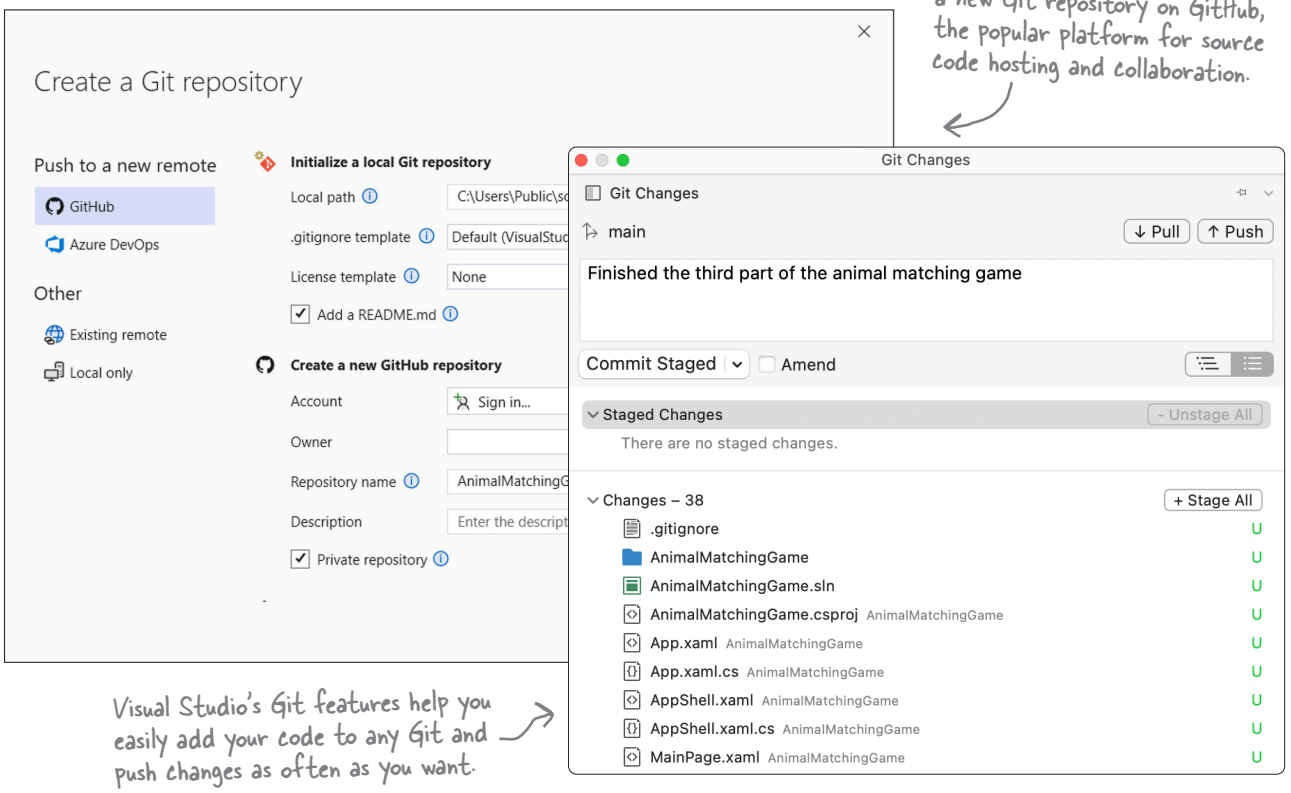
Here are just a few things Git can do for you

- It can save your files somewhere that you can access them form anywhere, any time
- It lets you save snapshots of your work so you can go back and see exactly what changed
- It lets you share your code with anyone (or keep it private!)
- It lets a group of people collaborate on a project together—so if you're learning C# with your friends, you can all work on code together

Visual Studio makes it easy to use Git

Git is a really powerful and flexible tool that can help you save, manage, and share your the code and files for all of your projects. It can also be complex and confusing at times! Luckily, Visual Studio has **built-in Git suupport** that takes care of the complexity. It helps you with Git, so you can concentrate on your code.

Visual Studio can help you create a new Git repository on GitHub, the popular platform for source code hosting and collaboration.

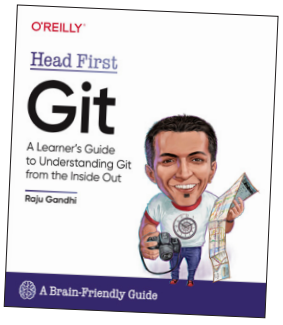


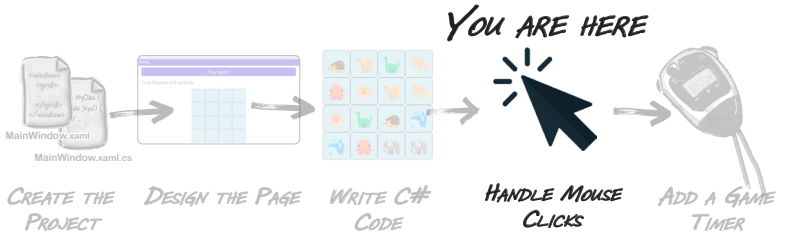
Visual Studio's Git features help you easily add your code to any Git and push changes as often as you want.

We recommend that you create a GitHub account and use it save the code for each of the projects in this book. That will make it easy for you to go back and revisit past projects any time!

Our free Head First C# Guide to Git PDF gives you a simple, step-by-step guide to saving your code in Git with Visual Studio. Download it from https://github.com/head-first-csharp/fifth-edition/

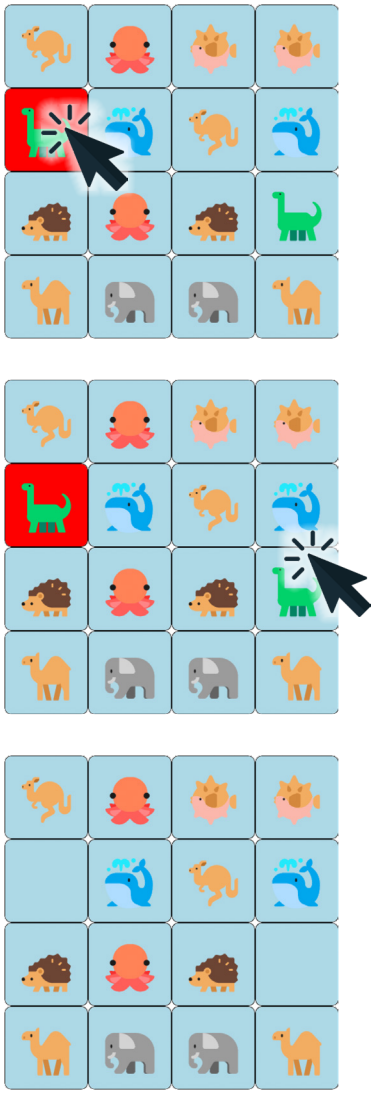
We'll give you everything you need to use Visual Studio to save and share your projects. But there is a lot more that you can do with Git, especially if you're working with large teams! If you're fascinated by what you see and want to do a deep dive into Git, check out Head First Git by Raju Gandhi.





Add C# code to handle mouse clicks

You've got buttons with random animal emoji. Now you need them to do something when the player clicks them. Here's how it will work:



The player clicks the first button.

The player clicks buttons in pairs. When they click the first button, the game keeps track of that particular button's animal. The button that the player changes color, so they can see what animal they clicked on.

The player clicks the second button.

The game looks at the animal on the second button and compares it against the first one they clicked on. The game compares its animal against the animal on one that it kept track of from the first click.

The game checks for a match.

If the animals ***match***, the game goes through all of the emoji in its list of shuffled animal emoji. It finds any emoji in the list that match the animal pair the player found and replaces them with blanks.

If the animals ***don't match***, the game doesn't do anything.

In ***either case***, it resets its last animal found so it can do the whole thing over for the next click.

The game repeats this until all eight pairs of animals are matched.

Sharpen your pencil

When you added the Clicked event handler to your animal button, Visual Studio **automatically added a method called Button_Clicked** to MainPage.xaml.cs. Here's the code that will go into that method. Before you add this code to your app, read through it and try to figure out what it does.

We've asked you a few questions about what the code does. Try writing down the answeres. ***It's okay if you're not 100% right!*** The goal is to start training your brain to recognize C# as something you can read and make sense of.

```
Button lastClicked;
bool findingMatch = false;
int matchesFound;

private void Button_Clicked(object sender, EventArgs e)
{
  if (sender is Button buttonClicked)
  {
    if (!string.IsNullOrEmpty(buttonClicked.Text) && (findingMatch == false))
    {
      buttonClicked.BackgroundColor = Colors.Red;
      lastClicked = buttonClicked;
      findingMatch = true;
    }
    else
    {
      if ((buttonClicked != lastClicked) && (buttonClicked.Text == lastClicked.Text))
      {
        matchesFound++;
        lastClicked.Text = "";
        buttonClicked.Text = "";
      }
      lastClicked.BackgroundColor = Colors.LightBlue;
      buttonClicked.BackgroundColor = Colors.LightBlue;
      findingMatch = false;
    }
  }

  if (matchesFound == 8)
  {
    matchesFound = 0;
    AnimalButtons.IsVisible = false;
    PlayAgainButton.IsVisible = true;
  }
}
```

1. What does `matchesFound` do?

2. What do these three lines of code do?

3. What does this block of code do?

3. What do the last 6 lines of the method starting with `if (matchesFound == 8)` and going to the end do?

Sharpen your pencil Solution

When you added the Clicked event handler to your animal button, Visual Studio **automatically added a method called Button_Clicked** to MainPage.xaml.cs. Here's the code that will go into that method. Before you add this code to your app, read through it and try to figure out what it does.

We've asked you a few questions about what the code does. Try writing down the answeres. ***It's okay if you're not 100% right!*** The goal is to start training your brain to recognize C# as something you can read and make sense of.

```
Button lastClicked;
bool findingMatch = false;
int matchesFound;

private void Button_Clicked(object sender, EventArgs e)
{
  if (sender is Button buttonClicked)
  {
    if (!string.IsNullOrEmpty(buttonClicked.Text) && (findingMatch == false))
    {
      buttonClicked.BackgroundColor = Colors.Red;
      lastClicked = buttonClicked;
      findingMatch = true;
    }
    else
    {
      if ((buttonClicked != lastClicked) && (buttonClicked.Text == lastClicked.Text))
      {
        matchesFound++;
        lastClicked.Text = "";
        buttonClicked.Text = "";
      }
      lastClicked.BackgroundColor = Colors.LightBlue;
      buttonClicked.BackgroundColor = Colors.LightBlue;
      findingMatch = false;
    }
  }

  if (matchesFound == 8)
  {
    matchesFound = 0;
    AnimalButtons.IsVisible = false;
    PlayAgainButton.IsVisible = true;
  }
}
```

1. What does `matchesFound` do?

It keeps track of the number of pairs of animals the player found, so the game can end when they found all 8 pairs.

2. What do these three lines of code do?

These lines are run when the first button of a potential match. They change its color to red and keep track of it.

3. What does this block of code do?

This block of code is run when the player clicks on the second button in the pair. If the animals match, it adds one to matchesFound and blanks out the animals on both buttons. It also resets the color of the first button back, and gets set for the player to click the first button in a pair again.

3. What do the last 6 lines of the method starting with `if (matchesFound == 8)` and going to the end do?

If matchesFound equals 8, the player found all 8 pairs of animals. Ween that happens, these lines reset the game by setting matchesFound back to zero, hiding the animal buttons, and showing the "Play again?" button so the player can start a new game by clicking the "Play again?" button.

Enter the code for the event handler

Did you do the "Sharpen your pencil" exercise? If not, take a few minutes and do it—you may not understand 100% of the code in the Button_Clicked event handler method yet, but you should at least have a basic sense of what's going on. And, more importantly, you've had a chance to look at it closely enough so that it should be familiar.

That familiarity will make it easier to **use Visual Studio to type the code into the method**. Stop your app if it's running, then MainPage.xaml.cs, find the Button_Clicked event handler method that Visual Studio added for you, and click on the line between its opening { and closing } curly brackets.

Now **start typing the code** line by line. If you haven't used an IDE like Visual Studio to write code before, it may be a little weird seeing its IntelliSense and IntelliCode suggestions pop up. Use them if you can—the more you get used to them, the faster and easier it will be to write code later on in the book.

You need to be really careful when you're entering code, because if your opening parentheses or brackets don't have matches, or if you miss a semicolon at the end of a statement, your code won't build. Luckily, Visual Studio have a lot of features to help you write code that builds:

- When you enter "if" it automatically adds the opening and closing parentheses () so you don't accidentally leave them out.
- If you put your cursor in front of an opening parentehse or bracket, it will highlight the closing one so you can easily see its match.
- A lot of the time, when you enter code that has problems—like writing `matchesFnd` instead of `matchesFound`, for example—it will often point out the error by drawing a red squiggly line underneath it.

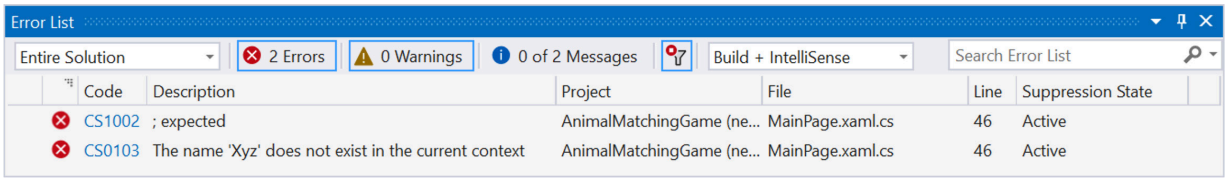
IDE Tip: The Error List

An operating system like Windows, macOS, Android, or iOS can't run C# code. That's why Visual Studio has to **build** your code, or turn it into a **binary** (a file that the operating system can run). Let's do an experiment and ***break your code***.

Go to the first line of code in your Button_Clicked method. Press Enter twice, then add this on its own line: `Xyz`

Check the bottom of the code editor again—you'll see an icon that looks like this: 2 or 2. If you don't see the icon, choose Build Solution from the Build menu to tell Visual Studio to try to build your code.

Click the icon (or choose Error List from the View menu) to open the Error List window. You'll see two errors in the window (if you're using a Mac it's called Errors and not Error List, and it looks a little different, but it displays the same information):



Visual Studio displayed these errors because `Xyz` is not valid C# code, and the errors prevent it from building your app. Your code won't run with those errors, so go ahead and delete the `Xyz` line that you added and build your app again.

If there are no other errors in your code, the Error List should be empty, and you'll see an icon that looks like this at the bottom of the Visual Studio window: No issues found or Build successful. – that tells you that your app builds.

Run your app and find all the pairs

Try running your app. If you entered all of the code correctly, it should start up and show you the "Play again?" button. Click the button to see a random list of animals. Then click each pair of animals one by one—each pair will disappear after you click it. Once you click the last pair of animals, the buttons will disappear and you'll see the "Play again?" button again.

If your game doesn't work the way it should or you don't see the bug on this page, go back and check the code you entered against the code in the book. It's really easy to overlook a typo. Finding those issues is a good use of your time, because spotting errors in your code is a really good developer skill to work on.

Try experimenting with your app. Click mismatched pairs. Click in the window but outside hte buttons. Click on the "Time elapsed" label. Click an empty button. Is your app working?

Uh-oh – there's a bug in your code

If you typed in all of the code correctly, you may have noticed a problem. Start your app, click the "Play again?" button to show the random animals, and click on a pair to make the animals disappear from their buttons. Now **click the blank button seven times**. Wait, what happened? Did the animal buttons disappear and the "Play again?" button appeared, as if you'd wont he game? That's not supposed to happen! Your game has a bug.

Don't worry, this bug is not your fault!

We left that bug in your code on purpose. You're going to be writing a lot of code throughout this book. Every chapter has several projects for you to work on… and there are opportunities for bugs in every one of those projects. Finding and fixing bugs is a normal and healthy part of writing code—and a really valuable skill for you to practice.

Every bug is caused by a problem in the code, so the first step in fixing a bug is figuring out what's causing it.

When you find a bug, you need to sleuth it out

Every bug is different. Code can break in many different ways. But there's one thing all bugs have common: every one of them ***is caused by a problem in the code***. So when there's a bug, your job is to figure out what's causing it, because you can't fix the problem until you know why it's happening.

If you've ever read a mystery novel or watched a detecive show, you know to solve a mystery: you need to **find the culprit**. So let's do that right now. It's time to put on your Sherlock Holmes cap, grab your magnifying glass, and **sleuth out what's causing the bug**.

Finding and fixing bugs is one part typing, nine parts thinking... and 100% guaranteed to make you a better developer. That's what these "Sleuth it Out" sections are all about.

Sleuth it Out

The Case of the Unexpected Match

You've probably heard the word "bug" before.

You might have even said something like this to your friends at some point in the past: "That game is really buggy, it has so many glitches." Every bug has an explanation, and everything in your program happens for a reason… but not every bug is easy to track down. That's why we'll include tips for sleuthing out bugs throught the book, starting with this "Sleuth it out" section.

Every bug has a culprit

Bugs are weird. They're what happens when your code does something you didn't expect it to do.

But bugs are also normal. Every developer spends time finding and fixing bugs. It's a normal part of writing code. You're going to write code that doesn't do what you expect it to. And when you do, the first thing you need to do is ***figure out what's causing the bug***.

The first step in finding a bug is thinking about what might have caused it

Sherlock Holmes once said, "Crime is common. Logic is rare. Therefore it is upon the logic rather than upon the crime that you should dwell." That's great advice for figuring out what caused the bug. Don't get frustrated because your app doesn't do what you want (that's dwelling on the crime!). Instead, think about the logic of the situation. So let's look at the code and figure out what's going on.

Read the code carefully and search for clues

We know that all of the code for handling mouse clicks is in the Button_Clicked event handler that you just added. So let's go back to the code and see if we can find clues about what went wrong.

Luckily, **you did that "Sharpen your pencil" exercise**. You looked closely at the code in the Button_Clicked event handler method to understand it. (If you haven't done that exercise yet, go back and do it now!)

Based on what we found in the "Sharpen your pencil" exercise, we already know a few things about the code:

- The event handler uses matchesFound to keep track of the number of pairs of animals the player found, so the game can end when they found all 8 pairs.
- There a part of the event handler that checks if the animals on the two buttons that the player clicked on match each other. If they do match, it adds one to matchesFound and blanks out both buttons.
- If matchesFound equals 8, the player found all 8 pairs of animals.There's code at the end of the event handler that checks to see if matchesFound is equal to 8, and if that's true is it resets the game.

Those are the important clues that will help us find and fix the bug. Before you go on, can you sleuth out what's causing the game to end early if you keep clicking a button that's already been cleared?

Why did the bug happen?

Let's think about those three clues for a minute. Here's what we know.

- The game uses matchesFound to keep track of the number of pairs of animals the player found.
- If the player clicks on a pair, the game increases matchesFound by 1 and blanks out the buttons the player clicked on.
- When matchesFound reaches 8, the game resets itself.

So what are these clues telling us? There's one conclusion that we can draw from these clues:

Somehow matchesFound is being increased by 1 when the player clicks on a button that's already blanked out.

Which means we have a starting point: the code that increases matchesFound by 1.

Go back to the scene of the crime

Here's the part of the code that increases matchesFound – the specific line that does that is in **`boldface`**:

```
if ((buttonClicked != lastClicked) && (buttonClicked.Text == lastClicked.Text))
{
    matchesFound++;
    lastClicked.Text = "";
    buttonClicked.Text = "";
}
```

This statement uses the ++ operator to increase the value of matchesFound by 1. You'll learn about ++ and other operators in the next chapter.

The first line of code in the statements that we just showed you is an **if statement**, which checks if something and executes statements if it's true. In this case, if the player clicked a different button than the first one in the pair (that's what "buttonClicked != lastClicked" checks for) and if the animals on those two buttons match ("buttonClicked.Text == lastClicked.Text"), it increases matchesFound by 1 and blanks out both buttons.

This is where things went wrong—which means it's also where we can fix the bug. We just need to find a way to keep matchesFound from getting increased by 1 **if the player clicked a button that's already blank**.v

We found the culprit, so now we can fix the bug

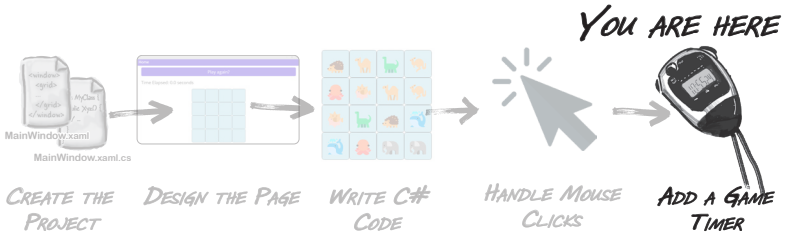
Position your cursor between the last two closing parentheses `))` in the if statement and press enter to add a line. Then enter the following code: `&& (buttonClicked.Text != "")`

Here's what your code should now look like:

```
if ((buttonClicked != lastClicked) && (buttonClicked.Text == lastClicked.Text)
    && (buttonClicked.Text != ""))
{
    matchesFound++;
    lastClicked.Text = "";
    buttonClicked.Text = "";
}
```

Adding this code to your if statement causes it to make sure the button that the player clicked on is not blank before adding 1 to matchesFound.

Once you've edited the if statement, run your app again. Now the bug should be fixed.



Finish the game by adding a timer

Our animal match game will be more exciting if players can try to beat their best time. We'll add a **timer** that "ticks" after a fixed interval by repeatedly calling a method.

Let's add some excitement to the game! The time elapsed since the game started will appear at the bottom of the window, constantly going up, and only stopping after the last animal is matched.

Timers "tick" every time interval by calling methods over and over again. You'll use a timer that starts when the player starts the game and ends when the last animal is matched.

Add a timer to your game's code

In this last part of your project, you'll add a timer to your game to make it more exciting. It will keep track of the time elapsed (in tenths of seconds), starting when the player clicks the "Play again?" button and stopping when they find the last match.

Do this!

① Add a line of code to the end of the PlayAgainButton_Clicked event handler to start a timer.

Go to the very end of the PlayAgainButton_Clicked event handler. There are two closing curly brackets } at the end of the method on separate lines. Add three lines between the brackets, then add the following line of code into that space that you created:

```
    foreach (var button in AnimalButtons.Children.OfType<Button>())
    {
        int index = Random.Shared.Next(animalEmoji.Count);
        string nextEmoji = animalEmoji[index];
        button.Text = nextEmoji;
        animalEmoji.RemoveAt(index);
    }

    Dispatcher.StartTimer(TimeSpan.FromSeconds(.1), TimerTick);
}
```

The line of code that you just added causes your app to **start a timer** that executes a method called TimerTick every .1 a second.

② Examine the error and click on "TimerTick" in the code you just added.

You just added a line of code to start a timer that "ticks" every tenth of a second. Every time it ticks, it calls a method called TimerTick. But hold on—your C# code doesn't have a TimerTick method. If you try to build your code, you'll see an error in the Error List window:

CS0103 The name 'TimerTick' does not exist in the current context

And there will be a red squiggly line underneath TimerTick in the line of code that you added. Click on "TimerTick" in the C# code—when you click on it, Visual Studio will display an icon shaped like a light bulb in the left margin.

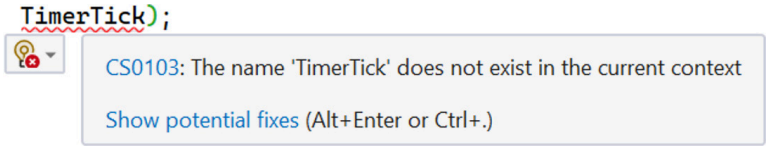
```
35              button.Text = nextEmoji;
36              animalEmoji.RemoveAt(index);
37          }
38
39          Dispatcher.StartTimer(TimeSpan.FromSeconds(.1), TimerTick);
40      }
41
```

When you click on TimerTick in the C# code, Visual Studio displays this icon. It looks a little different on a Mac, but works the same way.

The red squiggly line tells you under TimerTick that there's an error here.

③ Use Visual Studio to generate a new TimerTick method.

The code that you added has an error because it refers to a method called TimerTick that doesn't exist. When you clicked on it, a light bulb icon showed up in the left-hand margin. If you hover over it, you can see an error message and light bulb icon directly underneath it as well:



Clicking the light bulb icon to brings up the **Quick Actions menu**, which gives you some suggested potential fixes for the error. You can also press Alt+Enter or Ctrl+. on Windows or ⌥+Return on a Mac to bring up the menu:

The first option in the Quick Actions menu should be "Generate method 'TimerTick'" – and if you select that option, you'll see a preview to the right. **Choose that option.**

Visual Studio will ***generate the TimerTick method for you***. Look through your C# code in MainPage.xaml.cs and find the TimerTick method that Visual Studio added:

```
private bool TimerTick()
{
    throw new NotImplementedException();
}
```

When your C# code has errors, Visual Studio sometimes has suggestions for potential fixes that can generate code to fix the error.

Finish the code for your game

In this last part of your project, you'll add a timer to your game to make it more exciting. It will keep track of the time elapsed (in tenths of seconds), starting when the player clicks the "Play again?" button and stopping when they find the last match.

Do this!

Add a field to hold the time elapsed

Find the first line of the TimerTick method that you just generated. Place your mouse cursor at the beginning of the line, then press enter twice to add two spaces above it.

Add this line of code:

```
int tenthsOfSecondsElapsed = 0;
```

This is a field. You'll learn more about how fields work in Chapter 3.

```
private bool TimerTick()
```

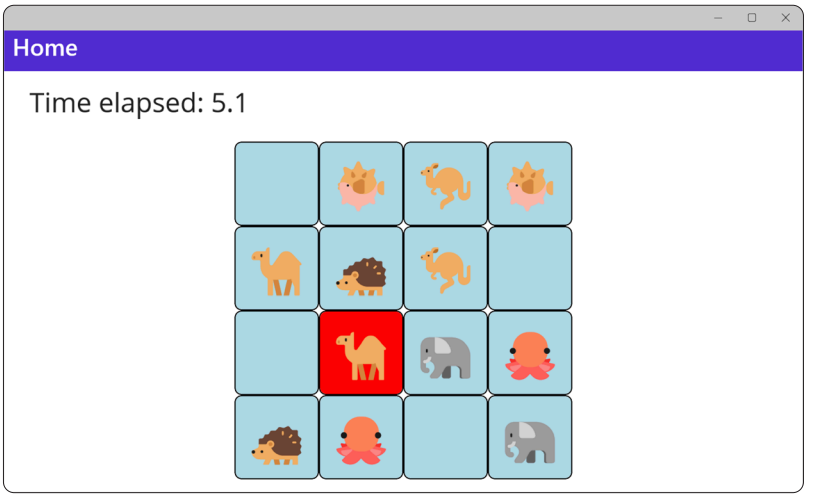
Finish your TimerTick method

Now you have everything you need to finish the TimerTick method. Here's the code for it:

```
private bool TimerTick()
{
    if (!this.IsLoaded) return false;

    tenthsOfSecondsElapsed++;

    TimeElapsed.Text = "Time elapsed: " +
        (tenthsOfSecondsElapsed / 10F).ToString("0.0s");

    if (PlayAgainButton.IsVisible)
    {
        tenthsOfSecondsElapsed = 0;
        return false;
    }

    return true;
}
```

We put an extra line break in this statement so it would fit on the page int he printed book, but you can put it all on one line if you want.

Run your game. Now the timer works!



Your TimerTick Method Up Close

Let's take a closer look at your TimerTick method to see how it, well, ticks. It has a total of seven statements, and each of them is important.

```
private bool TimerTick()
{
    if (!this.IsLoaded) return false;

    tenthsOfSecondsElapsed++;

    TimeElapsed.Text = "Time elapsed: " +
        (tenthsOfSecondsElapsed / 10F).ToString("0.0s");

    if (PlayAgainButton.IsVisible)
    {
        tenthsOfSecondsElapsed = 0;
        return false;
    }

    return true;
}
```

If you close your app the timer could still tick after the TimeElapsed label disappears, which could cause an error. This statement keeps that from happening.

The timer ticks every 10th of a second. Adding 1 to this field keeps track of how many of those 10ths have elapsed.

This statement updates the TimeElapsed label with the latest time, dividing the 10ths of second by 10 to convert it to seconds.

If the "Play Again" button is visible again, that means the game is over the timer can stop running. The if statement runs the next two statements only if the game is running.

We need to reset the 10ths of seconds counter so it starts at 0 the next time the game starts.

This statement causes the timer to stop, and no other statements in the method get executed.

This statement is only executed if the "if" statement didn't find the "Play again?" button visible. It tells the timer to keep running.

One last thing about the timer. The timer you used is guaranteed to fire no more than once every 10th of a second, but it may fire a little less frequently than that—which means the timer in the game may acutally run a little slow. For this game, that's absolutely fine!

Even better ifs...

Your game is pretty good. Nice work! Every game—in fact, pretty much every program—can be improved. Here are a few things that we thought of that could make the game better:

- Add different kinds of animals so the same ones don't show up each time.
- Keep track of the player's best time so they can try to beat it.
- Make the timer count down instead of counting up so the player has a limited amount of time.

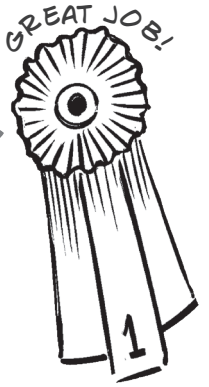
MINI

Sharpen your pencil

Can you think of your own "even better if" improvements for the game? This is a great exercise—take a few minutes and write down at least three improvements to the animal matching game.

We're serious—take a few minutes and do this. Stepping back and thinking about the project you just finished is a great way to seal the lessons you learned into your brain.

Congratulations—you built a game, but you did more than that! You took the time to really understand how it works, and that's a very important step in getting comfortable with C# concepts.



Did you save your code Git?

If you did, this is a great time to commit all of your changes and push it to the repository!

And if you still haven't, take a few minutes and check out our free ***Head First C# Guide to Git*** PDF. It gives you step-by-step instructions for keeping your code safe in Git.

Download it today from our own GitHub page: https://github.com/head-first-csharp/fifth-edition

Bullet Points

- An **event handler** is a method that your application calls when a specific event like a mouse click happens.
- Visual Studio makes it easy to **add and manage** your event handler methods.
- The IDE's **Error List window** shows any errors that prevent your code from building.
- A **timer** calls a method over and over again on a specified interval.
- **foreach** is a kind of loop that iterates through a collection of items.
- When you have a bug in your code, the first thing to do is try to **figure out what's causing it**.
- Bugs are normal, and sleuthing out bugs is an **important developer skill** that you'll work on throughout this book.
- Visual Studio makes it really easy to use **source control** to back up your code and keep track of all changes that you've made.
- You can commit your code to a remote **Git repository**. We use GitHub for the repository with the source code for all of the projects in this book.

2 variables, statements, and methods

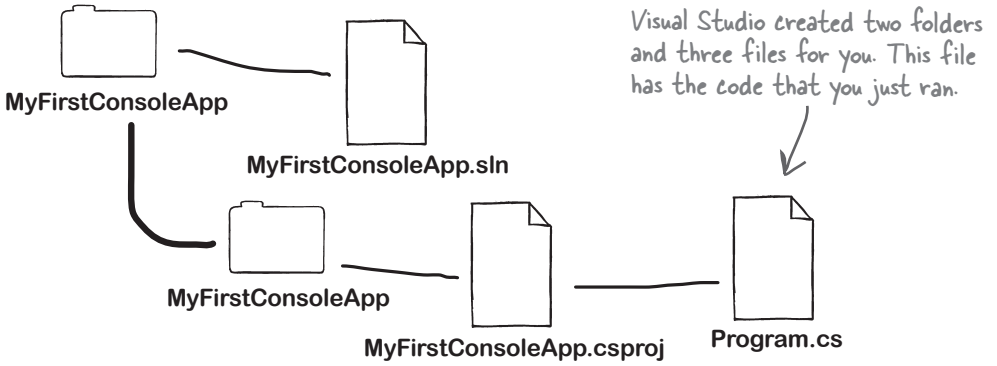
Dive into C# code

You're not just an IDE user. You're a developer.

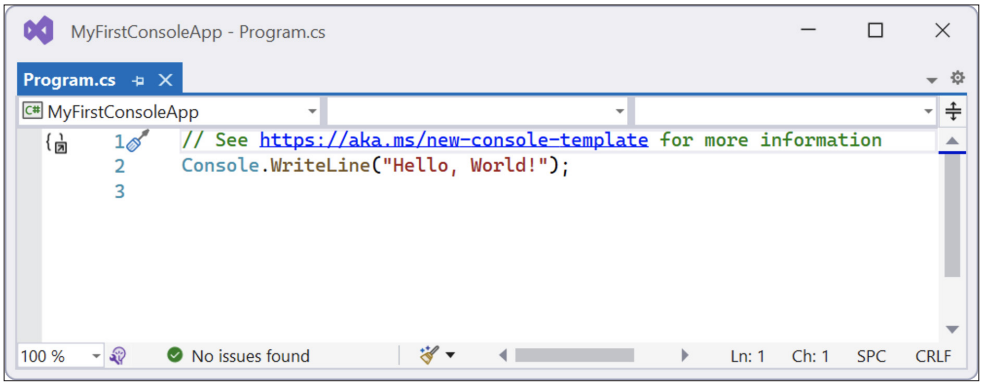
You can get a lot of work done using the IDE, but there's only so far it can take you. Visual Studio is one of the most advanced software development tools ever made, but a **powerful IDE** is only the beginning. It's time to ***dive* in to C# code**: how it's structured, how it works, and how you can take control of it…because there's no limit to what you can get your apps to do.

Take a closer look at the files in your console app

In the last chapter, you created a new C# Console App project and named it MyFirstConsoleApp. When you did that, Visual Studio created two folders and three files.



Let's take a closer look at the Program.cs file that it created. Open it up in Visual Studio:



This is a screenshot of Visual Studio for Windows. If you're using macOS the screen will look a little different, but the code will be the same.

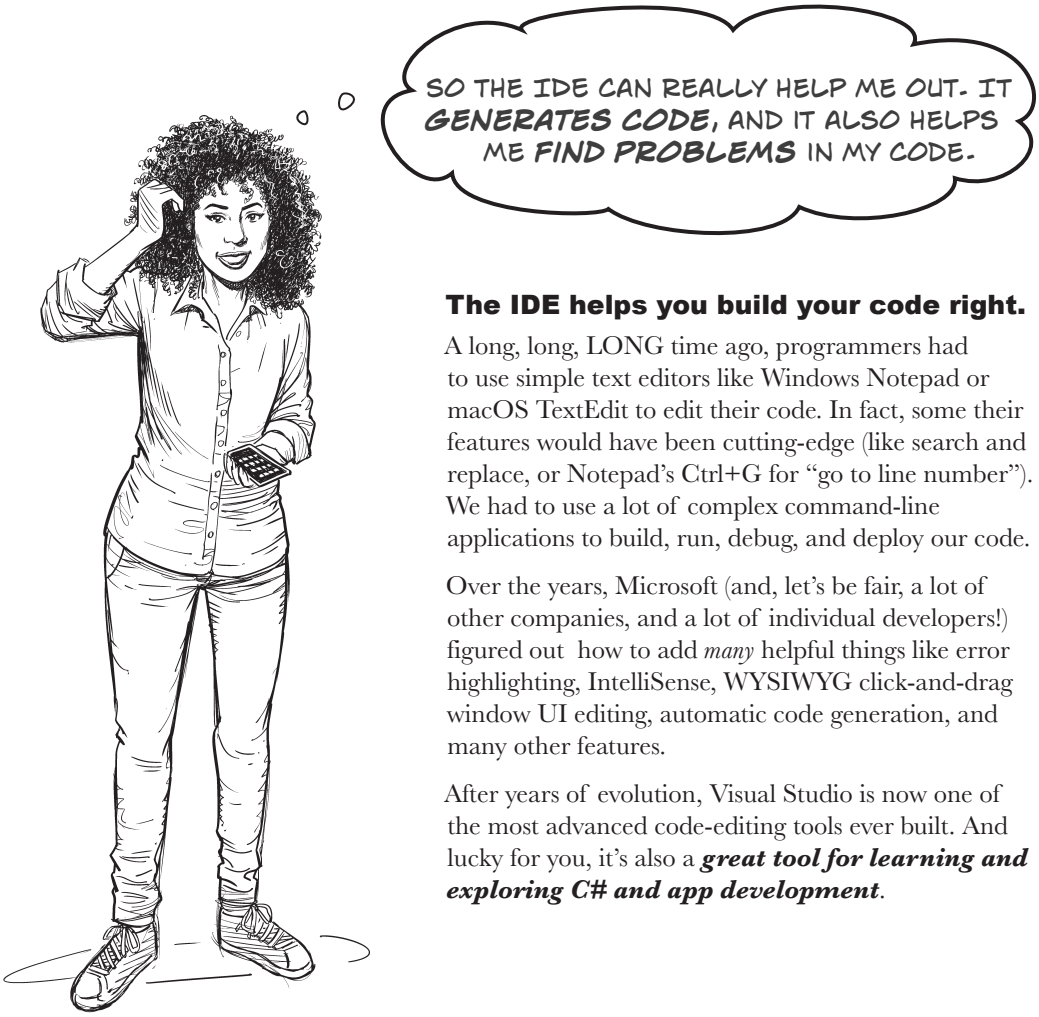
A statement performs one single action

A **console app** is an app with a text-only user interface. All its input and output goes to a console, like the Windows command prompt or MacOS Terminal.

Your app has two lines:

1. The first line is a **comment**. Comments start with two forward slashes **//** and everything after those slashes is ignored. You can can use comments to write notes about the code.
2. The second line is a **statement**. In this case, it's a Console.WriteLine statement, which writes a line of text. Statements are what actually make your code do things.

When you run your app, it starts with the first statement, and keeps executing statements until it runs out, and since it's a console app you'll see its output in a console window. Once it executes the last statements, the app exits.



The IDE helps you build your code right.

A long, long, LONG time ago, programmers had to use simple text editors like Windows Notepad or macOS TextEdit to edit their code. In fact, some their features would have been cutting-edge (like search and replace, or Notepad's Ctrl+G for "go to line number"). We had to use a lot of complex command-line applications to build, run, debug, and deploy our code.

Over the years, Microsoft (and, let's be fair, a lot of other companies, and a lot of individual developers!) figured out how to add *many* helpful things like error highlighting, IntelliSense, WYSIWYG click-and-drag window UI editing, automatic code generation, and many other features.

After years of evolution, Visual Studio is now one of the most advanced code-editing tools ever built. And lucky for you, it's also a ***great tool for learning and exploring C# and app development***.

there are no Dumb Questions

Q: I understand what Program.cs does—that's where the code for my program lives. But does my program need the other two files and folders?

A: When you created a new project in Visual Studio, it created a **solution** for you. A solution is just a container for your project. The solution file ends in *.sln* and contains a list of the projects that are in the solution, with a small amount of additional information (like the version of Visual Studio used to create it). The **project** lives in a folder inside the solution folder. It gets a separate folder because some solutions can contain multiple projects—but yours only contains one, and it happens to have the same name as the solution (MyFirstConsoleApp). The project folder for your app contains two files: a file called *Program.cs* that contains the code, and a **project file** called *MyFirstConsoleApp.csproj* that has all of the information Visual Studio needs to **build** the code, or turn it into a something your computer can run. You'll eventually see **two more folders** underneath your project folder: the **bin/ folder** will have the executable files built from your C# code, and the **obj folder** will have the temporary files used to build it.

Statements are the building blocks for your apps

Your app is made up of classes, and those classes contain methods, and those methods contain statements. A **statement** is a line of code that does something.

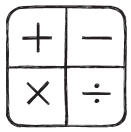
A statement can actually span multiple lines, which you'll see later in this book. But for now, you can just think of "statement" and "line of code" as the same thing.

So if we want to build apps that do a lot of things, we'll need a few **different kinds of statements** to make them work. You've already seen one kind of statement:

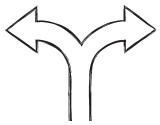
```
Console.WriteLine("Hello World!");
```

This is a **statement that calls a *method***—specifically, the Console.WriteLine method, which prints a line of text to the console. We'll also use a few other kinds of statements in this chapter and throughout the book. For example:

We use variables and variable declarations to let our app store and work with data.



Lots of programs use math, so we use mathematical operators to add, subtract, multiply, divide, and more.



Conditionals let our code choose between options, either executing one block of code or another.

Loops let our code run the same block over and over again until a condition is satisfied.

Statements live inside of methods

You wrote a **method** in Chapter 1 to set up your animal matching game. But what, exactly, is a method?

Methods do something

The Console.WriteLine method is part of .NET. It's not hard to guess that a method that starts with "Console." has something to do with reading or writing text in a console app. In this case, it writes a line of text to the console. It's a really useful method, and you'll use it—and a lot of other .NET methods (it has *thousands* of them!)—throughout this book.

You're going to write your own methods, and you're going to write code that **calls** those methods. To call a method, you write a statement that consists of the name of that method followed by parentheses and a semicolon. You can **pass** information to that method by putting it inside those parentheses—like passing "Hello, World!" when your code called the Console.WriteLine method.

Methods help you organize your code

Every method is made up of statements, and one method can contain many statements. Code tends to naturally organize into **blocks**, or lines of code which, taken together, do a specific thing. Methods are your way to take those code blocks, give them names, and make them easy to call.

When your program calls a method, it executes the first statement in that method, then the next, then the next, etc. When the method runs out of statements—or hits a **`return`** statement—it ends, and the program execution resumes after the statement that originally called the method.

Do you really *need* a method? You *could* copy the code in a method and paste it over the statement that called that method, and the app would still work. Putting the code into a method and ***giving it a name*** makes it a lot easier to understand what that code does.

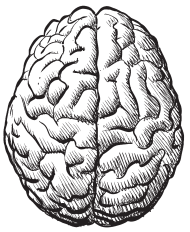
Brain Power

You'll use methods over and over again throughout this book to organize your code. Why do you think your code needs organizing?

When you're writing your code, you can take a block of code and turn it into a single method, multiple methods, or not use methods at all. How do you decide where to break up your code into methods?

The Console.WriteLine method writes a line to the console. Does that name make sense to you? Can you think of why it's useful for methods to have sensible names?

Your methods use variables to work with data



Every program, no matter how big or how small, works with data. Sometimes the data is in the form of a document, or an image in a video game, or a social media update—but it's all just data. That's where **variables** come in. A variable is what your program uses to store data.

Declare your variables

Whenever you **declare** a variable, you tell your program its *type* and its *name*. Once C# knows your variable's type, it will generate errors that stop your program from building if you try to do something that doesn't make sense, like subtract `"Fido"` from `48353`. Here's how to declare variables:

```
// Let's declare some variables
int maxWeight;
string message;
bool boxChecked;
```

Any line that starts with // is a comment and does not get executed. You can use comments to add notes to your code to help people read and understand it.

These are variable types. C# uses the type to define what data these variables can hold.

These are variable names. C# doesn't care what you name your variables—these names are for you.

This is why it's really helpful for you to choose variable names that make sense and are obvious.

Variables vary

A variable is equal to different values at different times while your program runs. In other words, a variable's value ***varies***. (Which is why "variable" is such a good name.) This is really important, because that idea is at the core of every program you'll write. Say your program sets the variable `myHeight` equal to 63:

```
int myHeight = 63;
```

Any time `myHeight` appears in the code, C# will replace it with its value, 63. Then, later on, if you change its value to 12:

```
myHeight = 12;
```

C# will replace `myHeight` with 12 from that point onwards (until it gets set again)—but the variable is still called `myHeight`.

Whenever your program needs to work with numbers, text, true/false values, or any other kind of data, you'll use variables to keep track of them. The variable's type defines what kind of data it can hold.

You need to assign values to variables before you use them

Try typing these statements just below the "Hello World" statement in your new console app:

Do this!

```
string z;
string message = "The answer is " + z;
```

Go ahead, try it right now. You'll get an error, and the IDE will refuse to build your code. That's because it checks each variable to make sure that you've assigned it a value before you use it. The easiest way to make sure you don't forget to assign your variables values is to combine the statement that declares a variable with a statement that assigns its value:

```
int maxWeight = 25000;
string message = "Hi!";
bool boxChecked = true;
```

These values are assigned to the variables. You can declare a variable and assign its initial value in a single statement (but you don't have to).

If you write code that uses a variable that hasn't been assigned a value, your code won't build. It's easy to avoid that error by combining your variable declaration and assignment into a single statement.

Once you've assigned a value to your variable, that value can change. So there's no disadvantage to assigning a variable an initial value when you declare it.

A few useful types

Every variable has a type that tells C# what kind of data it can hold. We'll go into a lot of detail about the many different types in C# in Chapter 4. In the meantime, we'll concentrate on the three most popular types. `int` holds integers (or whole numbers), `string` holds text, and `bool` holds **Boolean** true/false values.

var-i-a-ble, noun.
an element or feature likely to change.
Predicting the weather would be a whole lot easier if meteorologists didn't have to take so many ***variables*** *into account.*

Generate a new method to work with variables

In the last chapter, you learned that Visual Studio will **generate code for you**. This is quite useful when you're writing code—and ***it's also a really valuable learning tool***. Let's build on what you learned and take a closer look at generating methods.

Do this!

1 Add a method to your new MyFirstConsoleApp project.

Open the Console App project that you created in the last chapter. The Program.cs file has these lines:

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello World!");
```

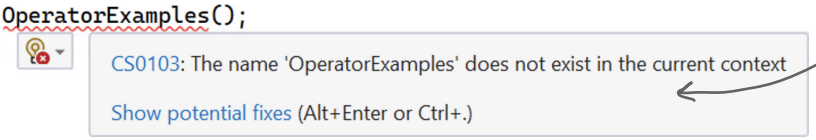
Replace those two lines with a statement that calls a method:

```
OperatorExamples();
```

A statement with the name of a method followed by opening and closing parentheses () calls that method.

2 Let Visual Studio tell you what's wrong.

As soon as you finish replacing the statement, Visual Studio will draw a red squiggly underline beneath your method call. Hover your mouse cursor over it. The IDE will display a pop-up window:

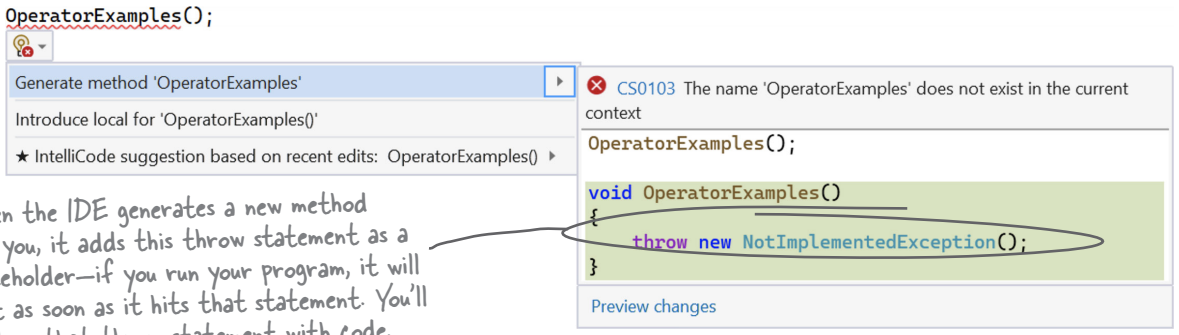


On a Mac, click the link or press Option+Return to show the potential fixes.

Visual Studio is telling you two things: that there's a problem—you're trying to call a method that doesn't exist (which will prevent your code from building)—and that it has a **potential fix**.

3 Generate the OperatorExamples method.

On **Windows**, the pop-up window tells you to press Alt+Enter or Ctrl+. to see the potential fixes. On **macOS**, it has a "Show potential fixes" link—press ⌥+Return to see the potential fixes. So go ahead and press either of those key combinations (or click on the dropdown to the left of the pop-up).

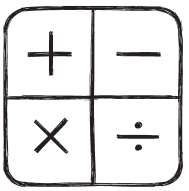


When the IDE generates a new method for you, it adds this throw statement as a placeholder—if you run your program, it will halt as soon as it hits that statement. You'll replace that throw statement with code.

This screenshot is from Windows. It looks a little different on a Mac, but has the same information.

The IDE has a solution: it will generate a method called OperatorExamples in your top-level statements. **Click "Preview changes"** to display a window that has the IDE's potential fix, adding a new method. Then **click Apply** to add the method to your code.

Add code that uses operators to your method



Once you've got some data stored in a variable, what can you do with it? Well, if it's a number, you might want to add or multiply it. If it's a string, you might join it together with other strings. That's where operators come in. Here's the method body for your new OperatorExamples method. **Add this code to your program**, and read the comments to learn about the operators it uses.

```
void OperatorExamples()
{
  // This statement declares a variable and sets it to 3
  int width = 3;

  // The ++ operator increments a variable (adds 1 to it)
  width++;

  // Declare two more int variables to hold numbers and
  // use the + and * operators to add and multiply values
  int height = 2 + 4;
  int area = width * height;
  Console.WriteLine(area);

  // The next two statements declare string variables
  // and use + to concatenate them (join them together)
  string result = "The area";
  result = result + " is " + area;
  Console.WriteLine(result);

  // A Boolean variable is either true or false
  bool truthValue = true;
  Console.WriteLine(truthValue);
}
```

String variables hold text. When you use the + operator with strings it joins them together, so adding "abc" + "def" results in a single string, "abcdef". When you join strings like that it's called concatenation.

MINI Sharpen your pencil

The statements you just added to your code will write three lines to the console: each Console.WriteLine statement prints a separate line. **Before you run your code**, figure out what they'll be and write them down. And don't bother looking for a solution, because we didn't include one! Just run the code to check your answers.

Here's a hint: converting a bool to a string results in either False or True.

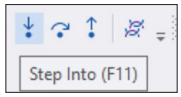
Line 1: _____

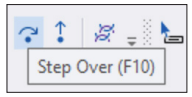
Line 2: _____

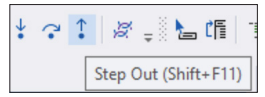
Line 3: _____

Use the debugger to watch your variables change

When you ran your program earlier, it was executing in the **debugger**—and that's an incredibly useful tool for understanding how your programs work. You can use **breakpoints** to pause your program when it hits certain statements and add **watches** to look at the value of your variables. Let's use the debugger to see your code in action. We'll use these three features of the debugger, which you'll find in the toolbar:



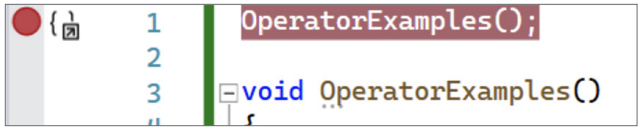




If you end up in a state you don't expect, just use the Restart button ( ) to restart the debugger.

1 Add a breakpoint and run your program.

Click on the first line of your program and **choose Toggle Breakpoint (F9 or ⌘/) from the Debug menu**. The line should now look like this—the line should be highlighted in red with a dot in the left margin:

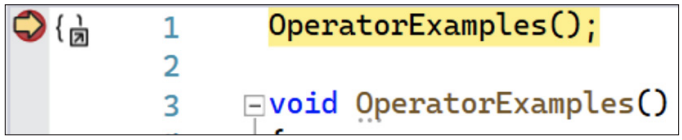


The debugging shortcut keys for Mac are Step Over (⇧⌘O), Step In (⇧⌘I), and Step Out (⇧⌘U). The screens will look a little different, but the debugger operates exactly the same.

Then press the MyFirstConsoleApp button to run your program in the debugger, just like you did earlier.

2 Step into the method.

Your debugger is stopped at the breakpoint on the statement that calls the OperatorExamples method.

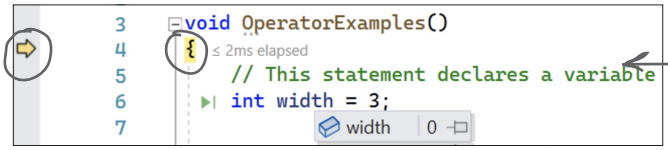


The red background and dot show you where you've set breakpoints. The yellow arrow and highlight show the line of code the debugger is paused on.

Press *Step Into (F11)*—the debugger will jump into the method and pause before it runs the first statement.

3 Examine the value of the width variable.

When you're **stepping through your code**, the debugger pauses after each statement that it executes. This gives you the opportunity to examine the values of your variables. Hover over the `width` variable.

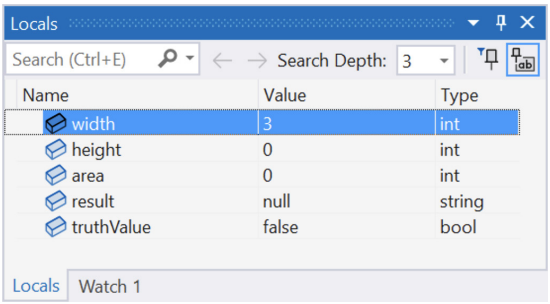


The highlighted bracket and arrow in the left margin mean the code is paused just before the first statement of the method.

The IDE displays a pop-up that shows the current value of the variable—it's currently 0. Now **press Step Over (F10 on Windows or ⇧⌘O on macOS)**—it goes past the comment to the first statement, which is now highlighted. Now **press Step Over again**, then hover over `width` again. It now has a value of 3.

4 The Locals window shows the values of your variables.

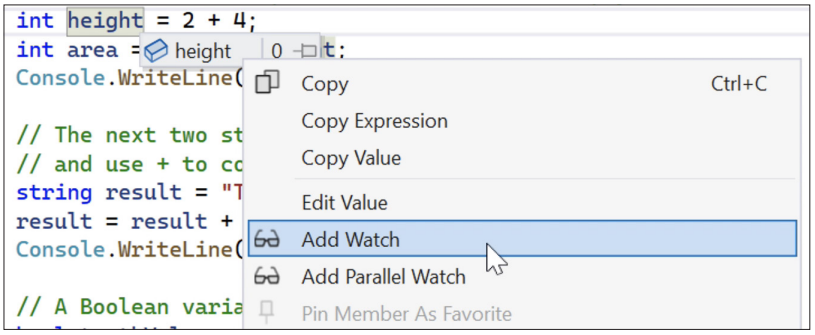
The variables that you declared are **local** to your OperatorExamples method—which just means that they exist only inside that method, and can only be used by statements in the method. Visual Studio displays their values in the Locals window at the bottom of the IDE when it's debugging.



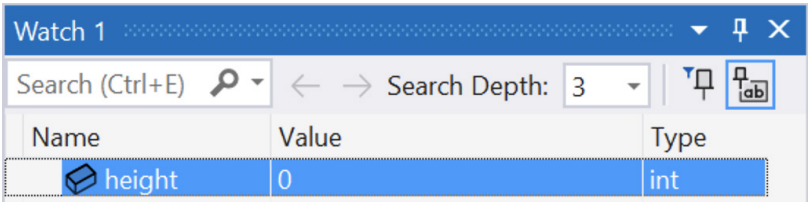
The Locals and Watch windows in Visual Studio for Mac look a little different than they do on Windows, but they contain the same information. You add watches the same way in both Windows and Mac versions of Visual Studio.

5 Add a watch for the height variable.

A really useful feature of the debugger is the **Watch window**, which is typically in the same panel as the Locals window at the bottom of the IDE. When you hover over a variable, you can add a watch by right-clicking on the variable name in the pop-up window and choosing Add Watch. Hover over the `height` variable, then right-click and choose **Add Watch** from the menu.



Now you can see the `height` variable in the Watch window.



The debugger is one of the most important features in Visual Studio, and it's a great tool for understanding how your programs work.

6 Step through the rest of the method.

Step over each statement in OperatorExamples. As you step through the method, keep an eye on the Locals or Watch window and watch the values as they change. On **Windows,** press **Alt+Tab** before and after the Console.WriteLine statements to switch back and forth to the Debug Console to see the output. On **macOS**, you'll see the output in the Terminal window so you don't need to switch windows.

Use operators to work with variables

Once you have data in a variable, what do you do with it? Well, most of the time you'll want your code to do something based on the value. That's where **equality operators**, **relational operators**, and **logical operators** become important:

Equality Operators

The == operator compares two things and is true if they're equal.

The != operator works a lot like ==, except it's true if the two things you're comparing are not equal.

Relational Operators

Use > and < to compare numbers and see if a number in one variable is bigger or smaller than another.

You can also use >= to check if one value is greater than or equal to another, and <= to check if it's less than or equal.

Logical Operators

You can combine individual conditional tests into one long test using the && operator for ***and*** and the || operator for ***or***.

Here's how you'd check if `i` equals 3 ***or*** `j` is less than 5:

```
(i == 3) || (j < 5)
```

Don't confuse the two equals sign operators!

You use one equals sign (=) to set a variable's value, but two equals signs (==) to compare two variables. You won't believe how many bugs in programs—even ones made by experienced programmers!—are caused by using = instead of ==. If you see the IDE complain that you "cannot implicitly convert type 'int' to 'bool'," that's probably what happened.

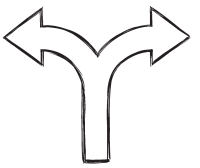
Use operators to compare two int variables

You can do simple tests by checking the value of a variable using a comparison operator. Here's how you compare two ints, x and y:

```
x < y (less than)
x > y (greater than)
x == y (equals - and yes, with two equals signs)
```

These are the ones you'll use most often.

"if" statements make decisions



Use **`if` statements** to tell your program to do certain things only when the **conditions** you set up are (or aren't) true. The `if` statement ***tests the condition*** and executes code if the test passed. A lot of `if` statements check if two things are equal. That's when you use the `==` operator. That's different from the single equals sign (=) operator, which you use to set a value.

```
int someValue = 10;
string message = "";

if (someValue == 24)
{
    message = "Yes, it's 24!";
}
```

Every `if` statement starts with a test in parentheses, followed by a block of statements in brackets to execute if the test passes.

The statements inside the curly brackets are executed only if the test is true.

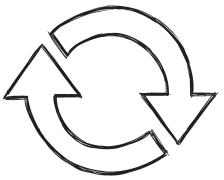
if/else statements also do something if a condition isn't true

`if/else` statements are just what they sound like: if a condition is true they do one thing ***or else*** they do the other. An `if/else` statement is an `if` statement followed by the **`else` keyword** followed by a second set of statements to execute. If the test is true, the program executes the statements between the first set of brackets. Otherwise, it executes the statements between the second set.

```
if (someValue == 24)
{
  // You can have as many statements
  // as you want inside the brackets
  message = "The value was 24.";
}
else
{
  message = "The value wasn't 24.";
}
```

REMEMBER – always use two equals signs to check if two things are equal to each other.

Loops perform an action over and over



Here's a peculiar thing about most programs (*especially* games!): they almost always involve doing certain things over and over again. That's what **loops** are for—they tell your program to keep executing a certain set of statements as long as some condition is true or false.

while loops keep looping statements while a condition is true

In a **while loop**, all of the statements inside the curly brackets get executed as long as the condition in the parentheses is true.

```
while (x > 5)
{
  // Statements between these brackets will
  // only run if  x is greater than 5, then
  // will keep looping as long as x > 5
}
```

do/while loops run the statements then check the condition

A **do/while** loop is just like a while loop, with one difference. The while loop does its test first, then runs its statements only if that test is true. The do/while loop runs the statements first, ***then*** runs the test. So if you need to make sure your loop always runs at least once, a do/while loop is a good choice.

```
do
{
  // Statements between these brackets will run
  // once, then keep looping as long as x > 5
} while (x > 5);
```

for loops run a statement after each loop

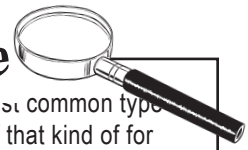
A **for loop** runs a statement after each time it executes a loop.

Every for loop has three statements. The first statement sets up the loop. It will keep looping as long as the second statement is true. And the third statement gets executed after each time through the loop.

```
for (int i = 0; i < 8; i = i + 2)
{
    // Everything between these brackets
    // is executed 4 times
}
```

The parts of the for statement are called the initializer (int i = 0), the conditional test (i < 8), and the iterator (i = i + 2). Each time through a for loop (or any loop) is called an iteration.
The conditional test always runs at the beginning of each iteration, and the iterator always runs at the end of the iteration.

for loops Up Close



A **for loop** is a little more complex—and more versatile—than a simple while loop or do loop. The most common type of for loop just counts up to a length. The **`for` code snippet** causes the IDE to create an example of that kind of for loop:

```
for (int i = 0; i < length; i++)
{

}
```

When you use the for snippet, press Tab to switch between i and length. If you change the name of the variable i, the snippet will automatically change the other two occurrences of it.

A for loop has four sections—an initializer, a condition, an iterator, and a body:

```
for (initializer; condition; iterator) {
    body
}
```

Most of the time you'll use the initializer to declare a new variable—for example, the initializer `int i = 0` in the `for` code snippet above declares a variable called `i` that can only be used inside the for loop. The loop will then execute the body—which can either be one statement or a block of statements inside curly braces—as long as the condition is true. At the end of each iteration the for loop executes the iterator. So this loop:

```
for (int i = 0; i < 10; i++) {
    Console.WriteLine("Iteration #" + i);
}
```

will iterate 10 times, printing `Iteration #0`, `Iteration #1`, ..., `Iteration #9` to the console.

Sharpen your pencil

Here are a few loops. Write down if each loop will repeat forever or eventually end. If it's going to end, how many times will it loop? Also, answer the questions in the comments in loops #2 and #3.

```
// Loop #1
int count = 5;
while (count > 0) {
   count = count * 3;
   count = count * -1;
}
```

Remember, a for loop always runs the conditional test at the beginning of the block, and the iterator at the end of the block.

```
// Loop #4
int i = 0;
int count = 2;
while (i == 0) {
   count = count * 3;
   count = count * -1;
}
```

```
// Loop #2
int j = 2;
for (int i = 1; i < 100;
     i = i * 2)
{
    j = j - 1;
    while (j < 25)
    {
       // How many times will
       // the next statement
       // be executed?
       j = j + 5;
    }
}
```

```
// Loop #5
while (true) { int i = 1;}
```

```
// Loop #3
int p = 2;
for (int q = 2; q < 32;
     q = q * 2)
{
   while (p < q)
   {
      // How many times will
      // the next statement
      // be executed?
      p = p * 2;
   }
   q = p - q;
}
```

Hint: p starts out equal to 2. Think about when the statement "p = p * 2" is executed.

When we give you pencil–and–paper exercises, we'll usually give you the solution on the next page.

Sharpen your pencil Solution

Here are a few loops. Write down if each loop will repeat forever or eventually end. If it's going to end, how many times will it loop? Also, answer the questions in the comments in loops #2 and #3.

```
// Loop #1
int count = 5;
while (count > 0) {
   count = count * 3;
   count = count * -1;
}
```

Loop #1 executes once.

Remember, `count = count * 3` multiplies count by 3, then stores the result (15) back in the same count variable.

```
// Loop #4
int i = 0;
int count = 2;
while (i == 0) {
   count = count * 3;
   count = count * -1;
}
```

Loop #4 runs forever.

```
// Loop #2
int j = 2;
for (int i = 1; i < 100;
     i = i * 2)
{
    j = j - 1;
    while (j < 25)
    {
       // How many times will
       // the next statement
       // be executed?
       j = j + 5;
    }
}
```

Loop #2 executes 7 times.

The statement `j = j + 5` is executed 6 times.

```
// Loop #5
while (true) { int i = 1;}
```

Loop #5 is also an infinite loop.

```
// Loop #3
int p = 2;
for (int q = 2; q < 32;
     q = q * 2)
{
   while (p < q)
   {
      // How many times
will
      // the next statement
      // be executed?
      p = p * 2;
   }
   q = p - q;
}
```

Loop #3 executes 8 times.

The statement `p = p * 2` executes 3 times.

Take the time to really figure out how loop #3 works. Here's a perfect opportunity to try out the debugger on your own! Set a breakpoint on `q = p - q;` and use the Locals window to watch how the values of `p` and `q` change as you step through the loop.

Use code snippets to help write loops

Do this!

You'll be writing a lot of loops throughout this book, and Visual Studio can help speed things up for you with **snippets**, or simple templates that you can use to add code. Let's use snippets to add a few loops to your OperatorExamples method.

If your code is still running, choose **Stop Debugging** (**Shift+F5**) from the Debug menu (or press the square Stop button ■ in the toolbar). Then find the line `Console.WriteLine(area);` in your OperatorExamples method. Click at the end of that line so your cursor is after the semicolon, then press Enter a few times to add some extra space. Now start your snippet. **Type `while` and press the Tab key twice.** The IDE will add a template for a while loop to your code, with the conditional test highlighted:

```
while (true)
{

}
```

Type **`area < 50`**—the IDE will replace `true` with the text. **Press Enter** to finish the snippet. Then add two statements between the brackets:

```
while (area < 50)
{
    height++;
    area = width * height;
}
```

IDE Tip: Brackets

If your brackets (or braces, either name will do) don't match up, your program won't build, which leads to frustrating bugs. Luckily, the IDE can help with this! Put your cursor on a bracket, and the IDE highlights its match.

Next, use the **`do/while` loop snippet** to add another loop immediately after the while loop you just added. Type **`do` and press Tab twice**. The IDE will add this snippet:

```
do
{

} while (true);
```

Type `area > 25` and press Enter to finish the snippet. Then add two statements between the brackets:

```
do
{
    width--;
    area = width * height;
} while (area > 25);
```

Now **use the debugger** to really get a good sense of how these loops work:

1. Click on the line just above the first loop and choose **Toggle Breakpoint** (**F9**) from the Debug menu to add a breakpoint. Then run your code and **press F5** to skip to the new breakpoint.
2. Use **Step Over** (**F10**) to step through the two loops. Watch the Locals window as the values for `height`, `width`, and `area` change.
3. Stop the program, then change the while loop test to **`area < 20`** so both loops have conditions that are false. Debug the program again. The while checks the condition first and skips the loop, but the do/while executes it once and then checks the condition.

Sharpen your pencil

Let's get some practice working with conditionals and loops. Update the code in your Program.file to matche the new code below, including TryAnIf, TryAnIfElse, and TrySomeLoops methods. ***Before you run your code,*** read it carefully and try to answer the questions based on how you think it will run. Then run your code and see if you got them right.

```
TryAnIf();
TrySomeLoops();
TryAnIfElse();

private static void TryAnIf()
{
    int someValue = 4;
    string name = "Bobbo Jr.";
    if ((someValue == 3) && (name == "Joe"))
    {
        Console.WriteLine("x is 3 and the name is Joe");
    }
    Console.WriteLine("this line runs no matter what");
}

private static void TryAnIfElse()
{
    int x = 5;
    if (x == 10)
    {
        Console.WriteLine("x must be 10");
    }
    else
    {
        Console.WriteLine("x isn't 10");
    }
}

private static void TrySomeLoops()
{
    int count = 0;

    while (count < 10)
    {
        count = count + 1;
    }

    for (int i = 0; i < 5; i++)
    {
        count = count - 1;
    }

    Console.WriteLine("The answer is " + count);
}
```

What does the TryAnIf method write to the console?

……

……

What does the TryAnIfElse method write to the console?

……

What does the TrySomeLoops method write to the console?

……

We didn't include answers for this exercise in the book. After you write down the answer, run the code check the the console output to see if your answer is correct.

Some useful things to keep in mind about C# code

- **Don't forget that all your statements need to end in a semicolon.**
```
name = "Joe";
```

- **Add comments to your code by starting a line with two slashes.**
```
// this text is ignored
```

- **Use /* and */ to start and end comments that can include line breaks.**
```
/* this comment
 * spans multiple lines */
```

- **Variables are declared with a *type* followed by a *name*.**
```
int weight;
// the variable's type is int and its name is weight
```

- **Most of the time, extra whitespace is fine.**
So this: `int j = 1234 ;`
Is exactly the same as this: `int j = 1234;`

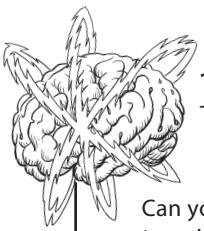
- **If/else, while, do, and for are all about testing conditions.**
Every loop we've seen so far keeps running as long as a condition is true.

THERE'S A FLAW IN YOUR LOGIC! WHAT HAPPENS IF I WRITE A LOOP WITH A CONDITIONAL TEST THAT NEVER BECOMES FALSE?

Then your loop runs forever.

Every time your program runs a conditional test, the result is either **`true`** or **`false`**. If it's **`true`**, then your program goes through the loop one more time. Every loop should have code that, if it's run enough times, should cause the conditional test to eventually return **`false`**. If it doesn't, then the loop will keep running until you kill the program or turn the computer off!

This is sometimes called an infinite loop, and there are definitely times when you'll want to use one in your code.



Brain Power

Can you think of a reason that you'd want to write a loop that never stops running?

Game design... and beyond

Mechanics

The **mechanics** of a game are the aspects of the game that make up the actual gameplay: its rules, the actions that the player can take, and the way the game behaves in response to them.

- Let's start with a classic video game. The **mechanics of Pac Man** include how the joystick controls the player on the screen, the number of points for dots and power pellets, how ghosts move, how long they turn blue and how their behavior changes after the player eats a power pellet, when the player gets extra lives, how the ghosts slow down as they go through the tunnel—*all* of the rules that drive the game.
- When game designers talk about a **mechanic** (in the singular), they're often referring to a single mode of interaction or control, like a double jump in a platformer or shields that can only take a certain number of hits in a shooter. It's often useful to isolate a single mechanic for testing and improvement.
- **Tabletop games** give us a really good way to understand the concept of mechanics. Random number generators like dice, spinners, and cards are great examples of specific mechanics.
- You've already seen a great example of a mechanic: the **timer** that you added to your animal matching game changed the entire experience. Timers, obstacles, enemies, maps, races, points…these are all mechanics.
- Different mechanics **combine** in different ways, and that can have a big impact on how the players experience the game. Monopoly is a great example of a game that combines two different random number generators—dice and cards—to make a more interesting and subtle game.
- Game mechanics also include the way the **data is structured and the design of the code** that handles that data—even if the mechanic is unintentional! Pac Man's legendary *level 256 glitch*, where a bug in the code fills half the screen with garbage and makes the game unplayable, is part of the mechanics of the game.
- So when we talk about mechanics of a C# game, **that includes the classes and the code**, because they drive the way that the game works.

Here's another one of those "Game design... and beyond" sections. This one is all about mechanics, an important part of video game design. If you're not interested in writing video games, read these sections anyway! They have important concepts that we'll build on throughout the book.

Definitely! Every program has its own kind of mechanics.

There are mechanics at every level of software design. They're easier to talk about and understand in the context of video games. We'll take advantage of that to help give you a deeper understanding of mechanics, which is valuable for designing and building any kind of project.

Here's an example. The mechanics of a game determine how hard or easy it is to play. Make Pac Man faster or the ghosts slower and the game gets easier. That doesn't necessarily make it better or worse—just different. And guess what? The same exact idea applies to how you design your classes! You can think of ***how you design your methods and fields*** as the mechanics of the class. The choices you make about how to break up your code into methods or when to use fields make them easier or more difficult to use.

Bullet Points

- **Methods** are made up of statements. Calling a method executes its statements in order.
- Putting statements into a method and giving it a name helps make your code **easier to read**.
- When a method runs out of statements or executes a **`return`** statement, **execution resumes after the statement** that originally called the method.
- A variable's **type** determines what kind of data—like whole or decimal numbers, text, or true/false values—that it can hold
- You need to **assign values** to variables before you can use them.
- **Operators** like +, -, *, and / perform manipulations on the data stored in variables. The = operator assigns a value, while the == operator compares two values.
- **`if` statements** tell your program to do certain things only when the conditions you set up are (or aren't) true.
- **Loops** execute a set of statements over and over again until a condition is met. **`for`**, **`while`**, and **`do`**/**`while`** loops all iterate over statements multiple times, but they work differently from each other.
- Visual Studio's **code snippets** feature helps you write if statements and loops.

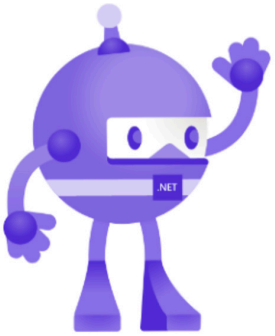
Controls drive the mechanics of your user interfaces

In the last chapter, you built a game using Button and Label **controls**. There are a lot of different ways that you can use controls, and the choices you make about what controls to use can really change your app. Does that sound weird? It's actually ***really similar to the way we make choices about mechanics in game design***. If you're designing a tabletop game that needs a random number generator, you can choose to use dice, a spinner, or cards. If you're designing a platformer, you can choose to have your player jump, double jump, wall jump, or fly (or do different things at different times). The same goes for apps: if you're designing an app where the user needs to enter a number, you can choose from different controls to let them do that… ***and that choice affects how your user experiences the app***.

Meet some of the controls you'll use in this book

Most of the chapters in this book feature a .NET MAUI projcet. We included them so you can go beyond console apps and start learning how to build visual apps. In those projects, you'll use many different controls to build each app's the **user interface** (or **UI**)—or the way the window is laid out so the user can interact with it—of each app.

Here are the controls you've seen so far.



An Image control does exactly what you'd expect it to do—it displays an image. In this case, it's displaying the image in a file called dotnet_bot.png.

Hello, World!

A Label displays text. You can set the font size, color, spacing, and text decorations (like italics or boldface).

Click me

A Button control shows a clickable button. It can call a method when you click it, and you can set or change its text.

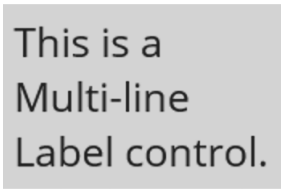
The user interface, or UI, is the part of the app that your user interacts with. In a console app, the UI is made up of text, and the user uses the keyboard to interact with it. In a .NET MAUI app, the UI is built using controls.

Other controls you'll use in this book

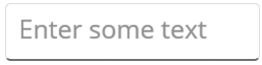
Controls are common user interface components, and they serve as the building blocks of your UI. The choices you make about what controls to use change the mechanics of your app.

Most of the chapters in this book contain a .NET MAUI project. You'll use various controls to build the UI for each of those apps. Here are a few of the ones that you'll use.

We can borrow the idea of mechanics from video games to understand our options, so we can make great choices for any of our own apps—not just games.



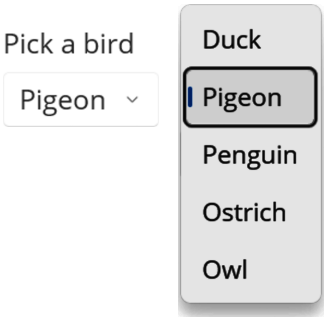
An Label can include multiple lines, which are separated by line breaks so it knows how to split them up.



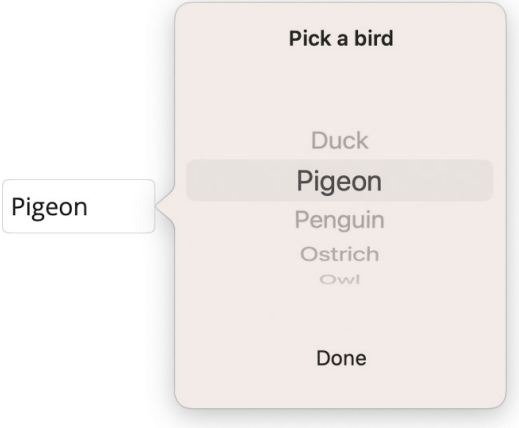
An Entry control lets your user enter text. It displays a placeholder, or lighter-colored text that gives the user some information about what they should type.

A Switch is a horizontal button that lets the user toggle (or switch back and forth) between two states, in this case on and off.

These are two different controls that let users enter numbers. A Stepper (on the left) presents the user with two buttons to increment or decrement—add or remove one—to a value. A Slider (on the right) lets the user slide back and forth to choose a decimal number.

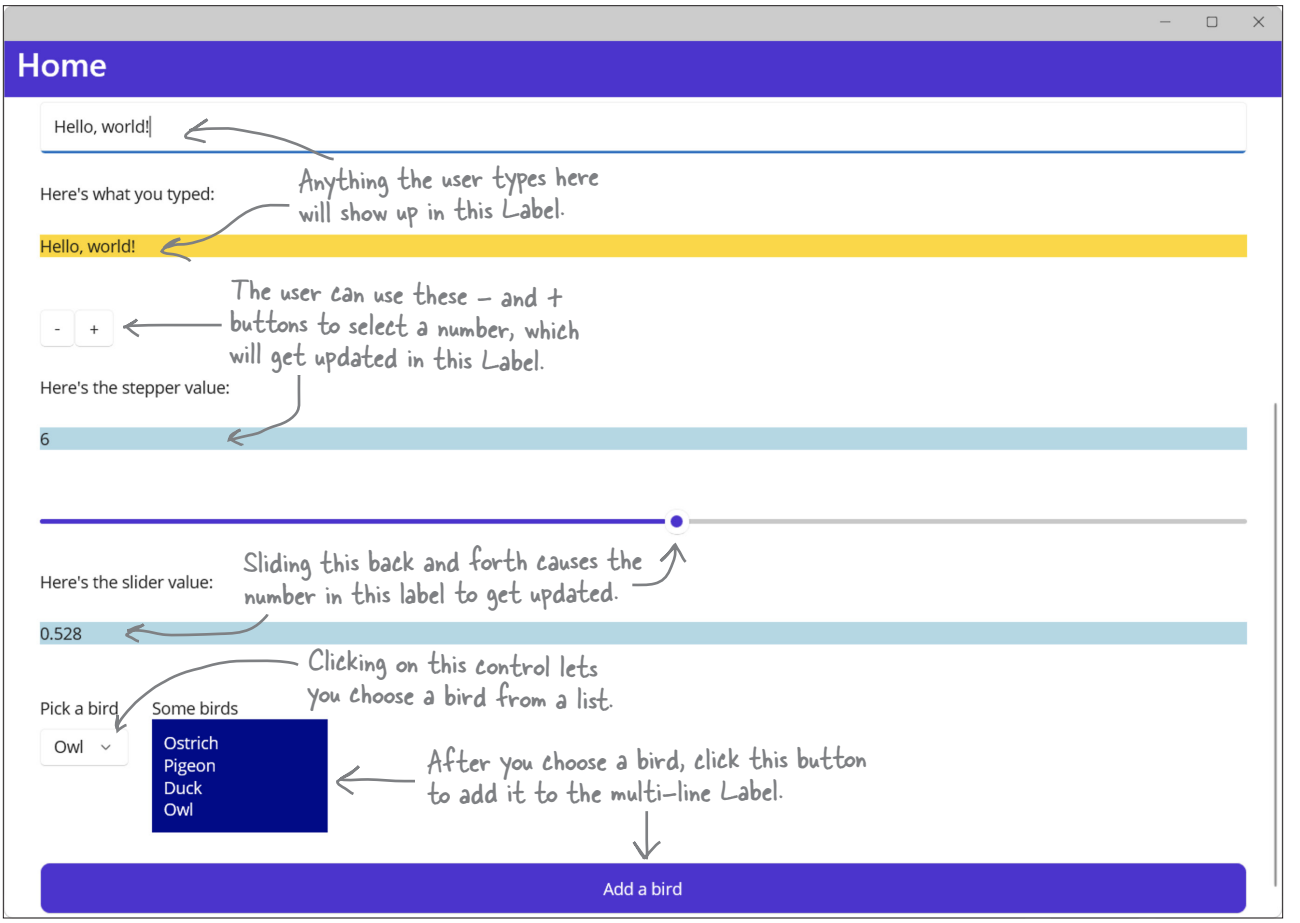


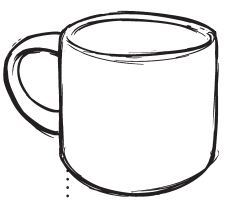
A Picker lets the user choose an item from a list. It looks a little different in Windows (on the left) and macOS (on the right), but both versions function in exactly the same way.



Build a .NET MAUI app to experiment with controls

You've probably seen most the controls we just showed you (even if you didn't know all of their official names). Now let's **create a .NET MAUI app** to get some practice using some of them. The app will be really simple—the user will use controls to enter values, and the app will display those values.





Relax

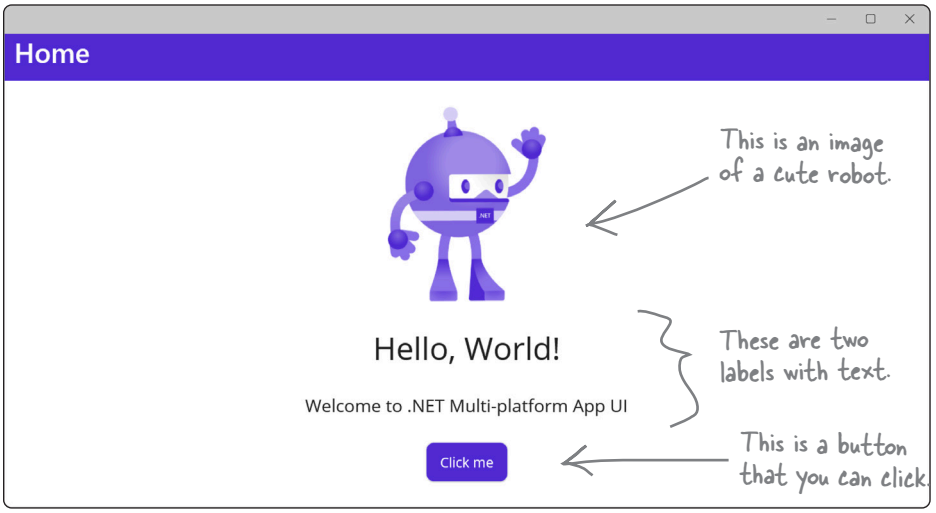
Don't worry about memorizing the XAML in this project. You'll pick it up throughout the book.

This Do this! and these exercises are all about getting some practice using XAML to build a UI with controls. You can always refer back to it when we use these controls in projects later in the book.

Create a new app to experiment with controls

Do this!

Go back to Visual Studio and **create a new .NET MAUI project**, just like you did in Chapter 1. Name your project ***ExperimentWithControls***. Run your new .NET MAUI app. It will pop up a window with a picture of a cute robot, text that says *Hello, World!*, and smaller text that says *Welcome to .NET Multiplatform APP UI*, and finally a button with the label *Click me*.



Now go back to Visual Studio and **double-click the file MainPage.xaml** to open it. Use the buttons in the left margin that look like a minus sign in a square to collapse the <Image>, <Label>, and <Button> tags. Each of those tags corresponds to one of the controls in your app.

```
1   <?xml version="1.0" encoding="utf-8" ?>
2   <ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
3                xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
4                x:Class="ExperimentWithControls.MainPage">
5
6       <ScrollView>
7           <VerticalStackLayout
8               Spacing="25"
9               Padding="30,0"
10              VerticalOptions="Center">
11
12              <Image ... />
17
18              <Label ... />
23
24              <Label ... />
30
31              <Button ... />
37
38          </VerticalStackLayout>
39      </ScrollView>
40
41  </ContentPage>
```

Click here to collapse or expand a tag.

This <Image> tag displays the picture of the robot in an Image control.

These two <Label> tags create Label controls that display the two lines of text.

This <Button> tag adds the Button control to the page.

The XAML for your MAUI page starts with a ContentPage tag, which can contain a single control—in this case, it's a ScrollView, which scrolls its content and displays a scrollbar on the side.

A ScrollView can contains a single control. Yours contains a VerticalStackLayout, which can contain multiple controls (like the Image, two Labels, and Button in your page) and displays them stacked on top of each other vertically.

Explore your new MAUI app and figure out how it works

When you created your new .NET MAUI app, Visual Studio used a **template** to create the files for your app, substituting the name that you specified(*ExperimentWithControls*) in various lines in the files. Let's dig in to the project that you created.

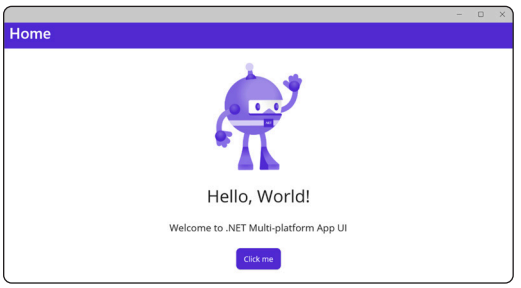
Do this!

1 Create a new .NET MAUI project called ExperimentWithControls.

Go back to Visual Studio and **create a new .NET MAUI project**, just like you did in Chapter 1. Name your project *ExperimentWithControls*.

2 Run your app and click on the button.

When you run the app, you'll see the app window—it should look like this:



MAUI is cross-platform, which means you'll see the same app—with the same code!—whether you're running Windows or macOS.

You know you want to click the "Click me" button. Go ahead! Lhe label on the button will change from "Click me" to "Clicked 1 time" and increment (or add one) every time you click the button.

3 Investigate how the counter on the button works.

Go to the Solution Explorer, expand MainPage.xaml, and **open MainPage.xaml.cs**. Find the line that has the statement `count++;` and **place a breakpoint** on it.

```
0 references
12  private void OnCounterClicked(object sender, EventArgs e)
13  {
14      count++;
15
16      if (count == 1)
17          CounterBtn.Text = $"Clicked {count} time";
18      else
19          CounterBtn.Text = $"Clicked {count} times";
20
21      SemanticScreenReader.Announce(CounterBtn.Text);
22  }
```

The C# code for a page in your MAUI app is called code-behind. The XAML code and the C# code in the code-behind file work together to make your page work.

Before you go to the next step, read the code.
Can you figure out how the button works?

4 Click on the button and step through the code.

Add a watch for the count variable, just like you did earlier in the chapter. Then use "Step Over" to step through the code. Here's waht the OnCounterClicked event handler method does:

- First it executes `count++` to increment (or add one to) the `count` variable.
- Next it uses an if statement to check if the `count` variable equals 1. If it does, then it sets the button's text to "Clicked 1 time".
- If it doesn't equal 1, it sets the button's text to "Clicked {count} times" – you'll learn more about exactly what that the $ dollar sign and {brackets} do in Chapter 5 (it's called *string interpolation*).

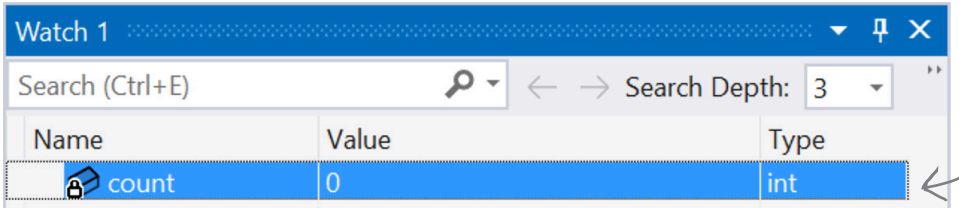
5 Click on the button and step through the code.

The program should pause on the breakpoint, just like you saw earlier in the chapter:

```
12  private void OnCounterClicked(object sender, EventArgs e)
13  {
14      count++;
15
16      if (count == 1)
17          CounterBtn.Text = $"Clicked {count} time";
18      else
19          CounterBtn.Text = $"Clicked {count} times";
20
21      SemanticScreenReader.Announce(CounterBtn.Text);
22  }
```

This if statement sets the text on the Count button to "time" if count equals 1, or "times" if it has any other value.

Add a watch for the `count` variable, just like you did earlier with the OperatorExamples console app. It should start out with the value 0. Step over to the



The count variable starts out with the value 0. After the count++ statement is executed, the new value is 1, and every time you click the button count++ increases the value of the count variable by 1.

Keep stepping through the code. The if statement checks whether the count value is equal to 1. If it is, it executes this statement:

```
CounterBtn.Text = $"Clicked {count} time";
```

That sets the text Go back to the window with the XAML code. Find this line:

```
<Button
    x:Name="CounterBtn"
```

This is the x:Name property. It gave the button the name "CounterBtn" that you can use in code.

Every control can have a name. The `x:Name` property sets the name of the control—in this case, the button is named CounterBtn—and your C# code can use that name to make the control do things.

The XAML For Your Button Up Close

You've been editing the the XAML code in your MainPage.xaml file—are you starting to get comfortable with it? This is a great time to take a closer look at the part of your XAML that displays the button.

Here's the Button tag. Take a look at each of its five properties. Can you figure out what all do?

```
<Button
    x:Name="CounterBtn"
    Text="Click me"
    SemanticProperties.Hint="Counts the number of times you click"
    Clicked="OnCounterClicked"
    HorizontalOptions="Center" />
```

The x:Name property gives your control a name you can use in your code

The first property is x:Name, which sets the name of the control so you can use it in your C# code:

```
x:Name="CounterBtn"
```

You just saw a control name in action. When you clicked the button, the event handler method executed this statement to set the button's text, using the name CounterBtn set by the x:Name property:

```
CounterBtn.Text = $"Clicked {count} time";
```

This line uses the CounterBtn name to update the text displayed on the button.

The x:Name property gives your control a name you can use in your code

The next property determines what text is displayed n the button:

```
Text="Click me"
```

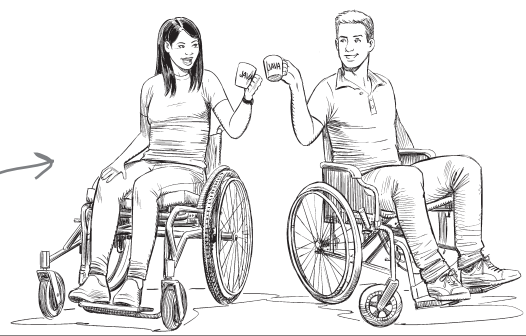
The button displays "Click me" when you first run the app. That line of code in the method changes the text to "Clicked 1 time" the first time you click it, then "Clicked 2 times" when you click it again. That line of code starts with the name of the control (CounterBtn), followed by a period, followed by Text, the name of the property.

SemanticProperties help you make your apps *accessible*

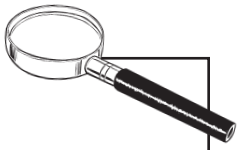
When we create our apps, we want everyone to be able to use them—and that includes people with disabilities.

A **screen reader** is a tool that lets people who are blind, visually impaired, or have learning disabilities or other conditions that interfere with their ability to read use our visual apps. Semantic properties help your app work with a screen reader.

A screen reader is an accessibility tool for people with visual, learning, or other disabilities—just like a wheelchair is an accessibility tool for people with mobility-related disabilities. They're both really important for helping to make everyday things more accessible to everyone.



The XAML For Your Button Up Close



Use a screen reader to experiment with the SemanticProperites.Hint property

The best way to make your apps accessible is to use them the way someone with accessibility issues would—in this case, using a screen reader built into your operating system.

- **In Windows, start the Narrator app.** You can run it from the Start menu, or use Windows logo key + Ctrl + Enter to turn Narrator on or off, and Windows logo key + Ctrl + N to bring up Narrator settings. Narrator will display a window with an overview of how Narrator works. It will also start to read the contents of that window, displaying a box around the section of the window that it's readingh. You can go back to that window to turn of Narrator.
- **In MacOS, start the VoiceOver utility.** It lives in the Applications/Utilities folder, but if your keyboard has Touch ID, the easiest way to turn it on or off is to press and hold the Command key while you quickly press Touch ID three times. By default the VoiceOver utility displays a welcome dialog—press the V key or click the "Use Voiceover" button to start VoiceOver.

Once you have Narrator or VoiceOver running, switch to your app window. You'll hear a voice telling you details about what's on the screen. People with visual impairments often have trouble using a mouse, so they use the keyboard to interact with apps—and you'll do the same thing. **Press the tab key** to navigate to the "Click Here" button. The screen reader will announce that you are on a button. Listen closely—you'll hear it speak the SemanticProperties.Hint value: "Counts the number of times you click"

Press Enter to click the button. Your app will execute code that includes this statement:

```
SemanticScreenReader.Announce(CounterBtn.Text);
```

When it does, the screen reader will announce the contents of the button ("Clicked 1 time").

The Clicked property tells your app what event handler method to run when the button is clicked

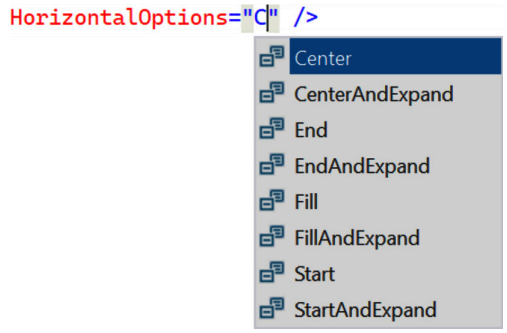
Take a look at the next property in the button's XAML code:

```
Clicked="OnCounterClicked"
```

When you clicked the button, your app used that property to figure out which method to run. You saw this in action when you placed a breakpoint on the first line of that method.

The HorizontalOptions property centers your button

When you run your app, the "Click Me" button is centered in the middle of the window. Go back to the code editor, select the word `Center` in that line of XAML code, and type C. Visual Studio will display an IntelliSense pop-up with all of the different options. Try selecting Start or End, then ren your app again—now the button will be displayed on the left or right side of the window. Experiment with all of the different horizontal options for Button control.



Add an Entry control to your app

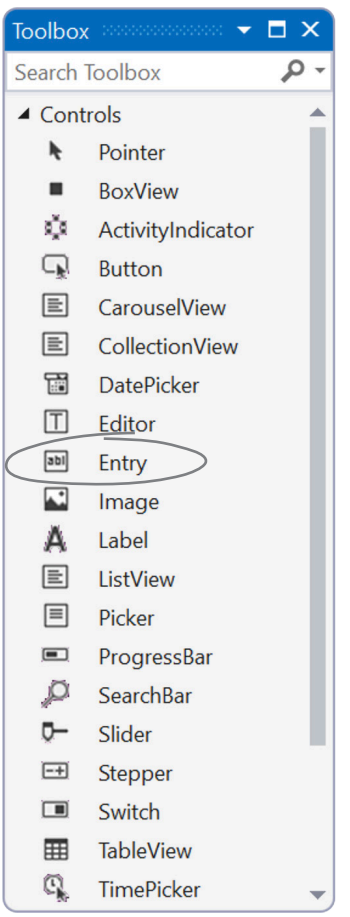
And **Entry control** gives your user a box where they can enter text. You'll add one your app—and you'll use a really useful tool in Visual Studio to do it: the **Toolbox window**. The The Toolbox is a feature of Visual Studio that makes it easy to add controls to your app.

1. Stop your app, then open the MainPage.xaml editor window in Visual Studio.
2. Place your mouse cursor just after the closing /> bracket at the end of your Button control tag, then **press enter three times** to add three blank lines. Click on the second line that you just added, so there's a blank line above your mouse cursor and another blank line below it.
3. Open the Toolbox window in Visual Studio (if it isn't already open) by **choosing "Toolbox" from the View menu**.
4. **Double-click Entry** in the Toolbox window. Visual Studio will automatically add an <Entry> tag at your cursor location, on that middle blank line you added.

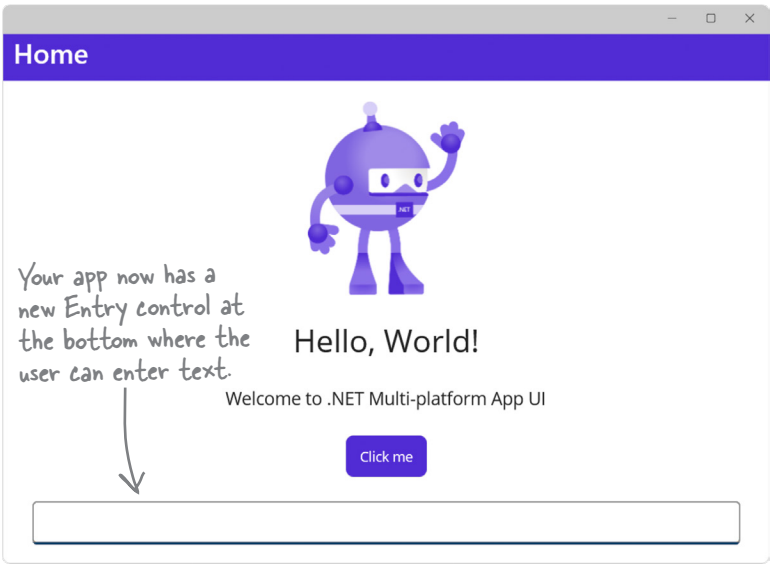
Visual Studio's Toolbox window helps you add new controls to your XAML code. If you don't see the Toolbox window, choose "Toolbox" from the View menu to display it.

Here's what you should see in your XAML code:

```
<Button
    x:Name="CounterBtn"
    Text="Click me"
    SemanticProperties.Hint="Counts the number of times you click"
    Clicked="OnCounterClicked"
    HorizontalOptions="Center" />

<Entry Placeholder="" />
```



Now run your app. Congratulations, you just added a control for entering text!



Add properties to your Entry control

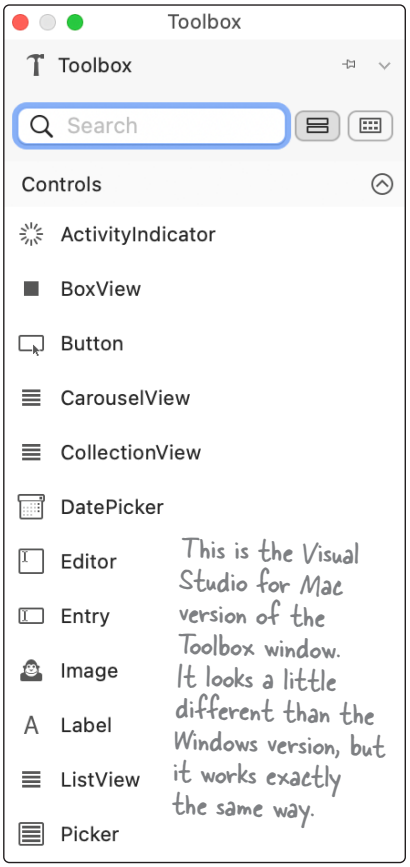
Let's make your Entry control a little more usable by adding **placeholder text**, or text that appears in a lighter color to help the user understand what they're supposed to enter.

Edit the XAML code for your Entry control to **add a Placeholder property**. And since we always want our apps to be accessible to people who use screen readers, **add a SemanticProperties.Hint property** too. Notice that when you add the properties, they show up in Visual Studio's typeahead pop-up window, making it easier for you to add them.

Your Entry tag should look like this:

```
<Entry
    Placeholder="Enter some text"
    SemanticProperties.Hint="Lets you enter some text" />
```

Now run your app – you'll see a new Entry control at the bottom. The placeholder text will appear as ("Enter some text" in a lighter color, and will disappear as soon as you you type text into it.



there are no Dumb Questions

Q: Why did the Entry control get added to the bottom of my app? How did it know where in the window to display?

A: When you created a new .NET MAUI app, Visual Studio used a template that generated the XAML code for the main page in the MainPage.xaml file. This file contains a set of **nested tags**, or tags that contain other tags—so one tag's start and end appear after the start and before the end of another tag. Each of these tags *creates a specific kind of control* that determines how the page is displayed.

The outermost tag in your app's XAML is a <ContentPage> opening tag, which defines a single view that contains the rest of the page. If you scroll down to the bottom of the file, you'll see the closing </ContentPage> tag. Right inside that <ContentPage> is a <ScrollView> tag—everything between the opening <ScrollView> and closing </ScrollView> tags defines contents that will automatically display a scrollbar that lets you scroll up and down if it's too long for the page. The <ScrollView> tag contains a <VerticalStackLayout> tag, with a matching </VerticalStackLayout> closing tag at the bottom. A VerticalStackLayout can contain a series of controls, one after another. Each of those controls will be displayed on the page in a vertical stack, in the order that they appear in the file.

So since the Entry control is at the bottom of the file just above the closing </VerticalStackLayout> tag, it will appear at the bottom of the page. And because it's nested inside the <ScrollView>...</ScrollView> tags, if you make your window shorter than the height of the page, you'll be able to scroll down to it.

Make your Entry control update a Label

Your app already has two Label controls. Let's add a third one and make it display everything the Entry does, so when you enter or update text in the Entry it automatically updates the Label.

① Use the Toolbox to add a new Label control to the bottom of your page.

When you drag the label out of the Toolbox, it will have an empty Text property:

```
<Label Text="" />
```

Change the Text property to make it display text. Then give it a SemanticProperties.Description property. This is what will get read aloud if your user is using a screen reader:

```
<Label Text="Here's what you typed:"
     SemanticProperties.Description="Here's what you typed:" />
```

You can add line breaks between properties to make them easier to read.

② Use the Toolbox to add a second Label control under the one you just added.

Every time the user changes the text in your Entry control, the app will update this new Label to show the text that they typed. Drag a new Label control out of the Toolbox and drop it in your XAML code between the Label control that you just added and the closing </VerticalStackLayout> tag. Then set its properties:

- You'll be writing code to set the label text, so delete the Text property.
- Since you're going to write code that updates the Label, you'll need to give it a name. Use an x:Name property to name it EntryOutput: `x:Name="EnteredText"`
- Keep making your app acceessible by adding a description for people using a screen reader: `SemanticProperties.Description="The text that the user entered"`

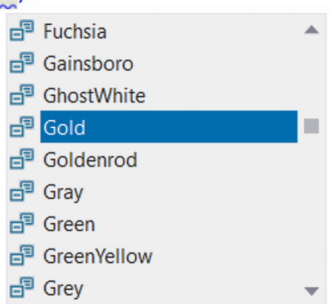
Your new Label should look like this:

```
<Label x:Name="EnteredText"
     SemanticProperties.Description="The text that the user entered" />
```

③ Give your label a background color.

Add a BackgroundColor property. When you start typing, Visual Studio will pop up a IntelliSense window. Choose Gold for the background color.

```
<Label x:Name="EnteredText"
    SemanticProperties.Description="The text that the user entered"
    BackgroundColor=""/>
```



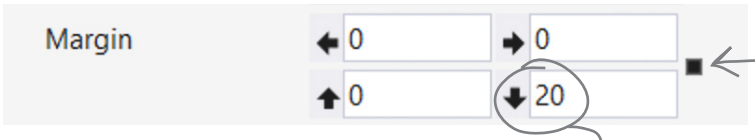
You can use Visual Studio's IntelliSense to help you add properties. Once you add it, you'll see a box with preview of the color in the XAML editor.

```
BackgroundColor="Gold"/>
```

4 Use the Properties window to add a bottom margin.

So far you've been adding properties by writing XAML code by hand. Luckily, Visual Studio has some useful tools to help you edit your XAML. The Properties window gives you an easy way to edit the properties on your controls.

Click the XAML for your Label control so the cursor is somewhere between the tags. Go to the Properties window (if you don't see it, use the View menu to display it) and find Margin. Enter 20 for the lower margin to give it a 20 pixel magin (where a pixel is 1/96th of an inch on an unscaled screen).



When this square turns black (or circle if you're using macOS), it means that the property has a value set. You can click it to see a menu.

Enter 20 for the bottom margin to give 20 units of space between the label and the bottom of the page.

Your property should now look like this:

```
<Label x:Name="EnteredText"
    SemanticProperties.Description="The text that the user entered"
    BackgroundColor="Gold" Margin="0,0,0,20"/>
```

5 Add an event handler method.

Back in Chapter 1 you used event handler methods so your Animal Matching Game could respond to mouse clicks and timer ticks. Now you know more about C# methods—this is a good chance to apply that knowledge by creating a new event handler method to update the EnteredText control ever time the user types in the Entry control. Add a TextChanged property to your Entry control. When it comes time to enter the value, Visual Studio will suggest the value ***<New Event Handler>***:

```
<Entry
    Placeholder="Enter some text"
    SemanticProperties.Hint="Lets you enter some text"
    TextChanged=""
    />
```

<New Event Handler>

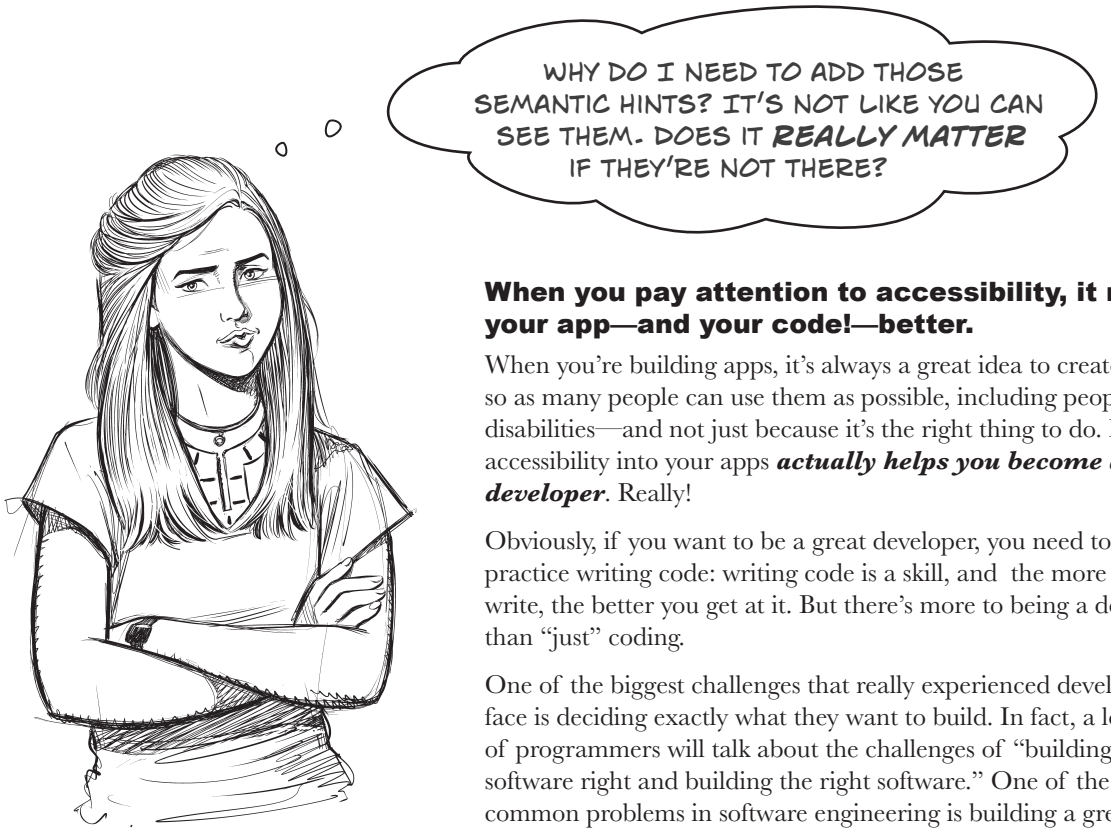
Press return or use the mouse to accept the suggestion—this will cause Visual Studio to **add a new event handler method called `Entry_TextChanged` automatically**. You probably also noticed that it also displayed this message when you were adding the event handler:

> Bind event to a newly created method called 'Entry_TextChanged'. Use 'Go To Definition' to navigate to the newly created method.

Right-click on Entry_TextChanged and choose Go To Definition. This will open up MainPage.xaml.cs and jump directly to the method that Visual Studio added. Add this line of code to the method:

```
private void Entry_TextChanged(object sender, TextChangedEventArgs e)
{
      EnteredText.Text = e.NewTextValue;
}
```

Now **run your app**. You should see a label that says "Here's what you typed:" followed by a gold-colored label. Click on the Entry control and type some text—it will appear in the gold-colored label immediately.



When you pay attention to accessibility, it makes your app—and your code!—better.

When you're building apps, it's always a great idea to create them so as many people can use them as possible, including people with disabilities—and not just because it's the right thing to do. Building accessibility into your apps ***actually helps you become a better developer***. Really!

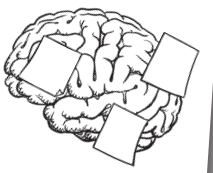
Obviously, if you want to be a great developer, you need to get practice writing code: writing code is a skill, and the more code you write, the better you get at it. But there's more to being a developer than "just" coding.

One of the biggest challenges that really experienced developers face is deciding exactly what they want to build. In fact, a lot of programmers will talk about the challenges of "building the software right and building the right software." One of the most common problems in software engineering is building a great product that doesn't do what your users need.

That's where accessibility can help you. Building accessible code well means taking the time to **really understand** how people with disabilities will use your app. Taking the time to understand and empathize with them will help you build your app better—and it's great practice for skills that will help you build the right software.

One of our big goals with this book is to help you learn important skills that will help you become an all-around great developer. Understanding your users is a really important skill, and paying attention to accessibility is a great way to get better at it!

Make it Stick

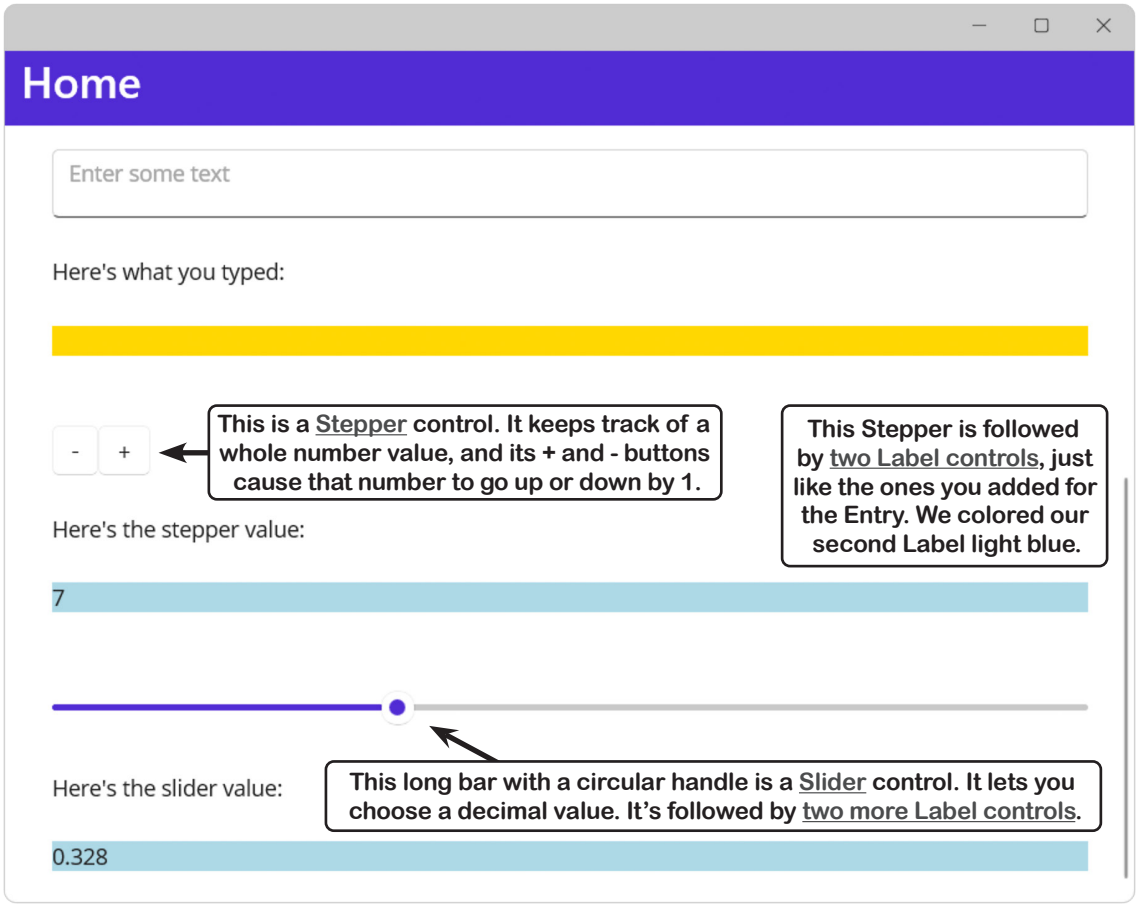


Here's an great way to get accessibility ideas to stick in your brain—especially if you don't have a disability. Turn on your screen reader, then ***leave it on*** *while you code or do other work. Once you're used to it,* ***close your eyes*** *and keep working. Can you work using just the screen reader?*

Using a screen reader is an effective way to really understandhow accessibility works.

Exercise

You added Entry and Label controls to your app—and Visual Studio's Toolbox window, Properties window, and IntelliSense helped you. Can you add six more controls to your app to let your user enter numeric values?



Use the Toolbox window, Properties window, and Visual Studio editor to **add a Stepper control, two Label controls, a Slider control, and two more Label controls** to your app.

The two Label controls that display the values should have the BackgroundColor property set to LightBlue. **Name them StepperValue and SliderValue**. Make sure you **add SematicProperties.Description properties**.

You want your app to automatically update those the StepperValue control every time the the stepper value changes, so **add a ValueChanged event handler** to the Stepper control. Add this line of code to the event handler:

```
StepperValue.Text = e.NewValue.ToString();
```

Then add a ValueChanged event handler **to the Slider control**. It should be identical, except that it updates the SliderValue label instead of the StepperValue label.

Don't forget to add SemanticDescription.Hint properties to your Stepper and Slider controls.

Exercise Solution

Here's the XAML to add the six controls to MainPage.xaml:

This is the Label control that was already in your XAML code — make sure you put your six new controls below it.

```
    <Label x:Name="EnteredText"
        SemanticProperties.Description="The text that the user entered"
        BackgroundColor="Gold" Margin="0,0,0,20"/>

    <Stepper Minimum="0" Maximum="10" Increment="1"
             SemanticProperties.Description="Lets you enter a whole number"
             ValueChanged="Stepper_ValueChanged" />

    <Label
        Text="Here's the stepper value:"
        SemanticProperties.Description="Here's the stepper value" />

    <Label x:Name="StepperValue"
        SemanticProperties.Description="The number the user chose with the Stepper"
        BackgroundColor="LightBlue" Margin="0,0,0,20"/>


    <Slider Minimum="0" Maximum="1" ValueChanged="Slider_ValueChanged" />

    <Label
        Text="Here's the slider value:"
        SemanticProperties.Description="Here's the stepper value" />

    <Label x:Name="SliderValue"
        SemanticProperties.Description="The number the user chose with the Slider"
        BackgroundColor="LightBlue" Margin="0,0,0,20"/>

</VerticalStackLayout>
```

These are the default properties when you drag the Stepper out of the Toolbox. Try experimenting with them.

You can add this ValueChanged property just like you did with TextChanged on your Entry control.

Here's the Slider control. It has the default properties, plus a ValueChanged property.

Here's the Label that displays the Slider value. It works exactly like the Label you used to show the value in the Entry control.

This is the closing VerticalStackLayout tag that was already in your XAML code — make sure you put your six new controls above it.

Here are the event handler methods to add to MainPage.xaml.cs:

```
private void Stepper_ValueChanged(object sender, ValueChangedEventArgs e)
{
    StepperValue.Text = e.NewValue.ToString();
}

private void Slider_ValueChanged(object sender, ValueChangedEventArgs e)
{
    SliderValue.Text = e.NewValue.ToString();
}
```

The two event handlers for the Stepper and Slider controls update the Label.

In the exercise instructions, we gave you this line of code:

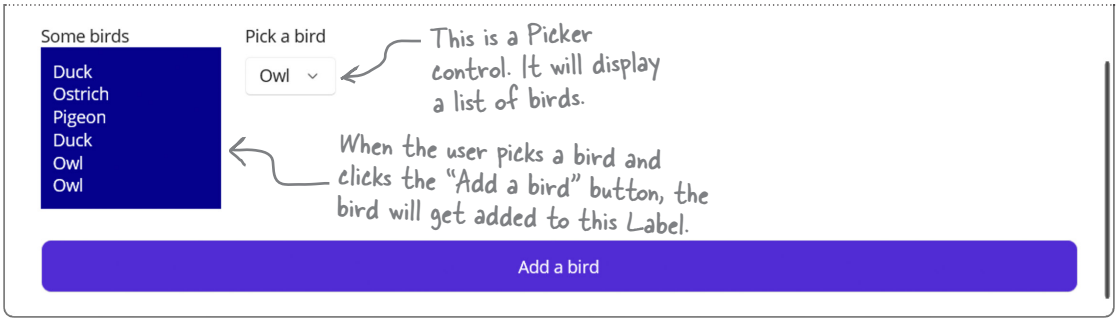
```
SliderValue.Text = e.NewValue.ToString();
```

What do you think `.ToString()` does?

Here's a hint: Stepper and Slider controls can only provide numeric values, but labels can only display text.

Combine horizontal and vertical stack layouts

In this last part of the exercise you'll add a **Picker control**, which displays a list of items that you can pick from. You'll also use a Label control to display the values that were picked. Here's what it will look like:



Notice how the Label and Picker controls are next to each other? You'll get that layout by using a **HorizontalStackLayout control**. It works just like the VerticalStackLayout control causes all of the controls you've added to your app so far to be stacked on top of each other, except instead they get stacked next to each other.

You'll nest one Layout inside another

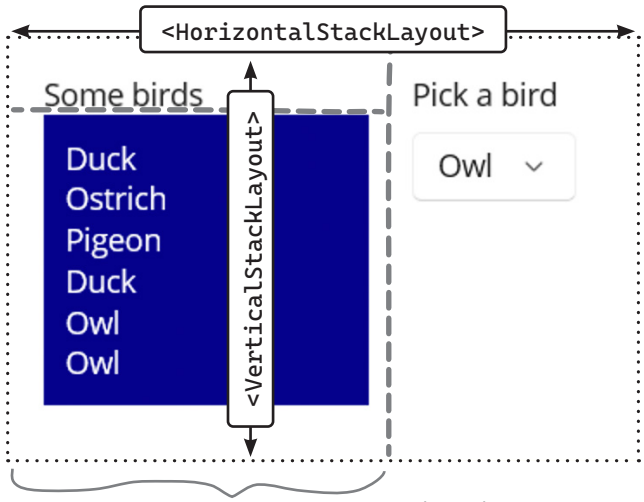
We'll use **nesting**—where one layout control lives inside another one—to create a more complex layout.

Here's how it will work:

This HorizontalStackLayout will get nested inside the outer VerticalStackLayout that's used to lay out the entire page.

```
<HorizontalStackLayout>
    <VerticalStackLayout>
        <Label Text="Some Birds" ... />
        <Label x:Name="Birds" ... />
    </VerticalStackLayout>
    <Picker x:Name="BirdPicker" ... />
</HorizontalStackLayout>
```

The HorizontalStackLayout contains two controls, a nested VerticalStackLayout and a Picker, which get stacked left to right.



This nested VerticalStackLayout contains two controls, a Label that says "Some birds" and a Label named Birds that will get updated when the user clicks the button, stacked vertically.

Add a Picker control to display a list of choices

A **Picker control** displays a list of items in a dropdown so the user can pick one of them. Let's add one to your app.

1 Add the XAML for a Picker control and a Label for it to update.

You've already seen how a VerticalStackLayout control lets you stack controls on top of each other. You can also stack controls horizontally by adding a **HorizontalStackLayout control**.

Go ahead and **add this XAML code** just above the closing </VerticalStackLayout> tag. You can type it all or use the Toolbox. When you add Clicked event for the button, **press Tab** to let Visual Studio generate an event handler method for you, just like you did earlier..

```
    <HorizontalStackLayout Spacing="20">

        <Picker x:Name="BirdPicker" Title="Pick a bird" />

        <VerticalStackLayout>

            <Label Text="Some birds" SemanticProperties.Description="Some birds"/>

            <Label x:Name="Birds"
                   Padding="10" MinimumWidthRequest="150"
                   TextColor="White" BackgroundColor="DarkBlue"
                   SemanticProperties.Description="Shows the added birds" />

        </VerticalStackLayout>

    </HorizontalStackLayout>

    <Button x:Name="AddBird" Clicked="AddBird_Clicked" Text="Add a bird"
            Margin="0,0,0,20" SemanticProperties.Hint="Adds a bird"/>

</VerticalStackLayou>
```

This <Button ... /> tag should be just above the closing </VerticalStackLayout> tag that's already in the MainPage.xaml file.

2 Initialize the Picker with a list of birds.

Open the MainPage.xaml.cs file and find the MainPage method at the top. This method gets run every time the page loads. Insert two lines after `InitializeComponent();` and **add this code**.

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

    BirdPicker.ItemsSource = new string[] {
        "Duck",
        "Pigeon",
        "Penguin",
        "Ostrich",
        "Owl"
    };
}
```

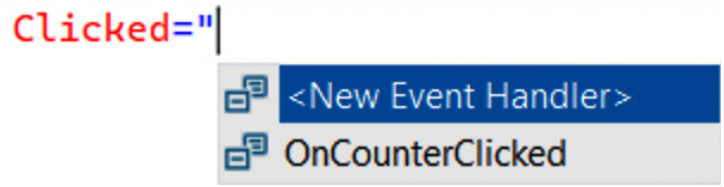
You used the x:Name property to name your Picker control "BirdPicker" – this sets the list of items in the picker that get displayed when the user clicks on it.

Put your code at the end of the MainPage method.

Open the MainPage.xaml.cs file and add this code to the MainPage method. Be careful with the square and curly braces, quotes, and commas.

3 Fill in the event handler for the Button control.

When you added the XAML for the Button control, you used Visual Studio's IntelliSense pop-up to help you add a new event handler to the C# code:



Since you used the x:Name property to name your Picker control AddBird, Visual Studio created an empty event handler method called AddBird_Clicked:

```
private void AddBird_Clicked(object sender, EventArgs e)
{


}
```

Add this line of code to the AddBird_Clicked method:

```
private void AddBird_Clicked(object sender, EventArgs e)
{
    Birds.Text = Birds.Text + Environment.NewLine + BirdPicker.SelectedItem;
}
```

Take a closer look at the line of code—let's break down exactly what it does.

1. The line starts with `Birds.Text =` ... which means it's setting the text in the Bird label.
2. The text is being set to `Birds.Text +` followed by additional things—this means it's going to take whatever is in the Label and **append text** to it, or add additional text to the end.
3. The first thing that gets appended is `Environment.NewLine`, which adds a line break. The Label control will display **multi-line text**, adding a line break every time it sees a line break.
4. After the line break, it appends `BirdPicker.SelectedItem`—this is the item that's currently selected in the Picker control.

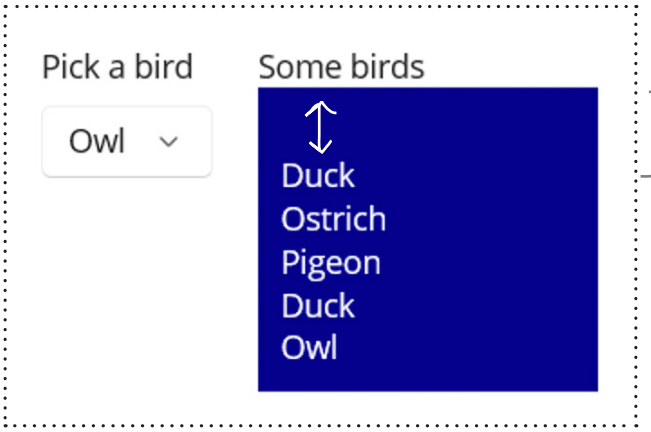
4 Run your app and use your new Picker control.

Scroll to the bottom of the app, choose a bird from the Picker, and click the Add a bird button—it will get added to the Label that contains the birds. Select a few more birds and add them.

You're right! The app doesn't match the screenshot.

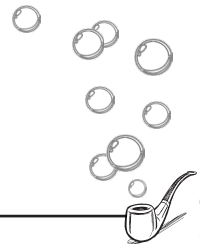
Take a look at the screenshot we showed you earlier:

Run your app and try adding those same birds. When you get to the first owl, you'll see extra space at the top of the Label:



Oops! It looks like we've got some extra space at the top of the Label that shows the birds that you picked.

Looks like we've got a bug. Time to put on your Sherlock Holmes cap.
Let's sleuth out this bug!



Sleuth it Out

The Case of the Extraneous Space

Understanding a bug is the first step in fixing it.

In the last chapter, we looked at the code carefully and found several clues to help us solve the Case of the Unexpected Match. But as you keep going through this book, your apps will get longer and longer, and while looking at the code is a good start, it may not always be the best way to figure out what's causing a bug.

Luckily, the Visual Studio debugger is a great tool for that. (That's why it's called a debugger: it's a tool that helps you get rid of bugs!)

Reproduce the bug

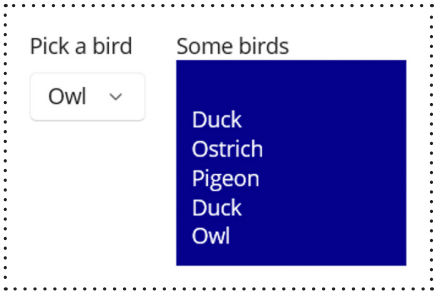
It's obvious that there's a problem! But as Sherlock Holmes once said, "There is nothing more deceptive than an obvious fact." When you're sleuthing out bugs, can't just rely on what's obvious. You need to confirm for yourself exactly what's going on. The way to do that is to **reproduce the bug**.

Stop your app. Make sure it's not running, so you've got a fresh start. Then do this:

1. Start your app again
2. Pick Duck and click the "Add a bird" button
3. Pick Ostrich and click the "Add a bird" button
4. Pick Pigeon and click the "Add a bird" button
5. Pick Duck and click the "Add a bird" button
6. Pick Owl and click the "Add a bird" button

> "There is nothing more deceptive than an obvious fact."
> — Sherlock Holmes

Your app should now look exactly like the screenshot:



Now restart your app, then try it again with different birds. You should still see extra space at the top of the Label. You can make the bug happen over and over again, at will. That means the problem is **reproducible**: you can follow a set of stpes to make it happen. Reproducing a bug is a great first step to fixing it.

Before you go on, can you sleuth out what's causing the extra space to get added?

Sleuth it Out

Every good investigation starts by identifying a list of suspects

When you're tracking down a bug, what's the first thing you should do? You could start placing breakpoints in the code… but where? **The first step in debugging is thinking.** Look at your code, think about how it works, and try to imagine where the bug might be. That will help you figure out where to put your breakpoints.

So let's think through the code. It starts with a button—and the button calls a method:

```
<Button x:Name="AddBird" Clicked="AddBird_Clicked" Text="Add a bird"
        Margin="0,0,0,20" SemanticProperties.Hint="Adds a bird"/>
```

All of the code to add the bird to the Label is in that **AddBird_Clicked method**. Now we have a suspect!

IDE Tip: Using the debugger

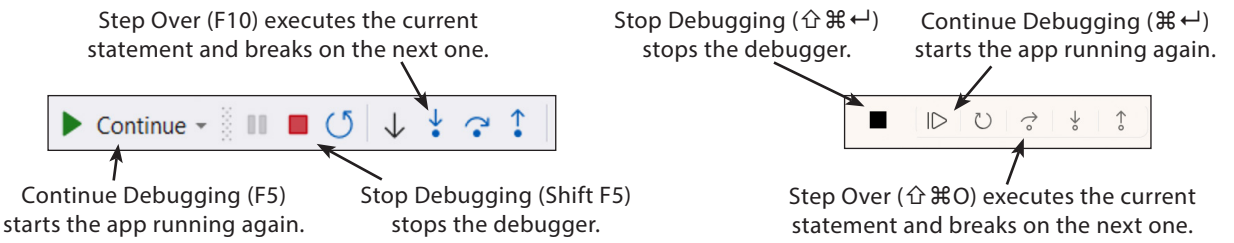
You're going to be using the debugger a lot in this book! We've walked you through it a few times, but you're as you get further in the book and write more and more code, you should feel comfortable using the debugger on your own.

Let's start with **a few tips** to help you get comfortable debugging your code.

- Think before you debug. Read through your code. Understand how it works (and not just how *you think it works*).
- Use the Watch window, Locals window, and hovering over variables to keep track of their values. They all do the same thing—show you the value of a variable—so you can decide which one you feel most comfortable with.
- Don't be afraid to restart your app. Stop and start your code frequently—every time you run your code, you're *running an experiment*. Run it as many times as it takes to understand what's going on.

Here's a handly **list of the debugger commands**. They may feel strange at first, but they'll be second nature soon.

- When you press the triangle Run button in the toolbar or choose Start Debugging (F5 or ⌘↵), Visual Studio starts running your code in the debugger. You can place a breakpoint whether or not the debugger is running.
- To place a breakpoint, click on a line of code and choose Toggle Breakpoint (F9 or ⌘/) form the Debug menu.
- When your code hits a breakpoint, it stops running so you can inspect variables.
- When Visual Studio breaks on a breakpoint, the toolbar shows you the commands you can use to keep executing. Debugging code can be a little weird to get used to if you haven't done it before, so try sticking to just these four commands—here's where you'll find them in the toolbar (for Windows or macOS) and using keyboard shortcuts:



Add a breakpoint and start debugging the code

Now that we have a suspect, let's catch it in the act. Add a breakpoint to the line in the AddBird_Clicked method:

```
private void AddBird_Clicked(object sender, EventArgs e)
{
    Birds.Text = Birds.Text + Environment.NewLine + BirdPicker.SelectedItem;
}
```

Now **run your code**. Pick a bird, then click the "Add a bird" button. The debugger stops on your breakpoint. Next, **add a watch for Birds.Text**, just like you did earlier in the chapter. The value should be **`null`**.

Name	Value	Type
Birds.Text	null View	string

Then **step over that line of code** (F10 on Windows or ⇧⌘O on macOS) to run it. You should see this value:

Name	Value	Type
Birds.Text	"\r\nPigeon" View	string

The value of Birds.Text is a string: `\r\n` followed by the bird you picked. What do you think `\r\n` does?

NOTE: If you're using macOS, you'll see `\n` instead of `\r\n`.

Continue debugging (F5 or ⌘↵) to start your app running again. Pick a different bird and step over the line of code. Now have a look at the Birds.Text watch:

Name	Value	Type
Birds.Text	"\r\nPenguin\r\nOstrich" View	string

Repeat the process a few more times: continue debugging, pick a bird, click the button, step over, check the watch. Eventually your Birds.Text value will look something like this (you'll see `\n` instead of `\r\n` on macOS):

```
"\r\nPenguin\r\nOstrich\r\nPigeon\r\nDuck\r\nOwl\r\nPigeon\r\nDuck\r\nOwl"
```

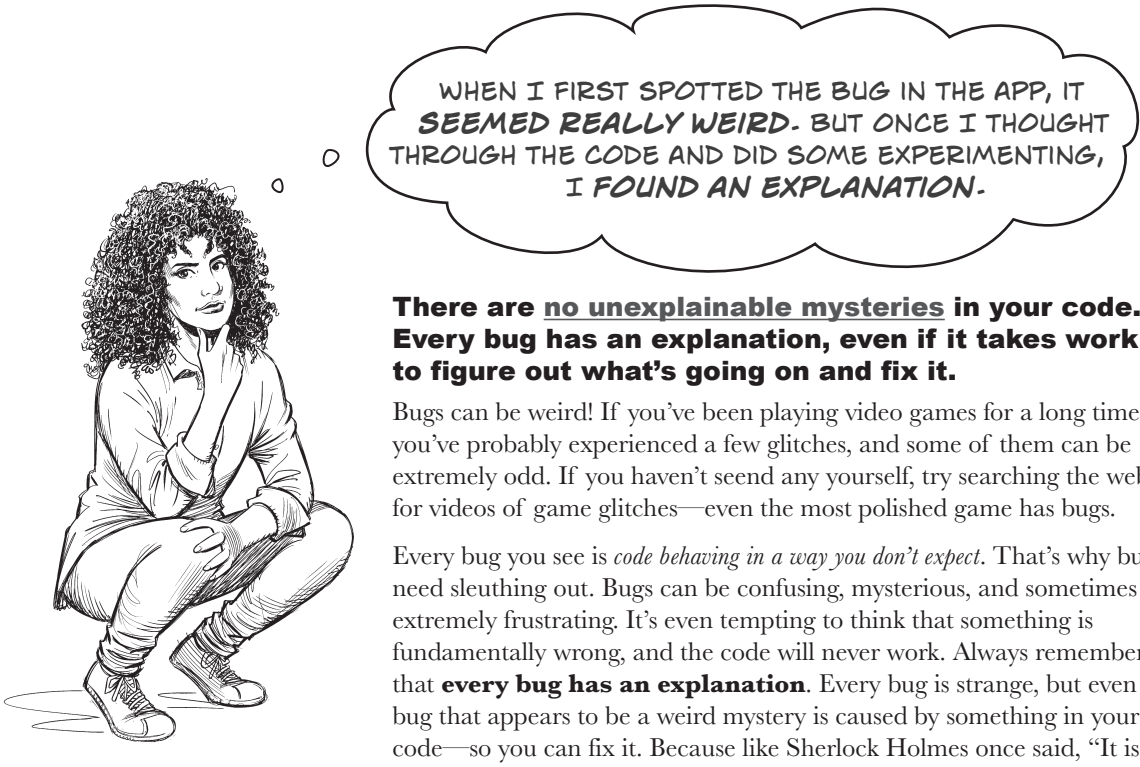
You've probably figured out by now that **the \r\n or \n is the line break**. The first time the AddBird_Clicked method is called, the Label text is empty (that's what the **`null`** value means), so when the app adds the current value (empty) plus a line break plus the bird, it adds an extra line break to at the start of the string.

Now that we've found the culprit, we can fix the app. **Replace the AddBird_Clicked** method with this code, which uses a special method, **String.IsNullOrEmpty**, which checks if a string is empty:

```
private void AddBird_Clicked(object sender, EventArgs e)
{
    if (!String.IsNullOrEmpty(Birds.Text))
    {
        Birds.Text = Birds.Text + Environment.NewLine;
    }
    Birds.Text += BirdPicker.SelectedItem;
}
```

String.IsNullOrEmpty(Birds.Text) checks the value of Birds.Text and returns true if it's empty or false if it's not. The ! in front of it reverses that value, so the line break is only added if Birds.Text is empty. You'll learn all about how a method can return a value in the next chapter.

Run your app again and add a few birds to the Label—there's no more empty space above it. **Your app is fixed!**



There are no unexplainable mysteries in your code. Every bug has an explanation, even if it takes work to figure out what's going on and fix it.

Bugs can be weird! If you've been playing video games for a long time, you've probably experienced a few glitches, and some of them can be extremely odd. If you haven't seend any yourself, try searching the web for videos of game glitches—even the most polished game has bugs.

Every bug you see is *code behaving in a way you don't expect*. That's why bugs need sleuthing out. Bugs can be confusing, mysterious, and sometimes extremely frustrating. It's even tempting to think that something is fundamentally wrong, and the code will never work. Always remember that **every bug has an explanation**. Every bug is strange, but even a bug that appears to be a weird mystery is caused by something in your code—so you can fix it. Because like Sherlock Holmes once said, "It is a mistake to confound strangeness with mystery."

Bullet Points

- You'll use many different **controls** to build your app's user interface (or UI). The UI is the part of the application that your user interacts with.
- The C# code for a page in a MAUI app is called **code-behind**. The XAML code and the C# code in the code-behind file work together to make the page work.
- The x:Name property gives your control a name you can use in your code.
- When you pay attention to **accessibility**, it makes your app—and your code!—better. **Semantic properties** help you make your apps accessible by providing descriptions and hints for people who use screen readers.
- In XAML you can have **nested controls**, or tags that contain other controls, so one control's start and end tag appear after the start tag and before the end of another tag.
- You can use nested **HorizontalStackLayout** and **VerticalStackLayout** controls to create more complex layouts.
- The first step in debugging is thinking: look at your code, think about how it works, and try to imagine where the bug might be.
- **Reproducing a bug** is an important tool that helps you fix it. When you're debugging, you're **running an experiment** every time you run your code. Run it as many times as it takes to understand what's going on.

Unity Lab #1
Explore C# With Unity

Welcome to your first **Head First C# Unity Lab**. Writing code is a skill, and like any other skill, getting better at it takes **practice and experimentation**. Unity will be a really valuable tool for that.

Unity is a cross-platform game development tool that you can use to make professional-quality games, simulations, and more. It's also a fun and satisfying way to **get practice with the C# tools and ideas** you'll learn throughout this book. We designed these short, targeted labs to **reinforce** the concepts and techniques you just learned to help you hone your C# skills. These labs are optional, but valuable practice... **even if you aren't planning to write games in C#**.

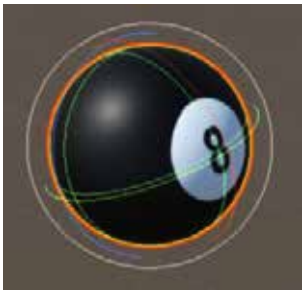
In this first lab, you'll get started with Unity. You'll get oriented with the Unity editor, and you'll start creating and manipulatintg 3D shapes. That will lay down a foundation to write code in the next lab.

Unity is a powerful tool for game design

Welcome to the world of Unity, a complete system for designing professional-quality games—both two-dimensional (2D) and three-dimensional (3D)—as well as simulations, tools, and projects. Unity includes many powerful things, including...

A cross-platform game engine

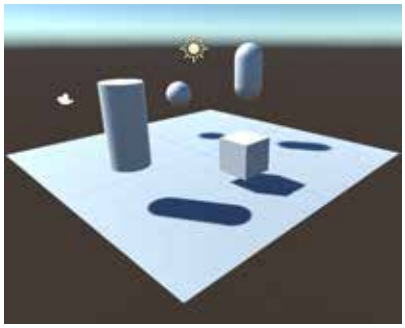
A **game engine** displays the graphics, keeps track of the 2D or 3D characters, detects when they hit each other, makes them act like real-world physical objects, and much, much more. Unity will do all of these things for the 3D games you build throughout this book.



A powerful 2D and 3D scene editor

You'll be spending a lot of time in the Unity editor. It lets you edit levels full of 2D or 3D objects, with tools that you can use to design complete worlds for your games. Unity games use C# to define their behavior, and the Unity editor integrates with Visual Studio to give you a seamless game development environment.

While these Unity Labs will concentrate on C# development in Unity, if you're a visual artist or designer, the Unity editor has many artist-friendly tools designed just for you. Check them out here: https://unity.com/solutions/artist-designers



An ecosystem for game creation

Beyond being an enormously powerful tool for creating games, Unity also features an ecosystem to help you build and learn. The Learn Unity page (https://unity.com/learn) has valuable self-guided learning resources, and the Unity forums (https://forum.unity.com) help you connect with other game designers and ask questions. The Unity Asset Store (https://assetstore.unity.com) provides free and paid assets like characters, shapes, and effects that you can use in your Unity projects.

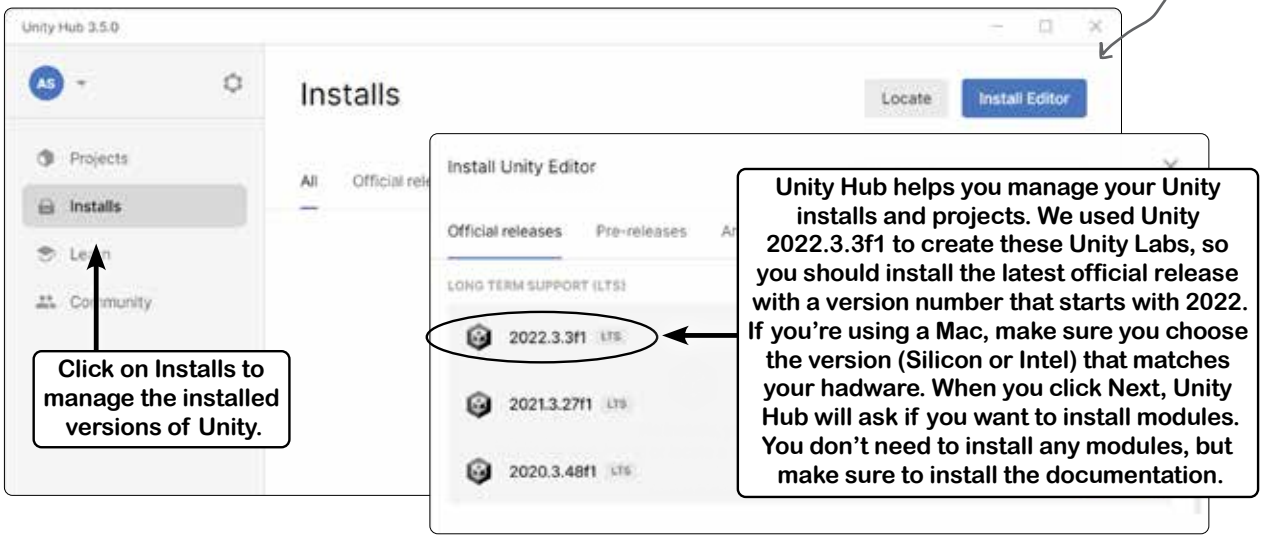
Our Unity Labs will focus on using Unity as a tool to explore C#, and practicing with the C# tools and ideas that you've learned throughout the book.

The *Head First C#* Unity Labs are laser-focused on a **developer-centric learning path**. The goal of these labs is to help you ramp up on Unity quickly, with the same focus on brain-friendly just-in-time learning you'll see throughout *Head First C#* to ***give you lots of targeted, effective practice with C# ideas and techniques***.

Download Unity Hub

All of the screenshots in this book were taken with the free Personal Edition of Unity. You'll need to enter your unity.com username and password into Unity Hub to activate your license.

Unity Hub is an application that helps you manage your Unity projects and your Unity installations, and it's the starting point for creating your new Unity project. Start by downloading Unity Hub from https://unity.com/developer-tools—then install it and run it.



Unity Hub lets you install multiple versions of Unity on the same computer, so you should install the same version that we used to build these labs. **Click the Install Editor button** and install the latest official release that starts with ***Unity 2022***—that's the same version we used to take the screenshots in these labs. Once it's installed, make sure that it's set as the preferred version.

The Unity installer may prompt you to install a different version of Visual Studio. You can have multiple installations of Visual Studio on the same computer too, but if you already have one version of Visual Studio installed there's no need to make the Unity installer add another one.

You can learn more about installing Unity Hub on Windows and macOS here: https://docs.unity3d.com/Manual/GettingStartedInstallingUnity.html

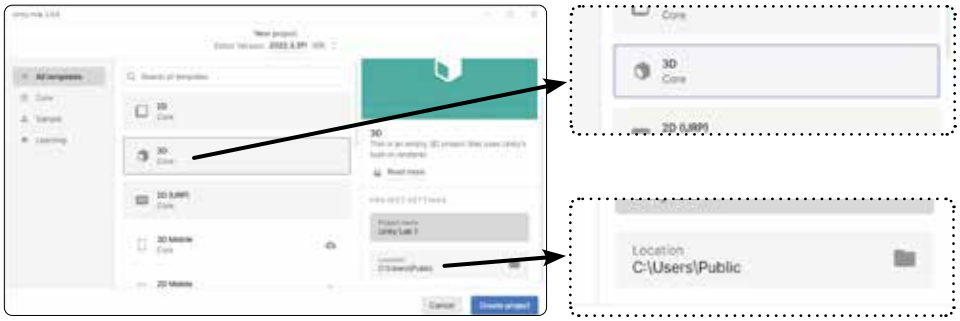
Unity Hub lets you have many Unity installs on the same computer. So even if there's a newer version of Unity available, you can use Unity Hub to install the version we used in the Unity Labs.

Unity Hub may look a little different

*The screenshots in this book were taken with **Unity 2022.3 in dark mode** and **Unity Hub 3.5.0 in light mode**. You can use Unity Hub to install many different versions of Unity on the same computer, but you can only install the latest version of Unity Hub. The Unity development team is constantly improving Unity Hub and the Unity editor, so it's possible that what you see won't quite match what's shown on this page. We update these Unity Labs for newer editions of Head First C#. We'll add PDFs of updated labs to our GitHub page: https://github.com/head-first-csharp/fifith-edition*

Use Unity Hub to create a new project

Click the New project button on the Project page in Unity Hub to create a new Unity project. Name it ***Unity Lab 1***, make sure the 3D template is selected, and check that you're creating it in a sensible location (usually the Unity Projects folder underneath your home directory).

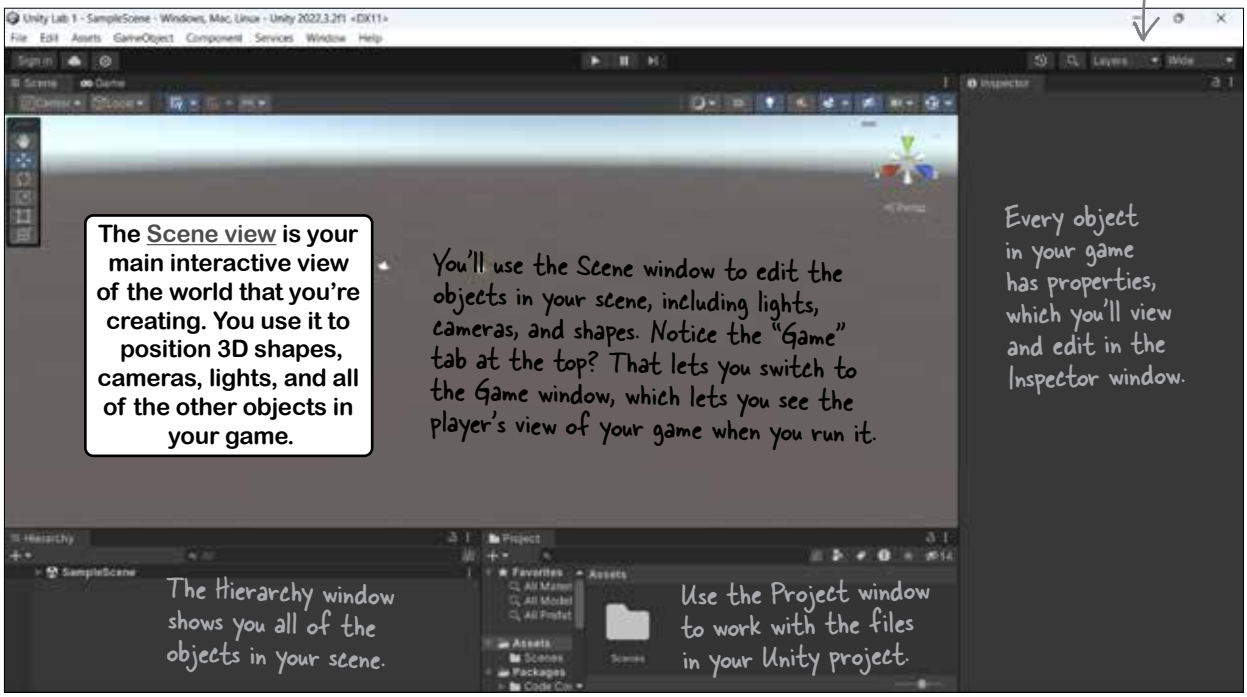


Click Create Project to create the new folder with the Unity project. When you create a new project, Unity generates a lot of files (just like Visual Studio does when it creates new projects for you). It could take Unity a minute or two to create all of the files for your new project.

Work with your project in the Unity editor

Once your project is created, it will load in the **Unity editor**, a powerful tool that you'll use to create 3D environments. Here are some important parts of the Unity editor:

You can use this dropdown to change the layout of the Unity editor.



OK! You're all ready to get started on your first Unity project.

Take control of the Unity layout

The Unity editor is like an IDE for all of the parts of your Unity project that aren't C#. You'll use it to work with scenes, edit 3D shapes, create materials, and so much more.

When you started up Unity, did you notice that your screen looked a little different from our screenshot? Just like in Visual Studio, the windows and panels in the Unity editor can be rearranged in many different layouts. We chose a layout that works well for screenshots in a book. We also chose dark mode, which we think is easier to read when these pages are printed.

You can download PDFs of all of these Unity Labs and print them out if that makes it easier for you to follow along.

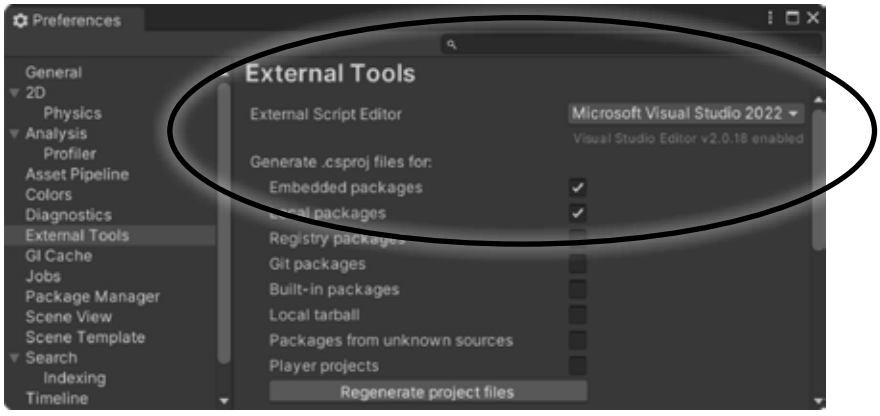
Choose the Wide layout to match our screenshots

We've chosen the Wide layout because it works well for the screenshots in these labs. Find the Layout dropdown in the toolbar and choose Wide so your Unity editor looks like ours.

Once you change the layout with the Layout dropdown on the right side of the toolbar, the dropdown changes its label to match the layout that you selected.

Set up Unity to work with Visual Studio

The goal of these Unity Labs is to use give you an **exciting and fun way to explore C#**. Luckily, Unity editor works hand-in-hand with the Visual Studio IDE to make it easy to edit and debug the code for your games. Open the **Unity Preferences Window** (on Windows choose Preferences... from the Edit menu, on a Mac choose Settings... ⌘, from the Unity menu). Click on External Tools on the left, click the External Script Editor dropdown, and **choose Visual Studio 2022** from the list of options.



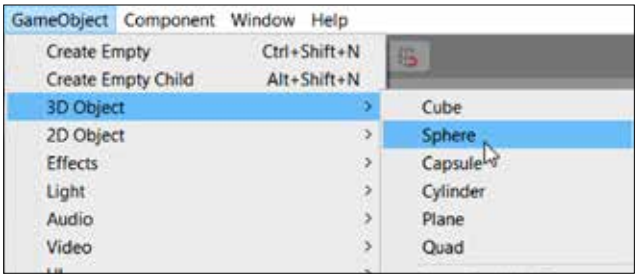
You can use Visual Studio to debug the code in your Unity games. Just choose Visual Studio as the external script editor in Unity's preferences.

If you don't see Visual Studio in the External Script Editor dropdown, choose Browse... and navigate to Visual Studio. On Windows it's normally an executable called devenv.exe in the folder C:\Program Files\Microsoft Visual Studio\2022\Community\Common7\IDE\. On a Mac it's typically an app called Visual Studio in the Applications folder.

Your scene is a 3D environment

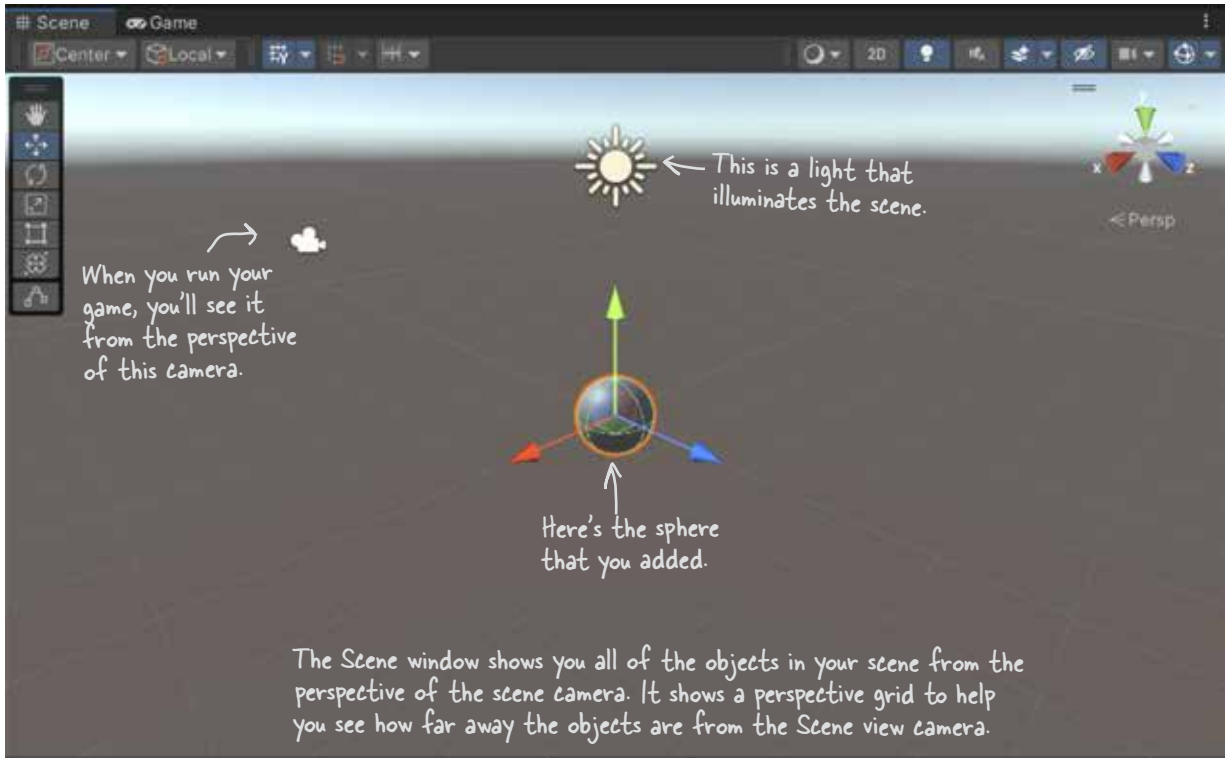
As soon as you start the editor, you're editing a **scene**. You can think of scenes as levels in your Unity games. Every game in Unity is made up of one or more scenes. Each scene contains a separate 3D environment, with its own set of lights, shapes, and other 3D objects. When you created your project, Unity added a scene called SampleScene, and stored it in a file called *SampleScene.unity*.

Add a sphere to your scene by choosing **GameObject >> 3D Object >> Sphere** from the menu:



These are called Unity's "primitive objects." We'll be using them a lot throughout these Untiy Labs.

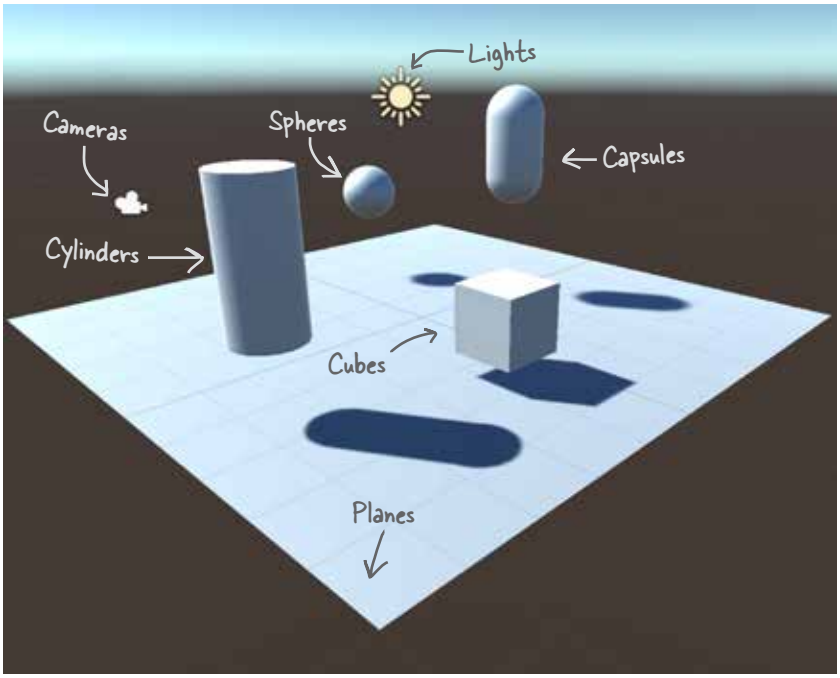
A sphere will appear in your Scene window. Everything you see in the Scene window is shown from the perspective of the **Scene view camera**, which "looks" at the scene and captures what it sees.



Unity games are made with GameObjects

When you added a sphere to your scene, you created a new **GameObject**. The GameObject is a fundamental concept in Unity. Every item, shape, character, light, camera, and special effect in your Unity game is a GameObject. Any scenery, characters, and props that you use in a game are represented by GameObjects.

In these Unity Labs, you'll build games out different kinds of GameObjects, including:



Each GameObject contains a number of **components** that provide its shape, set its position, and give it all of its behavior. For example:

- *Transform components* determine the position and rotation of the GameObject.
- *Material components* change the way the GameObject is **rendered**—or how it's drawn by Unity—by changing the color, reflection, smoothness, and more.
- *Script components* use C# scripts to determine the GameObject's behavior.

ren-der, verb.
to represent or depict artistically.
*Michelangelo **rendered** his favorite model with more detail than he used in any of his other drawings.*

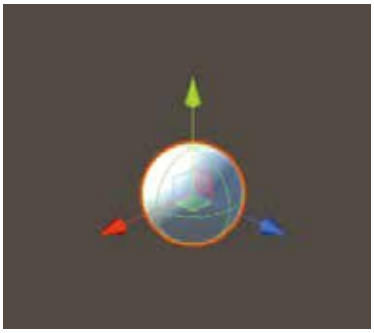
GameObjects are the fundamental objects in Unity, and components are the basic building blocks of their behavior. The Inspector window shows you details about each GameObject in your scene and its components.

Use the Move Gizmo to move your GameObjects

The Tools panel lets you choose **Transform tools**. If the Move Tool isn't selected, click on the sphere that you just added, then click the Move Tool in the **Tools Overlay** to select it.

The Tools overlay lets you choose tools to manipulate GameObjects. You'll use the Move Tool to move your sphere around the scene. In the Wide view the Tools overlay is vertical. You can right-click the two lines at the top to change its orientation so it's horizontal, or you can drag it to the toolbar or the side of the window to dock it.

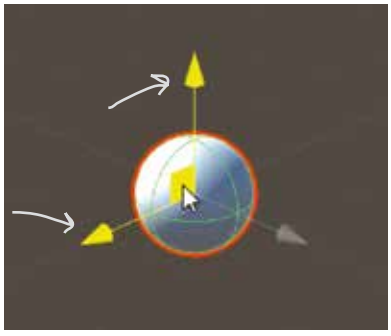
The **Move Tool** lets you use the **Move Gizmo** to move GameObjects around the 3D space. You should see red, green, and blue arrows and a cube appear in the middle of the window. This is the Move Gizmo, which you can use to move the selected object around the scene.



Using the Move Tool displays displays the Move Gizmo as arrows and a cube on top of the GameObject that's currently selected. When you click the sphere and then choose the Move Tool, you'll see the Move Gizmo appear on the sphere. Click anywhere else in the scene to deselect the sphere and the Move Gizmo goes away.

Move your mouse cursor over the cube at the center of the Move Gizmo—notice how each of the faces of the cube lights up as you move your mouse cursor over it? Click on the upper-left face and drag the sphere around. You're moving the sphere in the X-Y plane.

When you click on the upper-left face of the cube in the middle of the Move Gizmo, its X and Y arrows light up and you can drag your sphere around the X-Y plane in your scene.



The Move Gizmo lets you move GameObjects along any axis or plane of the 3D space in your scene.

Move your sphere around the scene to get a feel for how the Move Gizmo works. Click and drag each of the three arrows to drag it along each plane individually. Try clicking on each of the faces of the cube in the Scene Gizmo to drag it around all three planes. Notice how the sphere gets smaller as it moves farther away from you—or really, the scene camera—and larger as it gets closer.

The Inspector shows your GameObject's components

As you move your sphere around the 3D space, watch the **Inspector window**, which is on the right side of the Unity editor if you're using the Wide layout. Look through the Inspector window—you'll see that your sphere has four components labeled Transform, Sphere (Mesh Filter), Mesh Renderer, and Sphere Collider.

Every GameObject has a set of components that provide the basic building blocks of its behavior, and every GameObject has a **Transform component** that drives its location, rotation, and scale.

You can see the Transform component in action as you use the Move Gizmo to drag the sphere around the X-Y plane. Watch the X and Y numbers in the Position row of the Transform component change as the sphere moves.

If you accidentally deselect a GameObject, just click on it again. If it's not visible in the scene, you can select it in the Hierarchy window, which shows all of the GameObjects in the scene. When you reset the layout to Wide, the Hierarchy window is in the lower-left corner of the Unity editor.

Did you notice the grid in your 3D space? As you're dragging the sphere around, hold down the Control key. That causes the GameObject that you're moving to snap to the grid. You'll see the numbers in the Transform component move by whole numbers instead of small decimal increments.

Try clicking on each of the other two faces of the Move Gizmo cube and dragging to move the sphere in the X-Z and Y-Z planes. Then click on the red, green, and blue arrows and drag the sphere along just the X, Y, or Z axis. You'll see the X, Y, and Z values in the Transform component change as you move the sphere.

Now **hold down Shift** to turn the cube in the middle of the Gizmo into a square. Click and drag on that square to move the sphere in the plane that's parallel to the Scene view camera.

Once you're done experimenting with the Move Gizmo, use the sphere's Transform component context menu to reset the component to its default values. Click the **context menu button** (⋮) at the top of the Transform panel and choose Reset from the menu.

Use the context menu to reset a component. You can either click the three dots or right-click anywhere in the top line of the Transform panel in the Inspector window to bring up the context menu.

The position will reset back to [0, 0, 0].

You can learn more about the tools and how to use them to position GameObjects in the Unity Manual. Click Help >> Unity Manual and search for the "Positioning GameObjects" page.

Save your scene often! Use File >> Save or Ctrl+S / ⌘S to save the scene right now.

Add a material to your Sphere GameObject

Unity uses **materials** to provide color, patterns, textures, and other visual effects. Your sphere looks pretty boring right now because it just has the default material, which causes the 3D object to be rendered in a plain, off-white color. Let's make it look like a billiard ball.

① Select the sphere.

When the sphere is selected, you can see its material as a component in the Inspector window:

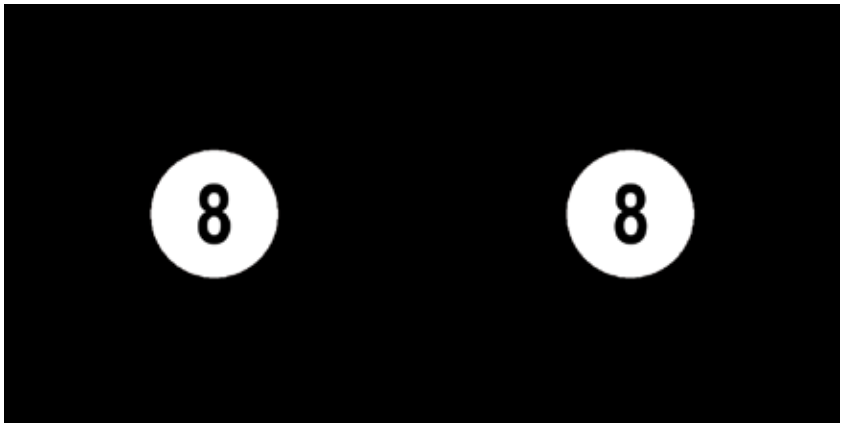
We'll make your sphere more interesting by adding a **texture**—that's just a simple image file that's wrapped around a 3D shape, almost like you printed the picture on a rubber sheet and stretched it around your object.

② Go to our Billiard Ball Textures page on GitHub.

Go to https://github.com/head-first-csharp/fifth-edition and click on the *Billiard Ball Textures* link to browse a folder of texture files for a complete set of billiard balls.

③ Download the texture for the 8 ball.

Click on the file *8 Ball Texture.png* to view the texture for an 8 ball. It's an ordinary 1200 × 600 PNG image file that you can open in your favorite image viewer.



We designed this image file so that it looks like an 8 ball when Unity "wraps" it around a sphere.

Download the file into a folder on your computer.

(You might need to right-click on the Download button to save the file, or click Download to open it and then save it, depending on your browser.)

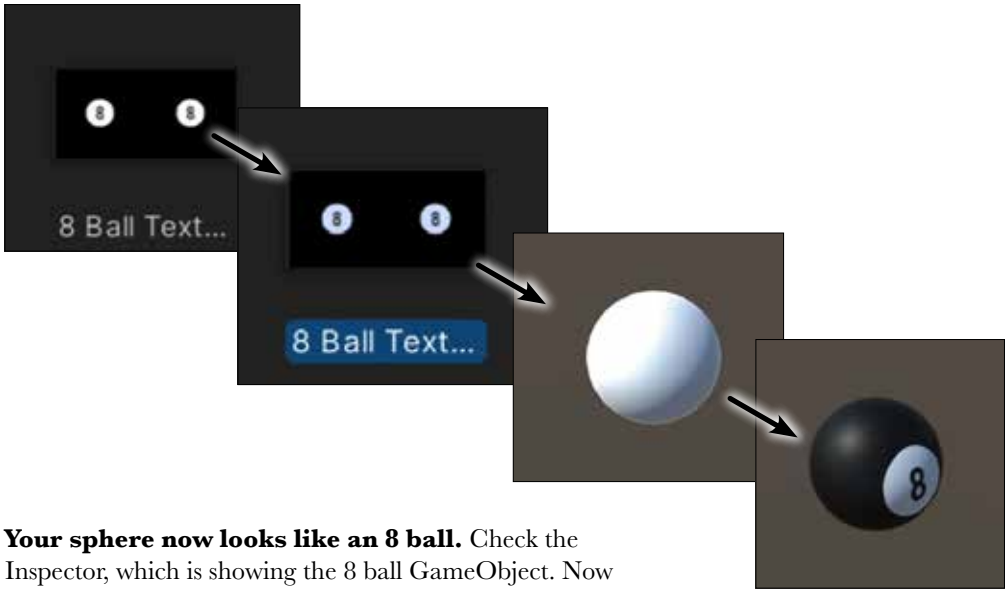
④ Import the 8 Ball Texture image into your Unity project.

Right-click on the Assets folder in the Project window, choose **Import New Asset...** and import the texture file. You should now see it when you click on the Assets folder in the Project window.

You right-clicked inside the Assets folder in the Project window to import the new asset, so Unity imported the texture into that folder.

⑤ Add the texture to your sphere.

Now you just need to take that texture and "wrap" it around your sphere. Click on 8 Ball Texture in the Project window to select it. Once it's selected, **drag it onto your sphere**.



Your sphere now looks like an 8 ball. Check the Inspector, which is showing the 8 ball GameObject. Now it has a new material component:

Check your Assets window again. Unity created a new Materials folder in it and added a material called 8 Ball Texture.

I'M LEARNING C# FOR MY JOB, NOT TO WRITE VIDEO GAMES. WHY SHOULD I CARE ABOUT UNITY?

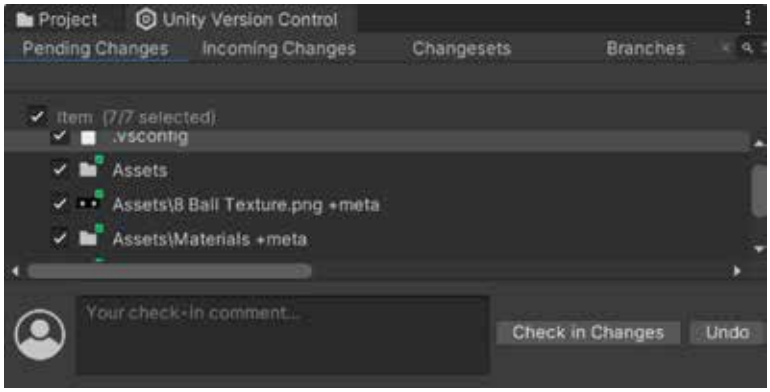
Unity is a great way to really "get" C#.

Programming is a skill, and the more practice you get writing C# code, the better your coding skills will get. That's why we designed the Unity Labs throughout this book to specifically **help you practice your C# skills** and reinforce the C# tools and concepts that you learn in each chapter. As you write more C# code, you'll get better at it, and that's a really effective way to become a great C# developer. Neuroscience tells us that we learn more effectively when we experiment, so we designed these Unity Labs with lots of options for experimentation, and suggestions for how you can get creative and keep going with each lab.

But Unity gives us an even more important opportunity to help get important C# concepts and techniques into your brain. When you're learning a new programming language, it's really helpful to see how that language works with lots of different platforms and technologies. That's why we included both console apps and WPF apps in the main chapter material, and in some cases even have you build the same project using both technologies. Adding Unity to the mix gives you a third perspective, which can really accelerate your understanding of C#.

Do you want to make sure your Unity projects are always backed up? Try Unity Version Control.

Unity Version Control is a version control system that lets you back up your projects to cloud storage that comes free with your Unity account—and it's ***built right into the Unity editor***, which makes it easy for you to use.



Click the Unity Version Control button in the toolbar to open the Unity Version Control window. The first time you use it, you'll get an option to log in or sign up. When you sign in with your Unity ID, you'll get to a web page where you can sign into your Unity account, then you can up for the free Unity VCS level and join your default organization. Then you can check in changes any time you want.

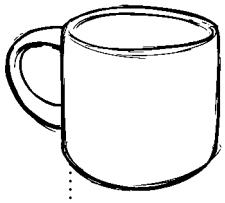
Go to the Head First C# GitHub page for a free PDF that gives you step-by-step instructions for setting Unity Version Control. https://github.com/head-first-csharp/fifth-edition/

Rotate your sphere

Click the **Rotate tool** in the toolbar. You can use the Q, W, E, R, T, and Y keys to quickly switch between the Transform tools—press E and W to toggle between the Rotate tool and Move Tool.

We switched the Tools overlay to a horizontal view by right-clicking on the two lines and choosing Horizontal. Try it out.

1. **Click on the sphere.** Unity will display a wireframe sphere Rotate Gizmo with red, blue, and green circles. Click the red circle and drag it to rotate the sphere around the X axis.



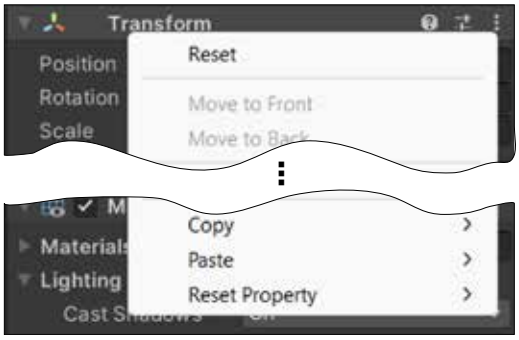
Relax

It's easy to reset your windows and scene camera.

If you change your Scene view so you can't see your sphere anymore, or if you drag your windows out of position, just use the layout dropdown in the upper-right corner to reset the Unity editor to the Wide layout. It will reset the window layout and move the Scene view

2. **Click and drag the green and blue circles to rotate around the Y and Z axes.** The outer white circle rotates the sphere along the axis coming out of the Scene view camera. Watch the Rotation numbers change in the Inspector window.

3. **Open the context menu of the Transform panel in the Inspector window.** Click Reset, just like you did before. It will reset everything in the Transform component back to default values—in this case, it will change your sphere's rotation back to [0, 0, 0].



Click the three dots (or right-click anywhere in the header of the Transform panel) to bring up the context menu. The Reset option at the top of the menu resets the component to its default values.

Use these options from further down in the context menu to reset the position and rotation of a GameObject.

Use File >> Save or Ctrl+S / ⌘S to save the scene right now. Save early, save often!

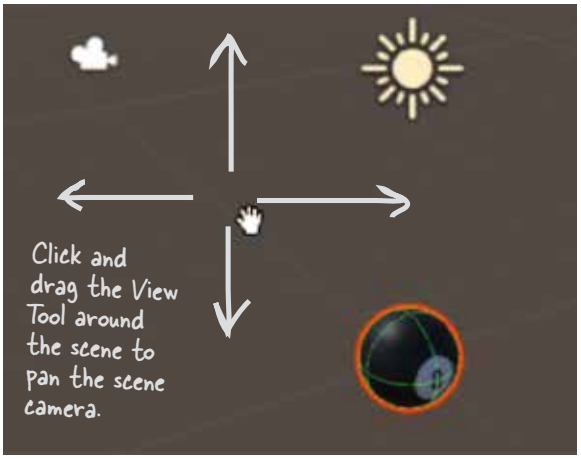
Move the Scene view camera with the View Tool and Scene Gizmo

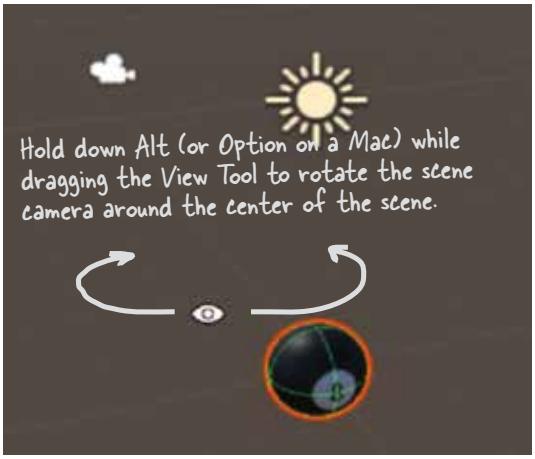
Use the mouse scroll wheel or scroll feature on your trackpad to zoom in and out, and toggle between the Move and Rotate Gizmos. Notice that the sphere changes size, but the Gizmos don't. The Scene window in the editor shows you the view from a virtual **camera**, and the scroll feature zooms that camera in and out.

Press Q to select the **View Tool**, or choose it from the toolbar. Your cursor will change to a hand.

Hold down Alt (Windows) or Option (Mac) while dragging and the View Tool turns into an eye and rotates the view around the center of the window

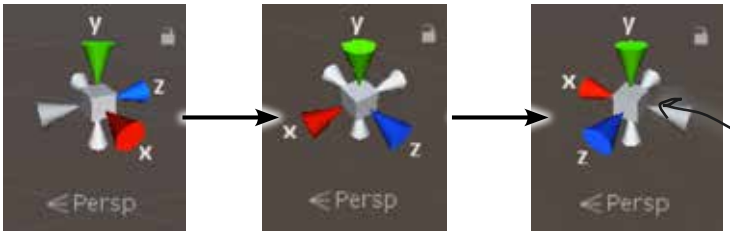
The View Tool pans around the scene by changing the position and rotation of the scene camera. When the View Tool is selected, you can click anywhere in the scene to pan.





When the View Tool is selected, you can ***pan*** the scene camera by **clicking and dragging**, and you can ***rotate*** it by holding **down Alt (or Option) and dragging**. Use the **mouse scroll wheel** to zoom. Holding down the **right mouse button** lets you ***fly through the scene*** using the W-A-S-D keys.

When you rotate the scene camera, keep an eye on the **Scene Gizmo** in the upper-right corner of the Scene window. The Scene Gizmo always displays the camera's orientation—check it out as you use the View Tool to move the Scene view camera. Click on the X, Y, and Z cones to snap the camera to an axis.



Click any of the cones in the Scene Gizmo to snap the camera to an axis. Drag them around to rotate the camera.

The Unity Manual has great tips on navigating scenes: https://docs.unity3d.com/Manual/SceneViewNavigation.html

Take a minute and look at this page—it's got some really useful stuff.

there are no Dumb Questions

Q: I'm still not clear on exactly what a component is. What does it do, and how is it different from a GameObject?

A: A GameObject doesn't actually do much on its own. All a GameObject really does is serve as a *container* for components. When you used the GameObject menu to add a Sphere to your scene, Unity created a new GameObject and added all of the components that make up a sphere, including a Transform component to give it position, rotation, and scale, a default Material to give it its plain white color, and a few other components to give it its shape, and help your game figure out when it bumps into other objects. These components are what make it a sphere.

Q: So does that mean I can just add any component to a GameObject and it gets that behavior?

A: Yes, exactly. When Unity created your scene, it added two GameObjects, one called Main Camera and another called Directional Light. If you click on Main Camera in the Hierarchy window, you'll see that it has three components: a Transform, a Camera, and an Audio Listener. If you think about it, that's all a camera actually needs to do: be somewhere, and pick up visuals and audio. The Directional Light GameObject just has two components: a Transform and a Light, which casts light on other GameObjects in the scene.

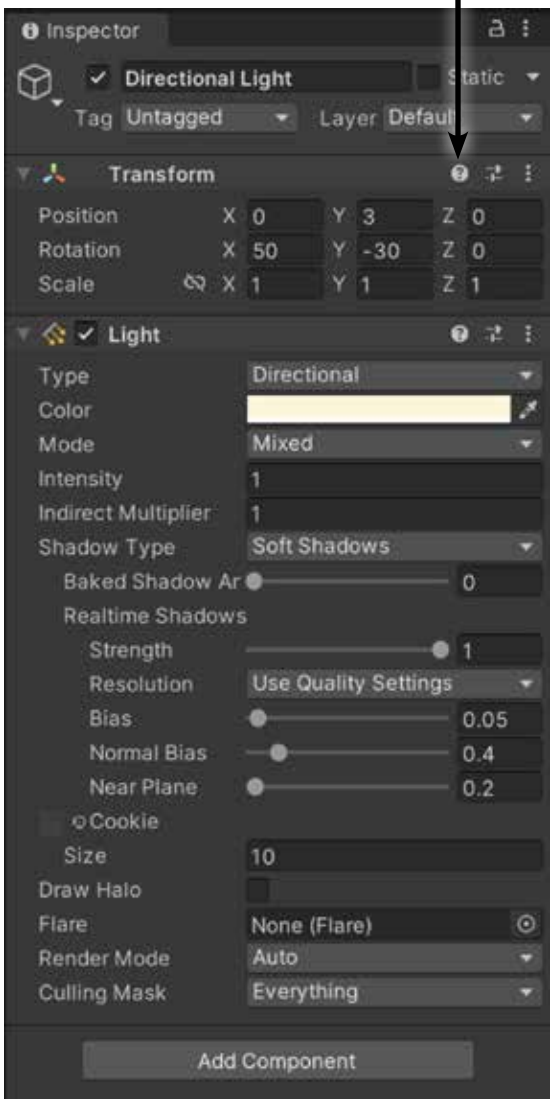
Q: If I add a Light component to any GameObject, does it become a light?

A: Yes! A light is just a GameObject with a Light component. If you click on the Add Component button at the bottom of the Inspector and add a Light component to your ball, it will start emitting light. If you add another GameObject to the scene, it will reflect that light.

Q: It sounds like you're being careful with the way you talk about light. Is there a reason you talk about emitting and reflecting light? Why don't you just say that it glows?

A: Because there's a difference between a GameObject that emits light and one that glows. If you add a Light component to your ball, it will start emitting light—but it won't look any different, because the Light only affects other GameObjects in the scene that reflect its light. If you want your GameObject to glow, you'll need to change its material or use another component that affects how it's rendered.

You can click on the Help icon for any component to bring up the Unity Manual page for it.



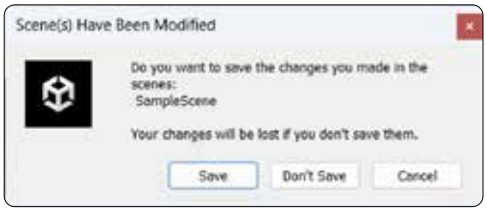
When you click on the Directional Light GameObject in the Hierarchy window, the Inspector shows you its components. It just has two: a Transform component that provides its position and rotation and a Light component that actually casts the light. What do you think you'll use the Add Component button for?

Get creative!

We built these Unity Labs to give you a **platform to experiment on your own with C#** because that's the single most effective way for you to become a great C# developer. This lab ***lays down the foundation*** to start writing Unity code—which you'll do in the next lab. At the end of each Unity Lab, we'll include suggestions for things that you can try on your own. Take some time to experiment with everything you just learned before moving to the next chapter:

- Add a few more spheres to your scene. Try using some of the other billiard ball maps. You can download them all from the same location where you downloaded *8 Ball Texture.png* from.
- Try adding other shapes by choosing Cube, Cylinder, or Capsule from the GameObject >> 3D Object menu.
- Experiment with using different images as textures. See what happens to photos of people or scenery when you use them to create textures and add them to different shapes.
- Can you create an interesting 3D scene out of shapes, textures, and lights?

When you're ready to move on to the next chapter, make sure you save your project, because you'll come back to it in the next lab.. Unity will prompt you to save when you quit.



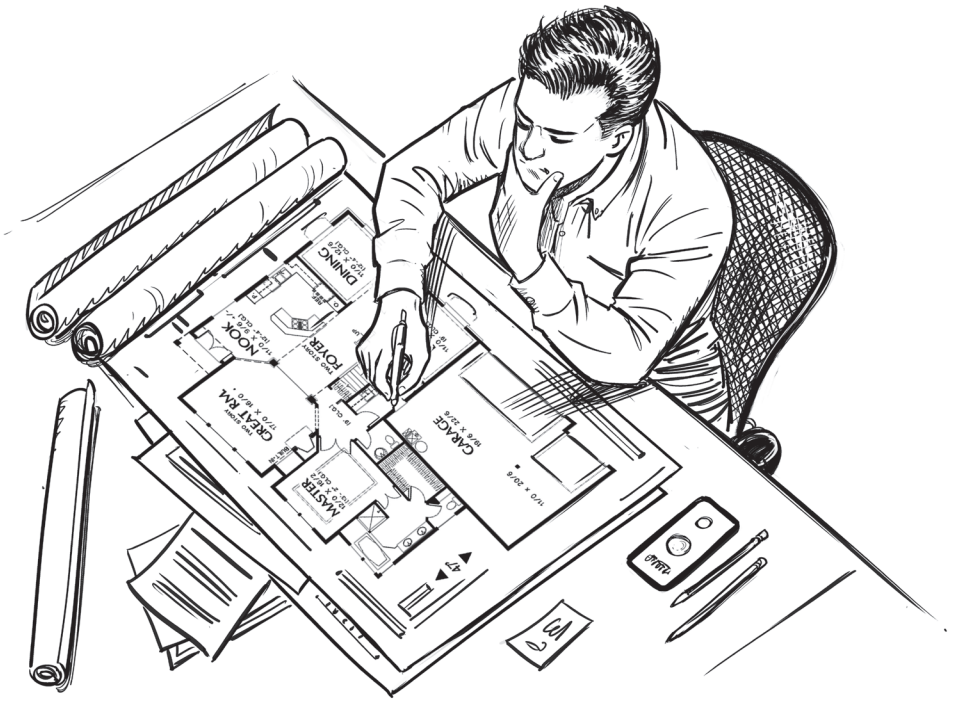
The more C# code you write, the better you'll get at it. That's the most effective way for you to become a great C# developer. We designed these Unity Labs to give you a platform for practice and experimentation.

Bullet Points

- The **Scene view** is your main interactive view of the world that you're creating.
- When you select an object and use the **Move Tool**, Unity displays the **Move Gizmo** lets you move objects around your scene.
- The **View Tool** lets you pan around the scene. The **Scene Gizmo** always displays the camera's orientation.
- Unity uses **materials** to provide color, patterns, textures, and other visual effects.
- Some materials use **textures**, or image files wrapped around shapes.
- Your game's scenery, characters, props, cameras, and lights are all built from **GameObjects**.
- GameObjects are the fundamental objects in Unity. **Components** are the building blocks for their behavior.
- Every GameObject has a **Transform component** that provides its position, rotation, and scale.
- The **Project window** gives you a folder-based view of your project's assets, including C# scripts and textures.
- The **Hierarchy window** shows all of the GameObjects in the scene.
- **Unity Version Control System (VCS)** is an easy way to back up projects to free cloud storage that comes with a Unity Personal account.
- **GitHub for Unity** (https://unity.github.com) makes it easy to save your Unity projects in GitHub.

3 namespaces and classes

Organizing your code



Great developers keep their code and data organized.

What's the first thing you do when you're creating an app? You think about what **it's supposed to do**, whether you're solving a problem, creating a game, or just having fun. But it's not always obvious how individual statements fit into your app's bigger picture… and that's where **classes** come in. They let you **organize your code** aroudn the features you're creating and the problems the app needs to solve. Classes can help you **organize your data** too, by using them to create **objects** that represent any "thing" your app needs to know about—and the classes that you design serve as "blueprints" for the objects used in your app.

Classes help you organize your code

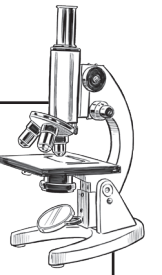
Let's be honest… you're going to write a lot of code thoughout this book. And as you keep going through the chapters, your projects will get bigger and bigger. This is a good thing!

Bigger apps present an interesting challenge. The app you built at the end of Chapter 2 had just a few methods. If you create a console app with the same number of methods, there's no reason not to put them all in Program.cs.

By the time you get to the end of the book, you'll be creating apps with *dozens* of methods. If you put all of those methods into one big Program.cs file, you'll have a hard time remembering which ones do what—and you'll drive yourself crazy trying to sleuth out bugs!

Luckily, C# has an answer for this organizational challenge. Your C# code is organized into **classes**, or units of code that contain methods. You could still put all of your methods into one big class, and many small apps could have just one class. But when you have a lot of code, it makes sense to ***organize your classes based on what they do***. When your classes are organized in a way that's intuitive, it helps you figure out where to add new methods —and it makes sleuthing out bugs a lot easier.

Anatomy of a C# app



Every C# program's code is structured in exactly the same way. All programs use namespaces, classes, and methods to make your code easier to manage.

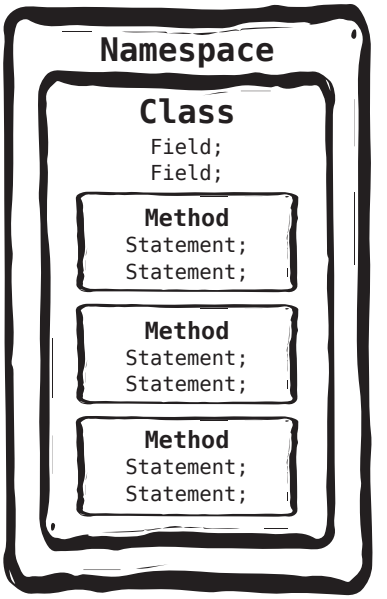
When you create your app, all of the code is inside a namespace. This helps keep your classes are separate from the ones that come with .NET.

A class contains a piece of your program. Some very small programs can have just one class, but most have more.

A class can have fields. A field is a variable, except that it's declared outside of the methods so all of the methods in the class can use it.

A class has one or more methods. Your methods must live inside a class. The order of the methods in the class file doesn't matter. Method 2 can just as easily come before method 1

Methods are made up of statements—like ones you used in your apps in the last two chapters.



If code is useful, classes can help you reuse it

Developers have been reusing code since the earliest days of programming, and it's not hard to see why. If you've written a class for one program, and you have another program that needs code that does exactly the same thing, then it makes sense to **reuse** the same class in your new program. So if we were going to build an app called PetManager, we might organize the code using classes called Dog and Cat.

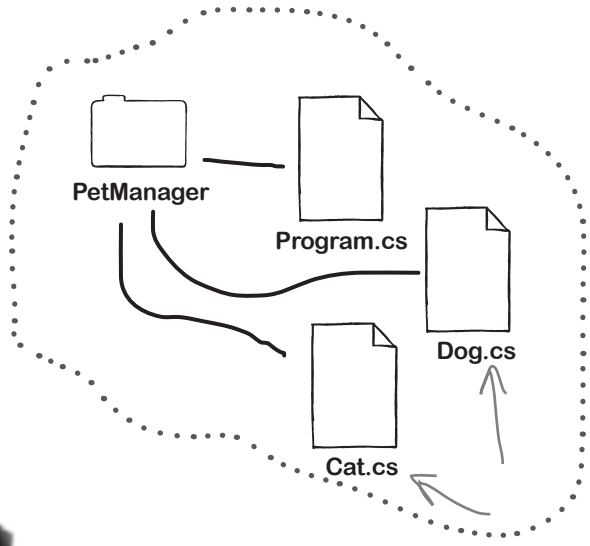
We built the Dog and Cat classes for our PetManagerApp console app...

Dog.cs

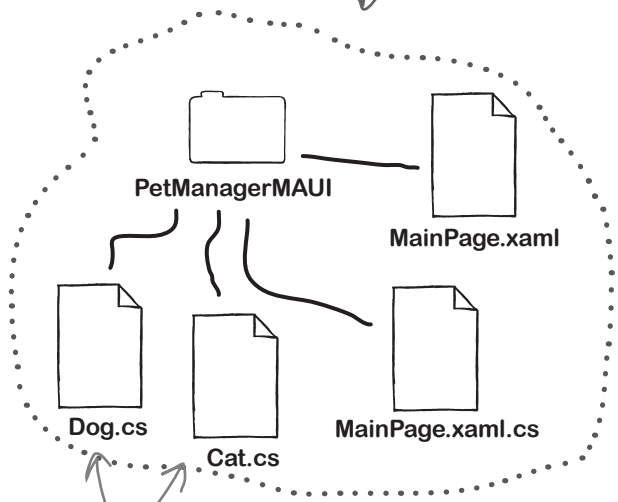
```
namespace Pets {

  class Dog {

    public void Bark() {
       // statements
    }

  }

}
```

Cat.cs

```
namespace Pets {

  class Cat {

    public void Meow() {
       // statements
    }

  }

}
```



...but we discovered we needed exactly the same classes in the MAUI version of our PetManager app, so we reused them.



Since we put our classes in the Pets namespace, we just had to copy the file into the new project and add "using Pets;" any time we wanted to use the Dog or Cat classes.

These are the same files with the Dog and Cat classes.

Some methods take parameters and return a value

You've seen methods that do things, like the SetUpGame method in Chapter 1 that sets up your game. Methods can do more than that: they can use **parameters** to get input, do something with that input, and then generate output with a **return value** that can be used by the statement that called the method.

Parameters are values that the method uses as input. They're declared as variables that are included in the method declaration (between the parentheses). The return value is a value that's calculated or generated inside the method, and sent back to the statement that called that method. The type of the return value (like *string* or *int*) is called the **return type**. If a method has a return type, then it must use a **return statement**.

Here's an example of a method with two int parameters and an int return type:

```
int Multiply(int factor1, int factor2)
{
    int product = factor1 * factor2;
    return product;
}
```

The return type is int, so the method must return an int value.

This method takes two int parameters called factor1 and factor2 as input. They're treated just like int variables.

The return statement passes the value back to the statement that called the method.

The method takes two **parameters** called **`factor1`** and **`factor2`**. It uses the multiplication operator **`*`** to calculate the result, which it returns using the **`return`** keyword.

This code calls the Multiply method and stores the result in a variable called **`area`**:

```
int height = 179;
int width = 83;
int area = Multiply(height, width);
```

You can pass values like 3 and 5 to methods, like this: Multiply(3, 5)—but you can also use variables when you call your methods. It's fine if the variable names don't match the parameter names.

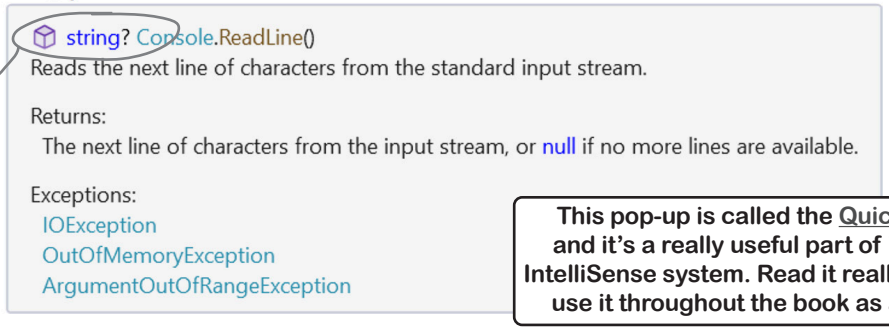
A method's parameters let you give it information that it can use, and its return value lets you use the result of the method in the statement that called it.

Visual Studio helps you explore parameters and return values

In the next app, you'll be using a .NET method called **Console.ReadLine** to get a line that the user types into the console. When you add the line into your app, you can hover over it to see more about it:

```
Console.ReadLine();
```

This method returns a string? value that holds a line of text that the user typed.



This pop-up is called the Quick Info window, and it's a really useful part of Visual Studio's IntelliSense system. Read it really carefully—we'll use it throughout the book as a learning tool.

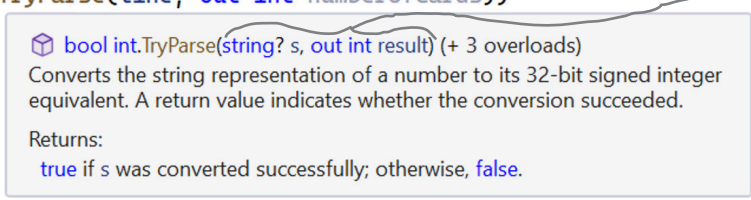
The IDE will pop up a box telling you what the method does. The very first thing in the box is its return type—in this case, it's a **`string?`** value that holds text. We'll learn a lot more about how strings work in the next chapter. In the meantime, what you need to know is that you call the method like this:

```
string? line = Console.ReadLine();
```

This calls the method to read a line of input, and stores the text the user typed in a variable called line.

You'll also use a method called int.TryParse(), which you'll use in an if statement like this:

```
if (int.TryParse(line, out int numberOfCards))
```



This method takes two parameters, a string? that contains text to turn into a number, and a result that the value gets saved into.

The IDE is telling you that it takes two parameters, a **`string?`** value and an **`out int value`**. (We'll learn a lot more about the **`out`** keywords later in the book—for now, we'll give you the code to use.).

Do this! → Since you're about to create methods that return values, right now is a perfect time to write some code and use the debugger to *really dig into how the* ***`return`*** *statement works*.

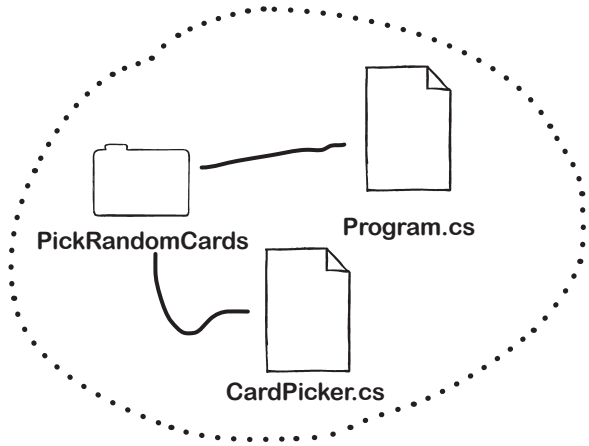
- What happens when a method is done executing all of its statements? See for yourself—open up one of the programs you've written so far, place a breakpoint inside a method, then keep stepping through it.
- When the method runs out of statements, *it* ***returns*** *to the statement that called it* and continues executing the next statement after that.
- A method can also include a **`return`** statement, which causes it to immediately exit without executing any of its other statements. Try adding a **`return`** statement in the middle of a method, then stepping over it.

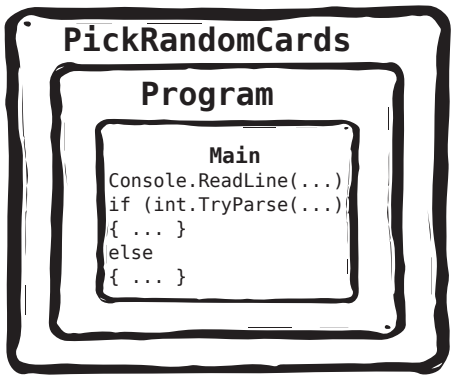
Let's build an app that picks random cards

In the first project in this chapter, you're going to build a console app called PickRandomCards that lets you pick random playing cards.

Let's use it as a way to **start using classes**. Here's what its structure will look like:

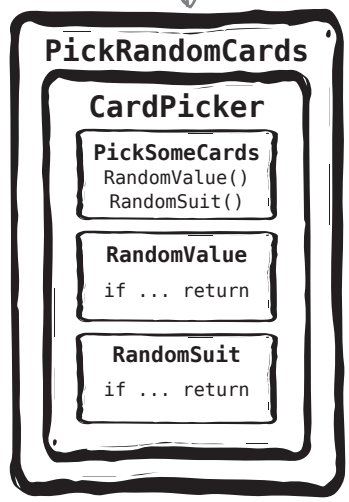
You're going to create a Console App that has a Main method instead of top-level statements (which we'll talk more about). Your Program.cs file will contain a class. That class will have a method called Main, which is the first thing that gets run when you start the app.





When you create your app you'll call it PickRandomCards, so Visual Studio will create a namespace for you that matches the name of the app.

Your Main method will have all the code that communictes with the user, displaying text and getting input. The code that has to do with picking random cards will be in a class called CardPicker.



The CardPicker class doesn't have any fields, and that's okay! We'll talk more about fields later in the chapter.

You'll use an array to store the cards

Your PickSomeCards method will use string values to represent playing cards. Let's say you want to use your CardPicker class to pick five random cards and store them in a variable called `cards`. Here's how you would do that:

```
string[] cards = CardPicker.PickSomeCards(5);
```

There's a lot going on in that line of code, so let's break it down. The PickSomeCards method is in the CardPicker class, but we're calling it from a top-level statement, so we need to use the class name to call it:

```
string[] cards = CardPicker.PickSomeCards(5);
```

We just learned about how methods can take parameters. The PickSomeCards method takes an int parameter, and we're passing the method the value 5 to tell it to pick five cards:

```
string[] cards = CardPicker.PickSomeCards(5);
```

The first part of the statement declares the cards variable. We just learned about return values—so the method will return a value to gets stored in the `cards` variable. But something looks different about it:

```
string[] cards = CardPicker.PickSomeCards(5);
```

The `cards` variable has a type that you haven't seen yet. Look closely at the type:

```
string[] cards = CardPicker.PickSomeCards(5);
```

The square brackets `[]` mean that it's an **array of strings**. Arrays let you use a single variable to store multiple values—in this case, strings with playing cards—which will get stored in the `cards` variable.

Here's an example of a string array that the PickSomeCards method might return:

```
{
    "10 of Diamonds",
    "6 of Clubs",
    "7 of Spades",
    "Ace of Diamonds",
    "Ace of Hearts",
}
```

This array of strings has five values in it. Each value is a separate string that has the name of a card.

After your array is generated, you can use a `foreach` loop to write each of the cards to the console:

```
foreach (string card in cards) {
    Console.WriteLine(card);
}
```

For the array above, running that foreach loop will generate this output:

```
10 of Diamonds
6 of Clubs
7 of Spades
Ace of Diamonds
Ace of Hearts
```

Create an app with a Main method

When you created console apps in the first two chapters, the IDE generated a two-line Program.cs file:

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

You may not have realized it at the time, but you were taking advantage of very useful feature of C# called **top-level statements** that lets you start creating a console app with a simple file that contains a set of statements that get executed in order.

Top-level statements are really convenient! A good way to understand what they do for you and how they work is to ***create a C# app that doesn't use top-level statements***.

Do this!

Create your PickRandomCards app without top-level statements

Use Visual Studio to **create a new console app called PickRandomCards**. But unlike previous chapters, when you're going through the steps to create the app keep an eye out for a checkbox like this and make sure that it's checked:

Make sure you check the "Do not use top-level statements box" when you create your app, otherwise your Program.cs file won't have a Program class with Main method.

When you create your new app, your Program.cs file should look like this:

```
namespace PickRandomCards
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

The very first statement that gets executed in an app is called its entry point. The entry point in an app with top-level statements is the first statement in Program.cs. In an app without top-level statements, the entry point is the Main method.

When the IDE created your app and generated the Program.cs file, **it added a class called Program**. This program was generated with one method called Main. The code inside the Main method is the familiar statement that prints "Hello, World!" to the console.

Run your app—it should look very familiar. Your new app does exactly the same thing as the "Hello, World!" app you created in Chapter 1. But instead of starting at the first statement in the Program.cs file, the first thing your app does is execute the Main method.

Your app can only have one entry point. If you add another class with a Main method, your code won't build.

Visual Studio remembers your checkbox choices.

The next time you create a Console App project in Visual Studio, it may remember that you checked the "Do not use top-level statements" box and check it agian for you. Make sure it's unchecked the next time you create a console app.

Top-Level Statements Behind the Scenes

Here's what happens when you use top-level statements

Here's the very first app that you created in Chapter 1:

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

The first line is a comment, so there's actually only one statement in this app. When you build the app, the compiler—the part of Visual Studio that turns your C# code into something that your operating system can execute—reads all of the lines in the top-level statement and adds them to a class. It creates a class that looks like the one that you just saw in your PickRandomCards app:

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

If you're using a Mac, you may not see the `internal` keyword in your PickRandomCards app. This is called an access modifier, and it's okay if Visual Studio doesn't add it because it's the default access modifier. You'll learn more about access modifiers in Chapter 9.

That looks a lot like the Program class that Visual Studio just created in your PickRandomCards app. But there's one difference—can you spot it?

Here's the Program class from your PickRandomCards app without top-level statements—we've made text that's the same a lighter color so you can see the difference:

```
namespace PickRandomCards
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

In an app with top-level statements, the code in your Program.cs file is not in a namespace. Every method in a C# program must be inside a class, but it's okay for classes to be outside of namespaces.

We'll learn more about how to work with namespaces work later in this chapter.

The rest of the console apps in this book will use top-level statements, so when you create the next one make sure to uncheck the "Do not use top-level statements" checkbox.

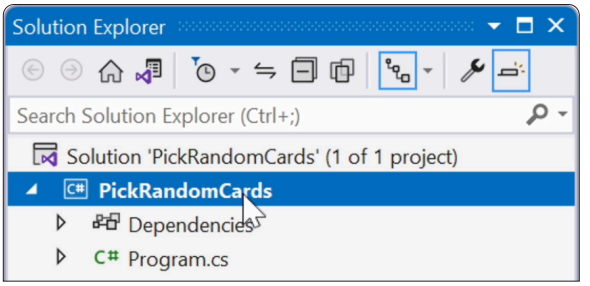
Add a class called CardPicker to your app

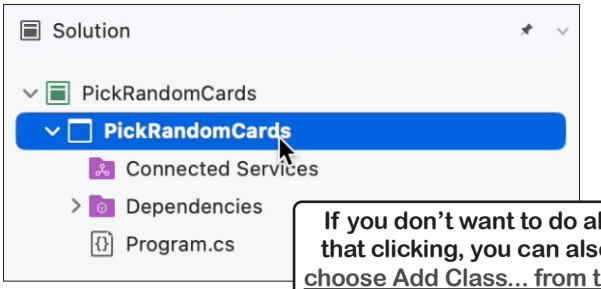
The next thing you'll do is add a class called CardPicker to your app. Here's a class diagram that shows the methods that you'll add to your CardPicker class:

This is a class diagram. It's a rectangle with the class name on top and a list of its methods on the bottom. Your CardPicker class will have three methods named PickSomeCards, RandomSuit, and RandomValue.

Do this!

Luckily, Visual Studio makes it easy for you to add classes to your project. Go to the Solution Explorer (in Windows) or Solution window (on a Mac) and Right-click on the PickRandomCards project. Make sure you're right-clicking on the project (the second line in the window), and not the solution (the top line):

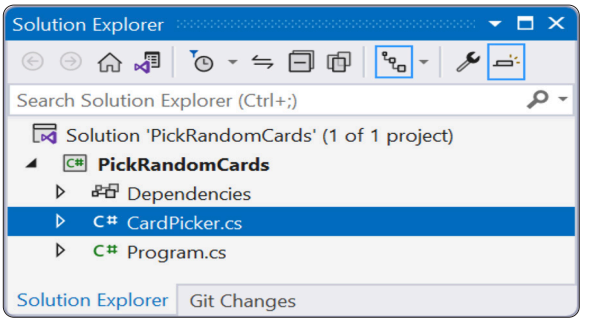


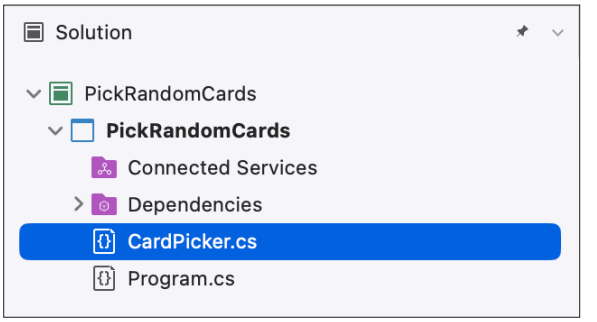


If you don't want to do all that clicking, you can also choose Add Class... from the Project menu to add a class, and jump straight to the window where Visual Studio prompts you for a filename.

When the right-mouse menu pops up, **choose Add >> Class… (Shift-Alt-C)** in Windows or **Add >> New Class…** in macOS. In Windows Visual Studio will prompt you for a filename for your new class—**choose CardPicker.cs** for the filename. On a Mac, it will ask you for the class name—**choose CardPicker**.

You should now see a file called CardPicker.cs in your project. It contains a class called Card Picker.





Open your new CardPicker class

Open the CardPicker class that you added to your project by double-clicking it in the Solution Explorer.

If you're using Visual Studio for Windows, the contents of your new CardPicker.cs file should look like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace PickRandomCards
{
    internal class CardPicker
    {


    }
}
```

If you're using Visual Studio for Mac, the contents of your new CardPicker.cs file should look like this:

```
using System;
namespace PickRandomCards
{
    public class CardPicker
    {
        public CardPicker()
        {

        }
    }
}
```

Visual Studio for Mac may generate this extra method for you. It's called a constructor, and you can either ignore it or delete it for this app. We'll talk about constructors in Chapter 5

The first line in the class is called the **class declaration**. Let's take a closer look at it:

Windows version

```
internal class CardPicker
```

Mac version

```
public class CardPicker
```

The first keyword in the declaration is the access modifier. Your CardPicker class might have the `internal` access modifier or the `public` access modifier, depending on which version of Visual Studio you're using. ***Either one will work just fine in this project***, and in all of the projects in the next few chapters. We'll come back to access modifiers later in the book.

After the class declaration are a matching pair of opening `{` and closing `}` curly brackets. Everything in your class goes between these brackets.

Add a new PickSomeCards method to your CardPicker class

Do this!

Put your cursor in the space between the curly brackets and carefully type in this method:

```
class CardPicker
{
    public static string[] PickSomeCards(int numberOfCards)
    {

    }
}
```

Make sure you include the public and static keywords. We'll talk more about them later in the chapter.

If you carefully entered this method declaration exactly as it appears here, you should see a red squiggly underline underneath PickSomeCards. What do you think it means?

Use the new keyword to create an array of strings

Do this!

Let's say you want to create an array of 5 strings and store it in a variable called myStrings. You can use the **new keyword** to create a new array of strings. You can create your array of 5 strings like this:

```
string myStrings = new string[5];
```

You can also use a variable, field, or method parameter instead of a number. Your PickSomeCards method has an parameter called numberOfCards—you'll use that parameter in your new statement: `new string[numberOfCards];`

The PickSomeCards method will pick five random cards. Each of the cards will have a random value and a random suit, so the class will also have two more methods that generate those the value and suit for each card.

1 Create a new array of strings and store it in a variable called pickedCards.

We saw earlier that the PickSomeCards method will return an array of strings, so the first thing we'll need is an array of strings to return. Add this line of code to your method:

```
public static string[] PickSomeCards(int numberOfCards)
{
    string[] pickedCards = new string[numberOfCards];
}
```

You'll see a red squiggly line under PickSomeCards. Visual Studio is telling you that your method is supposed to return something, but there's no corresponding return statement.

Now the method has a string array to work with.

2 Add a foreach loop to set the value of each card in the array.

Your method has an array of strings. Now it needs to set them. Add this foreach loop—it will call two methods called RandomValue and RandomSuit. Those methods don't exist yet, but that's okay.

```
public static string[] PickSomeCards(int numberOfCards)
{
    string[] pickedCards = new string[numberOfCards];
    for (int i = 0; i < numberOfCards; i++)
    {
        pickedCards[i] = RandomValue() + " of " + RandomSuit();
    }
}
```

The RandomValue and RandomSuit methods don't exist yet, so Visual Studio will warn you about them, too.

3 Finish the method by adding a return statement.

Add a return statement to send the pickedCards array back to the statement that called the method.

```
public static string[] PickSomeCards(int numberOfCards)
{
    string[] pickedCards = new string[numberOfCards];
    for (int i = 0; i < numberOfCards; i++)
    {
        pickedCards[i] = RandomValue() + " of " + RandomSuit();
    }
    return pickedCards;
}
```

Adding the return statement makes the warning on the method declaration line go away, but the warnings for the two method calls are still there.

③ Generate the RandomValue and RandomSuit methods.

In the last chapter, you generated a method called OperatorExamples. Follow exactly the same steps to **generate a method in the CardPicker class called RandomSuit**. Then do exactly the same thing to generate a method called RandomValue.

④ Implement the RandomSuit method.

Every card has a suit: Hearts, Clubs, Spades, or Diamonds. The RandomSuit method will pick a suit at random, store it in a string, and return it. It will use the same random number generator, Random.Shared, that you used in Chapter 1 to pick emoji from a list. The random number generator's Next method can take two parameters: random.Next(1, 5) returns a number that's at least 1 but less than 5—in other words, calling `Random.Shared.Next(1, 5)` returns a random number from 1 to 4.

Let's add code to your RandomSuit method that takes advantage of return statements to stop executing the method as soon as it finds a match:

```
private static string RandomSuit()
{
    // get a random number from 1 to 4
    int value = Random.Shared.Next(1, 5);
    // if it's 1 return the string Spades
    if (value == 1) return "Spades";
    // if it's 2 return the string Hearts
    if (value == 2) return "Hearts";
    // if it's 3 return the string Clubs
    if (value == 3) return "Clubs";
    // if we haven't returned yet, return the string Diamonds
    return "Diamonds";
}
```

You used a Random.Shared statement like this in Chapter 1 to pick random emoji from a list.

A method can have more than one return statement, and when it executes one of those statements it immediately returns—and does not execute any more statements in the method.

⑤ Implement the RandomValue method.

Every playing card can have one of 13 values—Ace, 2 through 10, Jack, Queen, or King. Here's the RandomValue method that generates a random value. Look closeley at it. Can you figure out how it works?

```
private static string RandomValue()
{
    int value = Random.Shared.Next(1, 14);
    if (value == 1) return "Ace";
    if (value == 11) return "Jack";
    if (value == 12) return "Queen";
    if (value == 13) return "King";
    return value.ToString();
}
```

Notice how your method returns `value.ToString()` and not just `value`? That's because `value` is an int variable, but the RandomValue method was declared with a string return type, so we need to convert `value` to a string. You can add `.ToString()` to any variable or value to convert it to a string.

The return statement causes your method to stop immediately and go back to the statement that called it.

Your finished CardPicker class

Here's the code for your finished CardPicker class. It lives inside the PickRandomCards namespace (which matches the name of the project) and has the methods we just added:

If you're using Visual Studio for Mac, your CardPicker class will have the public access modifier instead of internal. Also, we deleted the constructor method that Visual Studio for Mac added. It's okay if you didn't delete it.

```
namespace PickRandomCards
{
    internal class CardPicker
    {
        public static string[] PickSomeCards(int numberOfCards)
        {
            string[] pickedCards = new string[numberOfCards];
            for (int i = 0; i < numberOfCards; i++)
            {
                pickedCards[i] = RandomValue() + " of " + RandomSuit();
            }
            return pickedCards;
        }

        private static string RandomSuit()
        {
            // get a random number from 1 to 4
            int value = Random.Shared.Next(1, 5);
            // if it's 1 return the string Spades
            if (value == 1) return "Spades";
            // if it's 2 return the string Hearts
            if (value == 2) return "Hearts";
            // if it's 3 return the string Clubs
            if (value == 3) return "Clubs";
            // if we haven't returned yet, return the string Diamonds
            return "Diamonds";
        }

        private static string RandomValue()
        {
            int value = Random.Shared.Next(1, 14);
            if (value == 1) return "Ace";
            if (value == 11) return "Jack";
            if (value == 12) return "Queen";
            if (value == 13) return "King";
            return value.ToString();
        }
    }
}
```

We added these comments to help you understand how the RandomSuit method works. Try adding similar comments to the RandomValue method that explain how it works.

Brain Power

You used the `public` and `static` keywords when you added PickSomeCards. Visual Studio kept the `static` keyword when it generated the methods, and declared them as `private`, not `public`. What do you think these keywords do?

Exercise

Now that your CardPicker class has a method to pick random cards, you've got everything you need to finish your console app by **filling in the Main method**. You just need a few useful methods to make your console app read a line of input from the user and use it to pick a number of cards.

Useful method #1: Console.Write

You've already seen the Console.WriteLine method. Here's its cousin, Console.Write, which writes text to the console but doesn't add a new line at the end. You'll use it to display a message to the user:

```
Console.Write("Enter the number of cards to pick: ");
```

Useful method #2: Console.ReadLine

The Console.ReadLine method reads a line of text from the input and returns a string. You'll use it to let the user tell you how many cards to pick:

```
string? line = Console.ReadLine();
```

← We showed you this line of code earlier in the chapter.

Useful method #3: int.TryParse

Your CardPicker.PickSomeCards method takes an int parameter. The line of input you get from the user is a string, so you'll need a way to convert it to an int. You'll use the int.TryParse method for that:

```
if (int.TryParse(line, out int numberOfCards))
{
  // this block is executed if line COULD be converted to an int
  // value that's stored in a new variable called numberOfCards
}
else
{
   // this block is executed if line COULD NOT be converted to an int
}
```

← We also showed you this line of code earlier in the chapter.

Put it all together

Your job is to take these three new pieces and put them together in a brand-new Main method for your console app. Modify your *Program.cs* file and replace the "Hello World!" line in the Main method with code that does this:

- Use Console.Write to ask the user for the number of cards to pick.
- Use Console.ReadLine to read a line of input into a string variable called `line`.
- Use int.TryParse to try to convert it to an int variable called `numberOfCards`.
- If the user input ***could be converted*** to an int value, use your CardPicker class to pick the number of cards that the user specified: CardPicker.PickSomeCards(numberOfCards). Use a string[] variable to save the results, then use a `foreach` loop to call Console.WriteLine on each card in the array. Flip back to Chapter 1 to see an example of a `foreach` loop—you'll use it to loop through every element of the array. Here's the first line of the loop: foreach (string card in CardPicker.PickSomeCards(numberOfCards))
- If the user input ***could not be converted***, use Console.WriteLine to write a message to the user indicating that the number was not valid.

Exercise Solution

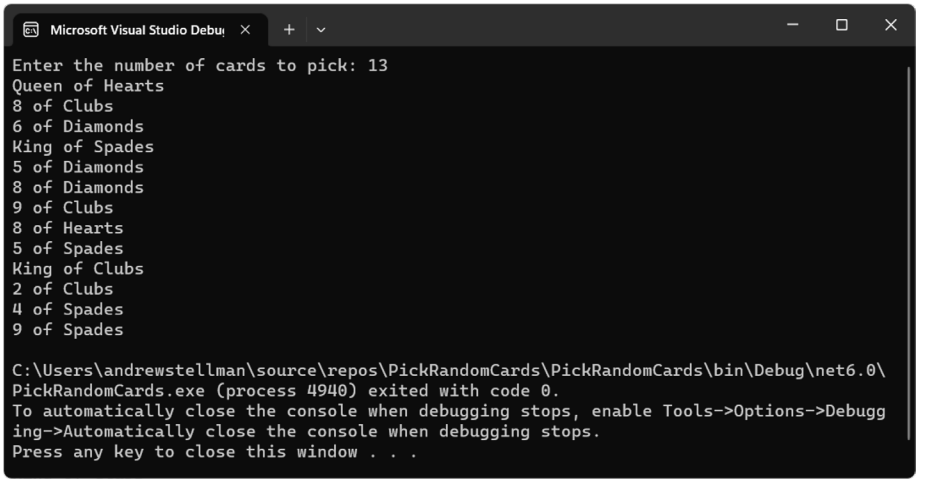
Here's the Main method for your console app. It replaces the one that Visual Studio created for you that prints "Hello, World!" This method prompts the user for the number of cards to pick, attempts to convert it to an int, and then uses the PickSomeCards method in the CardPicker class to pick that number of cards. PickSomeCards returns each of the picked cards in an array of strings, so it uses a `foreach` loop to write each of them to the console.

```
static void Main(string[] args)
{
    Console.Write("Enter the number of cards to pick: ");
    string? line = Console.ReadLine();
    if (int.TryParse(line, out int numberOfCards))
    {
        string[] cards = CardPicker.PickSomeCards(numberOfCards);
        foreach (string card in cards)
        {
            Console.WriteLine(card);
        }
    }
    else
    {
        Console.WriteLine("Please enter a valid number.");
    }
}
```

We gave you these lines of code.

This is just like the code we showed you earlier, except instead of passing a number like 5 to the method, you're passing it the numberOfCards variable.

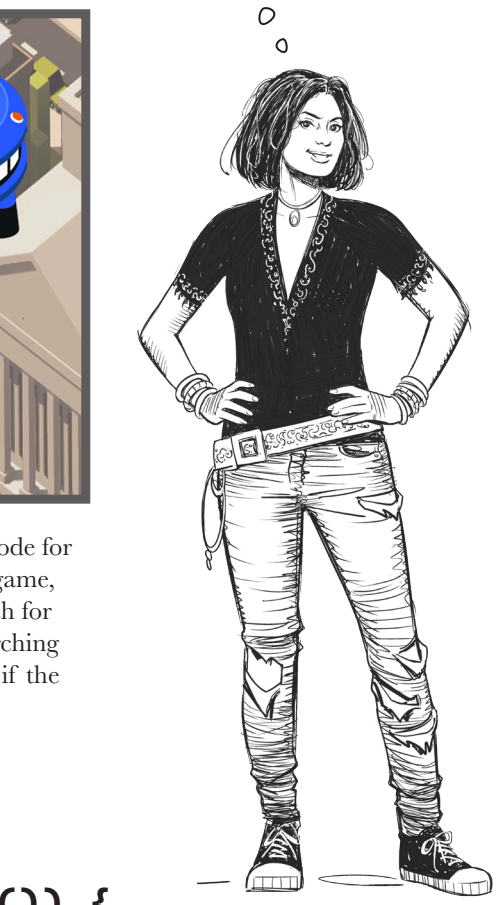
Here's what it looks like when you run your console app:



Take the time to really understand how this program works—this is a great opportunity to use the Visual Studio debugger to help you explore your code. Place a breakpoint on the first line of the Main method, then use Step Into to step through the entire program. Add a watch for the `value` variable, and keep your eye on it as you step through the RandomSuit and RandomValue methods.

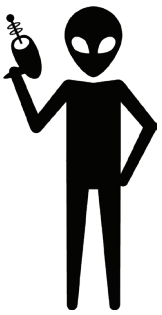
Ana's working on her next game

Meet Ana. She's an indie game developer. Her last game sold thousands of copies, and now she's getting started on her next one.

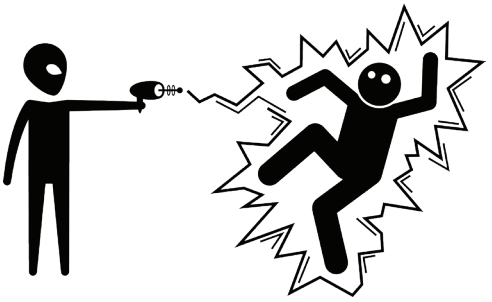


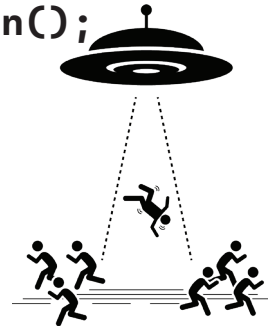
Ana's started working on some **prototypes**. She's been working on the code for the alien enemies that the player has to avoid in one exciting part of the game, where the player needs to escape from their hideout while the aliens search for them. Ana's written several methods that define the enemy behavior: searching the last location the player was spotted, giving up the search after a while if the player wasn't found, and capturing the player if the enemy gets too close.

```
SearchForPlayer();
```



```
if (SpottedPlayer()) {
    CommunicatePlayerLocation();
}
```

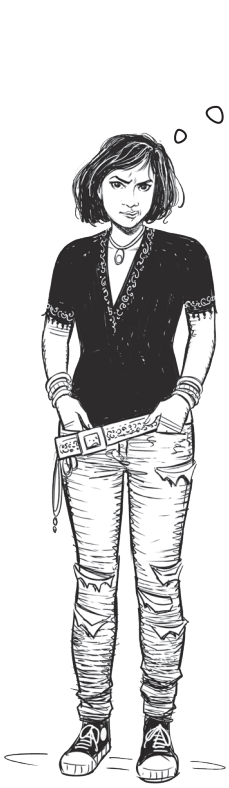




```
CapturePlayer();
```

Ana's game is evolving...

The humans versus aliens idea is pretty good, but Ana's not 100% sure that's the direction she wants to go in. She's also thinking about a nautical game where the player has to evade pirates. Or maybe it's a zombie survival game set on a creepy farm. In all three of those ideas, she thinks the enemies will have different graphics, but their behavior can be driven by the same methods.



...so how can Ana make things easier for herself?

Ana's not sure which direction the game should go in, so she wants to make a few different prototypes—and she wants them all to have the same code for the enemies, with the SearchForPlayer, StopSearching, SpottedPlayer, CommunicatePlayerLocation, and CapturePlayer methods. She's got her work cut out for her.

Brain Power

Can you think of a good way for Ana to use the same methods for enemies in different prototypes?

I PUT ALL OF THE ENEMY BEHAVIOR METHODS INTO A SINGLE ***ENEMY*** CLASS. CAN I ***REUSE THE CLASS*** IN EACH OF MY THREE DIFFERENT GAME PROTOTYPES?

Enemy
SearchForPlayer
SpottedPlayer
CommunicatePlayerLocation
StopSearching
CapturePlayer

Game design... and beyond

Prototypes

A **prototype** is an early version of your game that you can play, test, learn from, and improve. A prototype can be a really valuable tool to help you make changes early. Prototypes are especially useful because they let you rapidly experiment with a lot of different ideas before you've made permanent decisions.

- The first prototype is often a **paper prototype**, where you lay out the core elements of the game on paper. For example, you can learn a lot about your game by using sticky notes or index cards for the different elements of the game, and drawing out levels or play areas on large pieces of paper to move them around.
- One good thing about building prototypes is that they help you **get from an idea to a working, playable game** very quickly. You learn the most about a game (or any kind of program) when you get working software into the hands of your players (or users).
- Most games will go through **many prototypes**. This is your chance to try out lots of different things and learn from them. If something doesn't go well, think of it as an experiment, not a mistake.
- Prototyping is a **skill**, and just like any other skill, ***you get better at it with practice***. Luckily, building prototypes is also fun, and a great way to get better at writing C# code.

Prototypes aren't just used for games! When you need to build any kind of program, it's often a great idea to build a prototype first to experiment with different ideas.

Build a paper prototype for a classic game

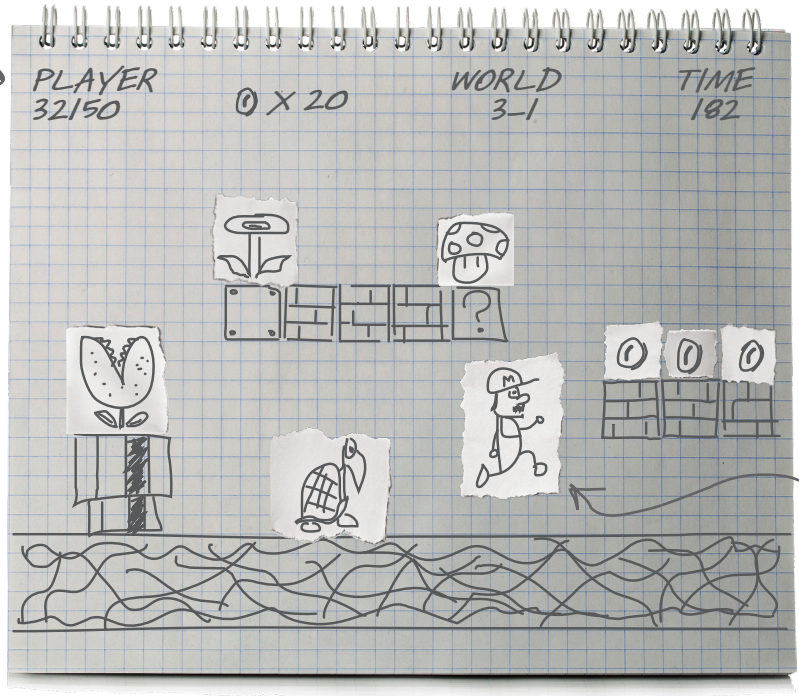
Paper prototypes are really useful for helping you figure out how a game will work before you start building it, which can save you a lot of time. There's a fast way to get started building them—all you need is some paper and a pen or pencil. Start by choosing your favorite classic game. Platform games work especially well, so we chose one of the **most popular, most recognizable** classic video games ever made... but you can choose any game you'd like! Here's what to do next.

1. **Draw the background on a piece of paper.** Start your prototype by creating the background. In our prototype, the ground, bricks, and pipe don't move, so we drew them on the paper. We also added the score, time, and other text at the top.

2. **Tear small scraps of paper and draw the moving parts.** In our prototype, we drew the characters, the piranha plant, the mushroom, the fire flower, and the coins on separate scraps. If you're not an artist, that's absolutely fine! Just draw stick figures and rough shapes. Nobody else ever has to see this!

3. **"Play" the game.** This is the fun part! Try to simulate player movement. Drag the player around the page. Make the non-player characters move too. It helps to spend a few minutes playing the game, then go back to your prototype and see if you can really reproduce the motion as closely as possible. (It will feel a little weird at first, but that's OK!)

The text at the top of the screen is called the HUD, or head-up display. It's usually drawn on the background in a paper prototype.



When the player catches a mushroom he grows to double his size, so we also drew a small character on a separate scrap of paper.

The ground, bricks, and pipe don't move, so we drew them on the background paper. There's no rule about what goes on the background and what moves around.

The mechanics of how the player jumps were really carefully designed. Simulating them in a paper prototype is a valuable learning exercise.

PAPER PROTOTYPES LOOK LIKE THEY'D BE USEFUL FOR **MORE THAN JUST GAMES**. I BET I CAN USE THEM IN MY OTHER PROJECTS, TOO.

All of the tools and ideas in "Game design... and beyond" sections are important skills that go way beyond just game development—but we've found that they're easier to learn when you try them with games first.

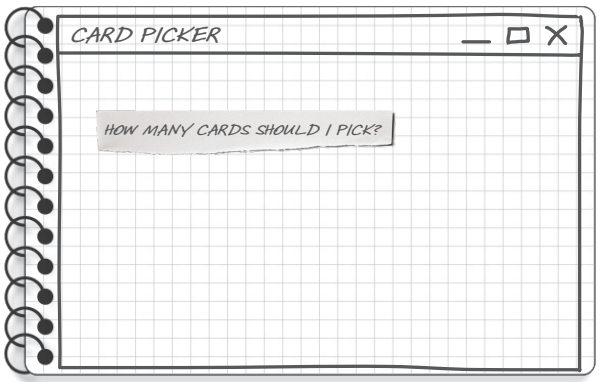
Yes! A paper prototype is a great first step for any project.

If you're building a desktop app, a mobile app, or any other project that has a user interface, building a paper prototype is a great way to get started. Sometimes you need to create a few paper prototypes before you get the hang of it. That's why we started with a paper prototype for a classic game…because that's a great way to learn how to build paper prototypes. **Prototyping is a really valuable skill for any kind of developer**, not just a game developer.

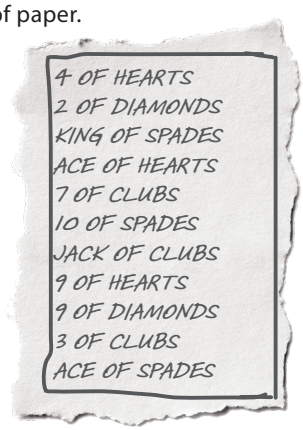
Sharpen your pencil

In the next project, you'll create a MAUI app that uses your CardPicker class to generate a set of random cards. In this paper-and-pencil exercise, you'll build a paper prototype of your app to try out various design options.

Start by drawing the window frame on a large piece of paper and a label on a smaller scrap of paper.



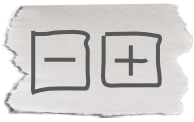
Your app needs to include a Button control with the text "Pick some cards" and a Label control to display the cards somewhere in the window.

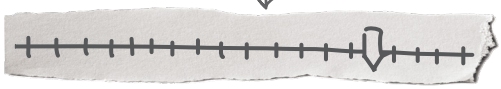


Next, draw a bunch of different types of controls on more small scraps of paper. Drag them around the window and experiment with ways to fit them together. What design do you think works best? There's no single right answer—there are lots of ways to design any app.

Your app needs a way for the user to choose the number of cards to pick. Try drawing an Entry control that they can use to type numbers into your app.

Try drawing Slider and Stepper controls, too. Can you think of other controls that you've used to input numbers into apps before? Maybe a Picker? Get creative!

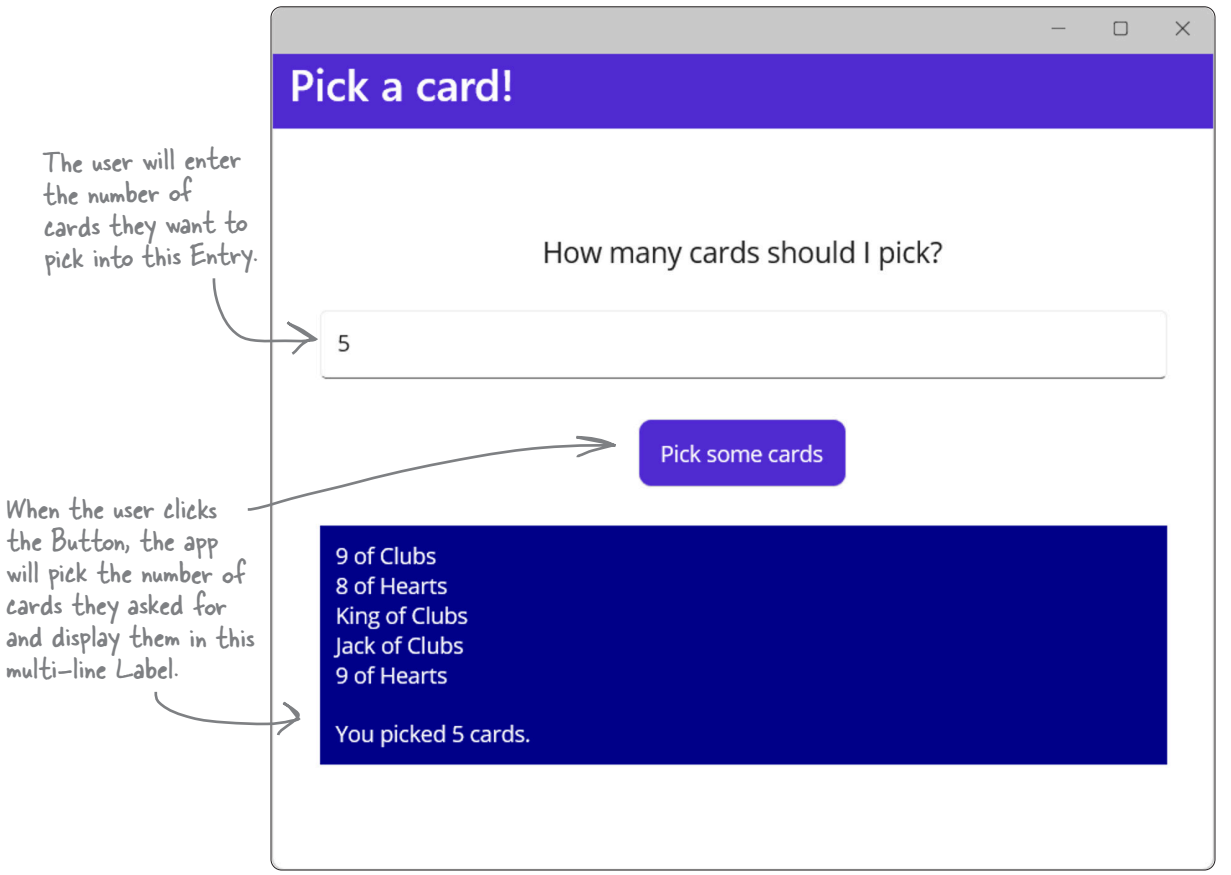




Build a MAUI version of your random card app

All of the code for picking random cards is conveniently organized into a class called CardPicker. Now you'll **reuse that class** in a .NET MAUI app.

Here's how the app will work.



Make your app accessible!

Accessibility is really important—and paying attention to accessibility is a great way to focus on important skills, like understanding your users and their needs.

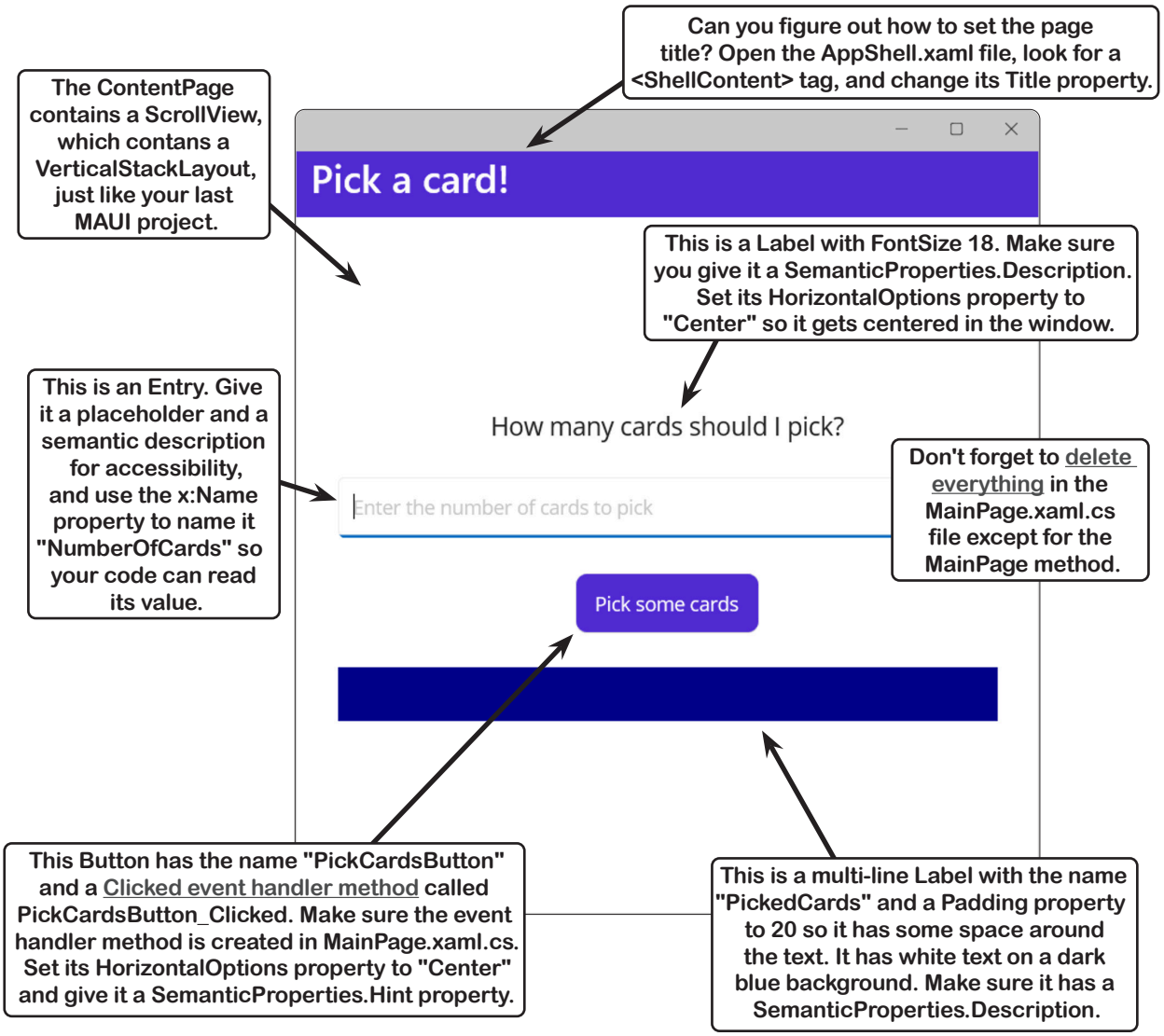
- The Label and Entry controls each have a **SemanticProperties.Description property** so the screen reader will read it out loud.
- The Button control has a **SemanticProperties.Hint property** because the screen reader will read the contents of the button but we still want to give people who use accessibilty tools additional context for the control.

Exercise

You already have the tools you need to create the XAML for the MAUI card picker app! In this exercise, you'll use what you learned about XAML in the first two chapters to create the main page for your app. You may need to go back to the XAML code you wrote in Chapter 2 to see how you added controls to your page.

Create a new .NET MAUI app called **PickRandomCardsMAUI**. Edit the MainPage.xaml.cs file to delete the controls inside the VerticalStackLayout (just like you did in Chapter 2), then add the controls for your card picker app.

Bonus: Edit the AppShell.xaml file to set the page title! We haven't showed you how to do that yet—can you figure it out?



Peeking at the solution is not cheating! It's actually a great way to get these ideas to stick in your brain.

Exercise Solution

Your MainPage.xaml.cs file should have a `public MainPage()` method that calls InitializeComponent and an empty Clicked event handler method and nothing else.

Here's the XAML for the contents of MainPage.xaml (we didn't include the outer <ContentPage> tag):

```
<ScrollView>
    <VerticalStackLayout
        Spacing="25"
        Padding="30,0"
        VerticalOptions="Center">

        <Label
            Text="How many cards should I pick?"
            SemanticProperties.Description="How many cards should I pick?"
            FontSize="18"
            HorizontalOptions="Center" />

        <Entry
            x:Name="NumberOfCards"
            SemanticProperties.Description="Enter the number of cards to pick"
            Placeholder="Enter the number of cards to pick" />

        <Button
            x:Name="PickCardsButton"
            Text="Pick some cards"
            SemanticProperties.Hint="Picks some cards"
            Clicked="PickCardsButton_Clicked"
            HorizontalOptions="Center" />

        <Label x:Name="PickedCards" Padding="20"
               TextColor="White" BackgroundColor="DarkBlue"
               SemanticProperties.Description="Shows the cards that were picked" />

    </VerticalStackLayout>
</ScrollView>
```

These are the same ScrollView and VerticalStackLayout tags that Visual Studio created using the .NET MAUI template.

The HorizontalOptions property centers the label on the page. Try the other options—do you like the way they look better?

You gave the Entry, Button, and Label controls names that you'll use in your C# code.

Make sure Visual Studio added the PickCardsButton_Clicked event handler method that gets called when the button is clicked. You'll use it in the second part of this project.

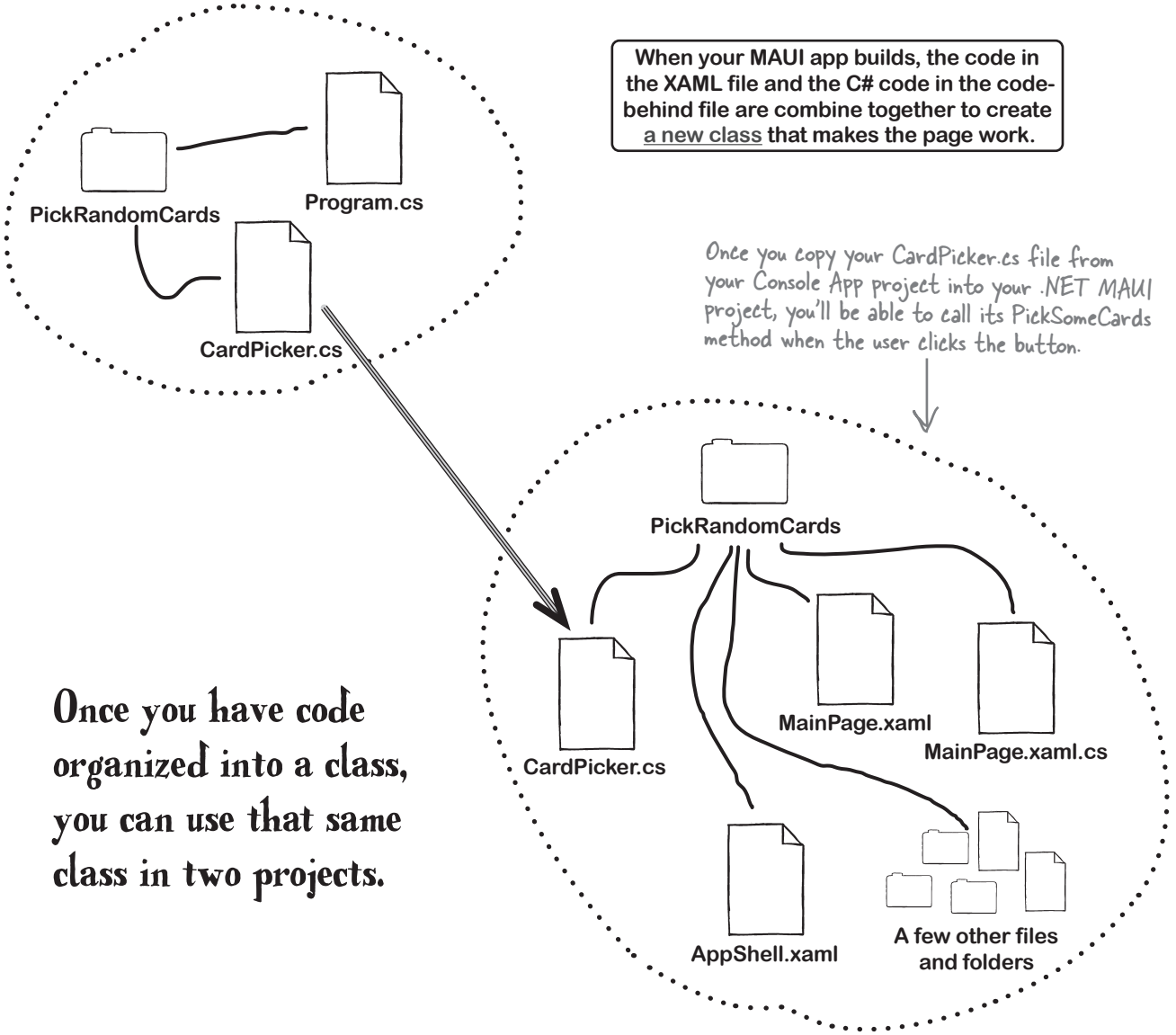
We made this change to AppShell.xaml to set the title of the page to "Pick a card!":

```
<?xml version="1.0" encoding="UTF-8" ?>
<Shell
    ...
    Shell.FlyoutBehavior="Disabled" BackgroundColor="Red">

    <ShellContent
        Title="Pick a card!"
        ContentTemplate="{DataTemplate local:MainPage}"
        Route="MainPage" />

</Shell>
```

The XAML in AppShell.xml tells your MAUI app what to do when it first starts up. The ShellContent's Route property tells it to load the page in your MainPage.xaml file. Try setting the BackgroundColor of the outer Shell tag—what does that change in the app?

Make your MAUI app pick random cards

You've got an app that looks like it's supposed to, and that's a great start! In the second part of this project, you'll make it work, so when the user enters a number and clicks the button it picks random cards. That's where your CardPicker class comes in. You've already created a class that picks random cards. Now you just need to **copy that class into your new APP**. Once it's copied, you'll be able to make your button's event handler method call the PickSomeCards method in the CardPicker class.



Once you have code organized into a class, you can use that same class in two projects.

Reuse your the CardPicker class

You took the time to put all of the random card picking code into a convenient class. Now it's time to take advantage of that class by **reusing it in your new MAUI app**.

1 Choose 'Add Existing Item' or 'Add Existing Files' in Visual Studio.

Do this!

You created a file called CardPicker.cs in your PickRandomCards console app. Now you'll tell Visual Studio to **add that class file** to your MAUI project, which will cause it to copy the file into your MAUI app's project folder..

- On Windows, right-click on the project in the the Solution Explorer window and choose Add >> Existing Item... (Shift+Alt+A), or choose Add Existing Item from the Project menu.
- On macOS, right-click on the project in the Solution window and choose Add >> Existing Files... (⌥⌘A). Visual Studio will ask whether you want to copy, move, or add a link to the file. Make sure you copy the file so it exists in the MAUI project folder.

2 Find your CardPicker.cs file and add it to your project.

Visual Studio will pop up a folder explorer window. Navigate to the folder with your PickACard console app and **double-click on CardPicker.cs**. You should now see CardPicker in the Solution Explorer.

3 Try to use your CardPicker class in the MainPage.xaml.cs code.

Open MainPage.xaml.cs. Edit the PickCardsButton_Clicked event handler method and try add a statement that calls your CardPicker.PickSomeCards method.

```
public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();
    }

    private void PickCardsButton_Clicked(object sender, EventArgs e)
    {
        CardPicker.
    }
}
```

Here's the event handler method that Visual Studio added to your C# code when you added a Clicked event handler to the XAML for the button.

Hold on—something's wrong!

When you start typing the statement to call CardPicker.PickSomeCards, Visual Studio doesn't pop up its normal IntelliSense window, and there's a squiggly error line under CardPicker.

Why do you think Visual Studio is treating CardPicker like that?

Add a using directive to use code in another namespace

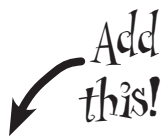
When you created your CardPicker class, it added a **namespace declaration** at the top that matches the name you chose for the the project, followed by a pair of curly brackets that contain the entire class:

```
namespace PickRandomCards
{
    ... your class is in the PickRandomCards namespace ...
}
```

Compare that to the code at the top of your MainPage.xaml.cs file in your MAUI app:

```
namespace PickRandomCardsMAUI;

public partial class MainPage : ContentPage
{
  ... your MAUI app's code is in the PickRandomCardsMAUI namespace ...
}
```

The reason your MainPage class can't access the methods in your CardPicker class is because they're in different namespaces. Luckily, C# has an easy way to deal with this. You'll add a **using directive** in your code that calls the methods in CardPicker—that's a special line that you put at the top of a class file to tell it to use code in another namespace.



Add this line to the top of your MainPage.xaml.file. If you chose a different name for your console app, replace PickRandomCards with then namespace in your CardPicker.cs file.

```
using PickRandomCards;
```

This using directive will let you add code to your MainPage.xaml.cs file that uses classes in the PickRandomCards namespace—so now you can can write code that calls methods in your CardPicker class. You might see other using directives at the top of the file, too.

Now go back to the event handler method for your button. Start typing `CardPicker.` like you did before. Now Visual Studio will pop up its IntelliSense window, just like you'd expect it to.

Exercise

Here's a C# coding challenge for you! Now that you added the using directive to the top of your MainPage.xaml.cs file, you can call the your CardPicker class. Can you finish the event handler method to make your app work?

- The first thing the method does is call int.TryParse to convert NumberOfCards.Text to a number.
- If the number is valid, it calls CardPicker.PickSomeCards just like in your console app. If it isn't, it makes the PickedCards label display a message: `PickedCards.Text = "Please enter a valid number.";`
- Instead of writing to the console, it sets PickedCards.Text to a string value to make text appear in the PickedCards Label control. You can clear the text in PickedCards like this: `PickedCards.Text = String.Empty;`
- After it clears the PickedCards label, it uses a foreach loop that works just like the one in your console app.
- Add this statement after the foreach loop to tell the user how many cards they picked: `PickedCards.Text += Environment.NewLine + "You picked " + numberOfCards + " cards.";`

Exercise Solution

Heres the finished event handler method.

```
private void PickCardsButton_Clicked(object sender, EventArgs e)
{
  if (int.TryParse(NumberOfCards.Text, out int numberOfCards))
  {
    string[] cards = CardPicker.PickSomeCards(numberOfCards);
    PickedCards.Text = String.Empty;
    foreach (string card in cards)
    {
      PickedCards.Text += card + Environment.NewLine;
    }
    PickedCards.Text += Environment.NewLine + "You picked " + numberOfCards + " cards.";
  }
  else
  {
    PickedCards.Text = "Please enter a valid number.";
  }
}
```

Now that you have a using directive at the top of your MainPage.xaml.cs file, you can use the CardPicker class.

The foreach loop works just like the one in the console app, except instead of writing a line of text to the console it adds a line to the multi-line PickedCards Label control.

What happens if you don't add this last line to your PickedCards Label? Does it look weird? Can you sleuth out how to fix it?

Bullet Points

- Classes have methods that contain statements that perform actions. Well-designed classes have sensible method names.
- Some methods have a **return type**. You set a method's return type in its declaration. A method with a declaration that starts with the `int` keyword returns an int value. Here's a statement that returns an int value: `return 37;`
- When a method has a return type, it **must** have a `return` statement that returns a value that matches a return type. So if a method declaration has the string return type then you need a `return` statement that returns a string.
- As soon as a `return` statement in a method executes, your program jumps back to the statement that called the method.
- Not all methods have a return type. A method with a declaration that starts `public void` doesn't return anything at all. You can still use a `return` statement to exit a void method, as in this example: `if (finishedEarly) { return; }`
- Developers often **reuse** the same code in multiple programs. Classes can help you make your code more reusable.
- When you **select a control** in the XAML code editor, you can edit its properties in the Properties window.
- The XAML code combines with the C# code in the code-behind file to **create a new class**.

Ana's prototypes look great...

Ana found out that whether her player was being chased by an alien, a pirate, a zombie, or an evil killer clown, she could use the same methods from her Enemy class to make them work. Her game is starting to shape up.

Enemy
SearchForPlayer
SpottedPlayer
CommunicatePlayerLocation
StopSearching
CapturePlayer

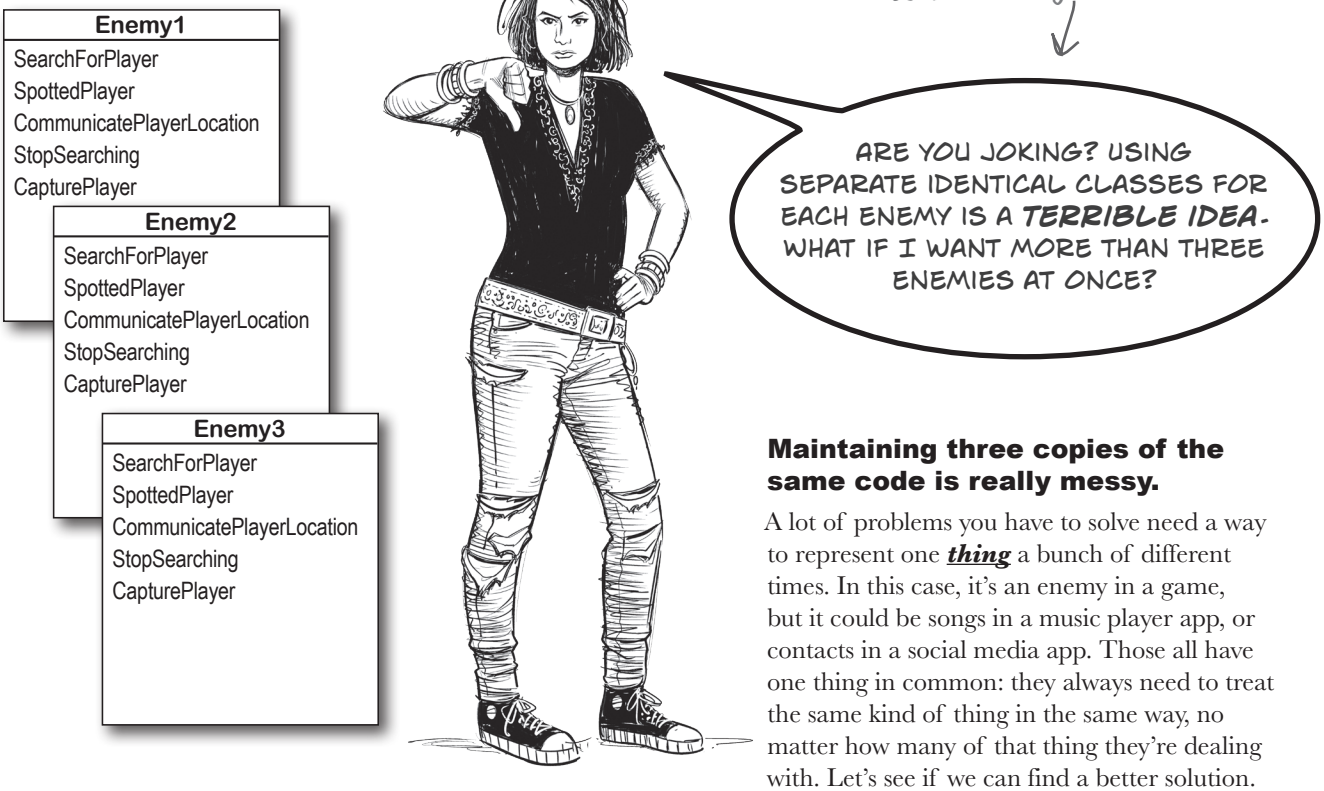
...but what if she wants more than one enemy?

And that's great…until Ana wants more than one enemy, which is all there was in each of her early prototypes. What should she do to add a second or third enemy to her game?

Ana *could* copy the Enemy class code and paste it into two more class files. Then her program could use methods to control three different enemies at once. Technically, we're reusing the code…right?

Hey Ana, what do you think of that idea?

She has a point. What if she wants a level with, say, dozens of zombies? Creating dozens of identical classes just isn't practical.

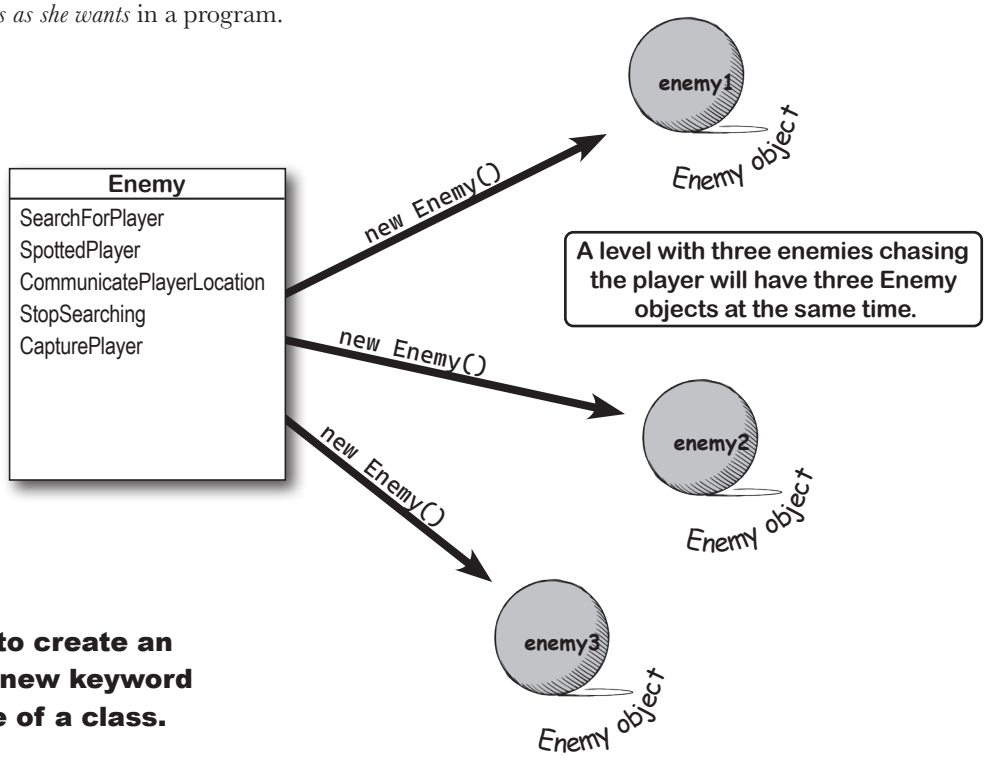


Maintaining three copies of the same code is really messy.

A lot of problems you have to solve need a way to represent one ***thing*** a bunch of different times. In this case, it's an enemy in a game, but it could be songs in a music player app, or contacts in a social media app. Those all have one thing in common: they always need to treat the same kind of thing in the same way, no matter how many of that thing they're dealing with. Let's see if we can find a better solution.

Ana can use objects to solve her problem

Objects are C#'s tool that you use to work with a bunch of similar things. Ana can use objects to program her Enemy class just once, and use it *as many times as she wants* in a program.



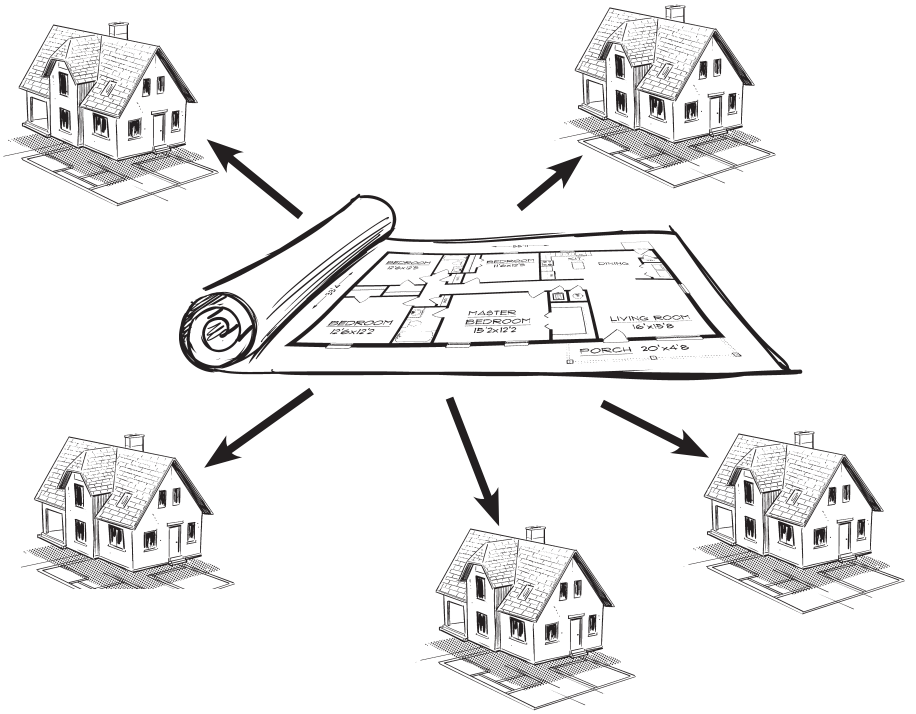
All you need to create an object is the new keyword and the name of a class.

```
Enemy enemy1 = new Enemy();
enemy1.SearchForPlayer();
if (enemy1.SpottedPlayer()) {
   enemy1.CommunicatePlayerLocation();
} else {
   enemy1.StopSearching();
}
```

Now you can use the object! When you create an object from a class, that object has all of the methods from that class.

You use a class to build an object

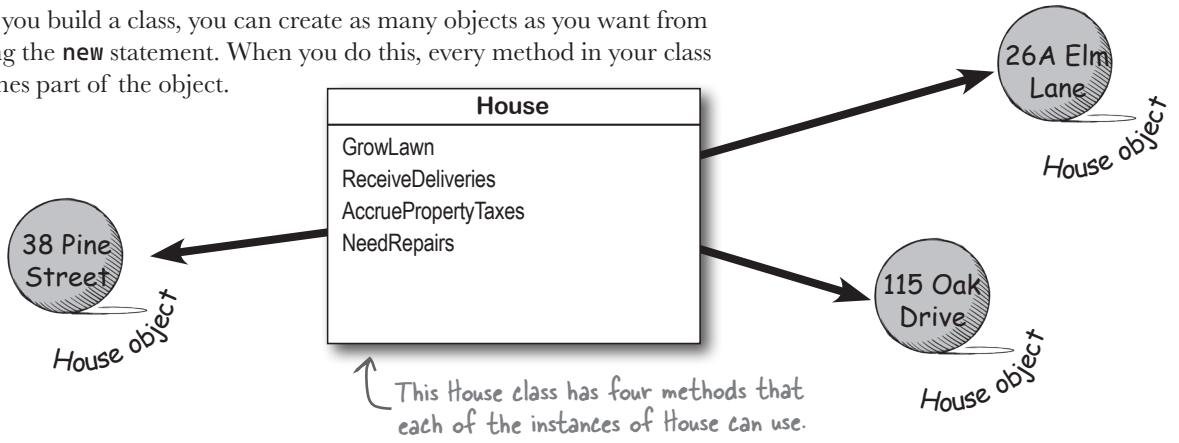
A class is like a blueprint for an object. If you wanted to build five identical houses in a suburban housing development, you wouldn't ask an architect to draw up five identical sets of blueprints. You'd just use one blueprint to build five houses.



A class defines its members, just like a blueprint defines the layout of the house. You can use one blueprint to make any number of houses, and you can use one class to make any number of objects.

An object gets its methods from its class

Once you build a class, you can create as many objects as you want from it using the `new` statement. When you do this, every method in your class becomes part of the object.



When you create a new object from a class, it's called an instance of that class

You use the **new keyword** to create an object. All you need is a variable to use with it. Use the class as the variable type to declare the variable, so instead of int or bool, you'll use a class like House or Enemy.

> in-stance, noun.
> an example or one occurrence of something. *The IDE search-and-replace feature finds every* ***instance*** *of a word and changes it to another.*

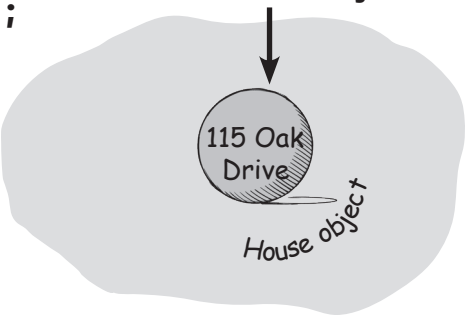
Before: here's a picture of your computer's memory when your program starts.

Your program executes a `new` statement.

```
House mapleDrive115 = new House();
```

This new statement creates a new House object and assigns it to a variable called mapleDrive115.

After: now it has an instance of the House class in memory.



Yes! You've already created instances in your own code.

Go back to your animal matching program and look for this line of code:

```
Random random = new Random();
```

You created an instance of the Random class, and then you called its Next method. Now look at your CardPicker class and find the **new** statement. You've been using objects this whole time!

A better solution for Ana...brought to you by objects

Ana used objects to reuse the code in the Enemy class without all that messy copying that would've left duplicate code all over her project. Here's how she did it.

1. Ana created a Level class that stored the enemies in an **Enemy array** called `enemyArray`, just like you used string arrays to store cards and animal emoji.

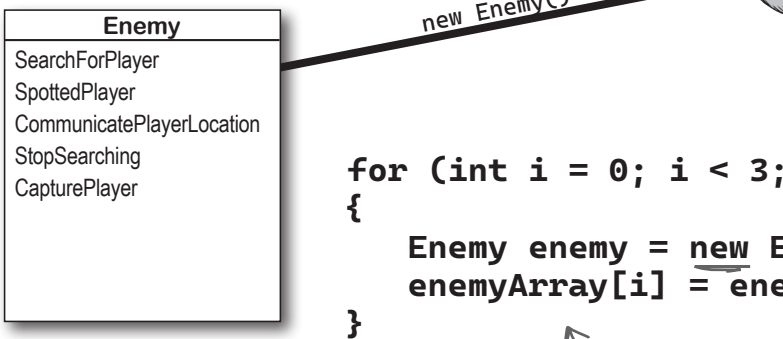
```
public class Level {
    Enemy[] enemyArray = new Enemy[3];
```

Hmm, this array is inside the class, but outside of the methods. What do you think is going on?

Use the name of a class to declare an array of instances of that class.

We're using the new keyword to create an array of Enemy objects, just like you did earlier with strings.

2. She used a loop that called `new` statements to create new instances of the Enemy class for the level and add them to an array of enemies.



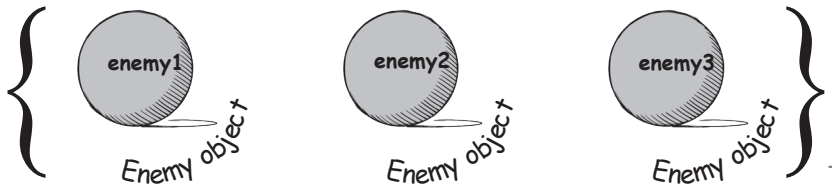
The enemy1 object is an instance of the Enemy class.

```
for (int i = 0; i < 3; i++)
{
    Enemy enemy = new Enemy();
    enemyArray[i] = enemy;
}
```

This statement uses the new keyword to create an Enemy object.

This statement adds the newly created Enemy object to the array.

3. She called methods of each Enemy instance during every frame update to implement the enemy behavior.



The foreach loop iterates through the array of Enemy objects.

```
foreach (Enemy enemy in enemyArray)
{
    // code that calls the Enemy methods
}
```

When you create a new instance of a class, it's called instantiating that class.

WAIT A MINUTE! YOU DIDN'T GIVE ME *NEARLY ENOUGH INFORMATION* TO BUILD ANA'S GAME.

That's right, we didn't.

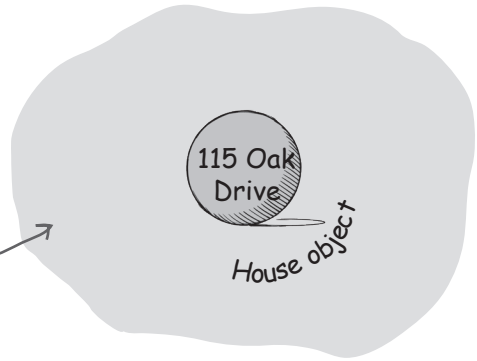
Some game prototypes are really simple, while others are much more complicated—but complicated programs follow the same patterns as simple ones. Ana's game program is an example of how someone would use objects in real life. And this doesn't just apply to game development! No matter what kind of program you're building, you'll use objects in exactly the same way that Ana did in her game. Ana's example is just the starting point for getting this concept into your brain. We'll give you **lots more examples** over the rest of the chapter—and this concept is so important that we'll revisit it in future chapters, too.

Theory and practice

Speaking of patterns, here's a pattern that you'll see over and over again throughout the book. We'll introduce a concept or idea (like objects) over the course of a few pages, using pictures and short code excerpts to demonstrate the idea. This is your opportunity to take a step back and try to understand what's going on without having to worry about getting a program to work.

```
House mapleDrive115 = new House();
```

When we're introducing a new concept (like objects), keep your eyes open for pictures and code excerpts like these.



Sharpen your pencil

Now that you've got a better idea of how objects work, it's a great time to go back to your CardPicker class and get to know the Random class that you're using.

1. Open any Console App project that uses top-level statements (or create a new one). Press Enter to start a new statement, then type **`Random.Shared.`**—as soon as you type the second period, Visual Studio will pop up an IntelliSense window that shows its methods. Each method is marked with a cube icon () on Windows or an M on a MAc. We filled in some of the methods. Finish filling in the class diagram for the Random class.

Random
Equals GetHashCode GetType .... ToString

In the last chapter we showed youthree types, int (for whole numbers), string (for text), and bool (for true/false values). A double is another type that's used for numbers with decimal places. The computer science term for a number with decimal places is a floating-point number. You'll learn about more types in the next chapter.

2. Write code to create a new array of Doubles called **`randomDoubles`**, then use a **`for`** loop to add 20 double values to that array. Use the IntelliSense pop-up to help you choose the right method from the Random class to use in your code—make sure you're calling the method that returns a random floating-point number that is greater than or equal to 0.0, and less than 1.0. (We'll talk about what "floating point" means in the next chapter.)

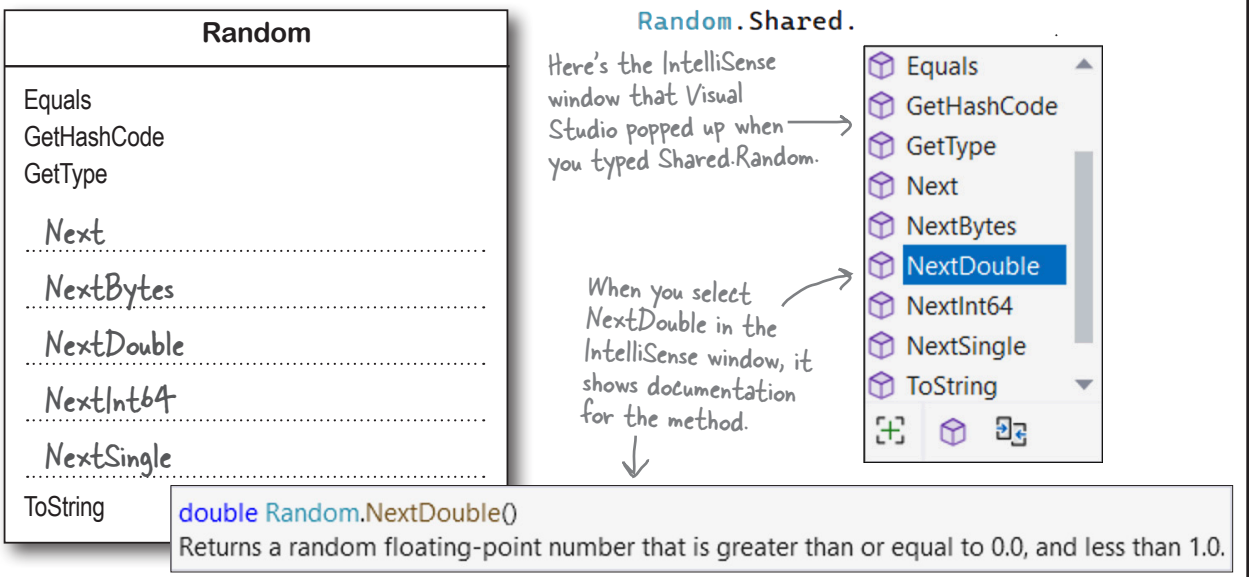
```
double[] randomDoubles = new double[20];

{
    double value =

}
```

We filled in part of the code, including the curly braces. Your job is to finish those statements and then write the rest of the code.

Sharpen your pencil Solution

Now that you've got a better idea of how objects work, it's a great time to go back to your CardPicker class and get to know the Random class that you're using.

1. Open any Console App project that uses top-level statements (or create a new one). Press Enter to start a new statement, then type **`Random.Shared.`**—as soon as you type the second period, Visual Studio will pop up an IntelliSense window that shows its methods. Each method is marked with a cube icon () on Windows or an M on a MAc. We filled in some of the methods. Finish filling in the class diagram for the Random class.



2. Write code to create a new array of Doubles called **`randomDoubles`**, then use a **`for`** loop to add 20 double values to that array. Use the IntelliSense pop-up to help you choose the right method from the Random class to use in your code—make sure you're calling the method that returns a random floating-point number that is greater than or equal to 0.0, and less than 1.0. (We'll talk about what "floating point" means in the next chapter.)

```
double[] randomDoubles = new double[20];
for (int i = 0; i < 20; i++)
{
    double value = Random.Shared.NextDouble();
    randomDoubles[i] = value;
}
```

This is really similar to the code that you used in your CardPicker class.

An instance uses fields to keep track of things

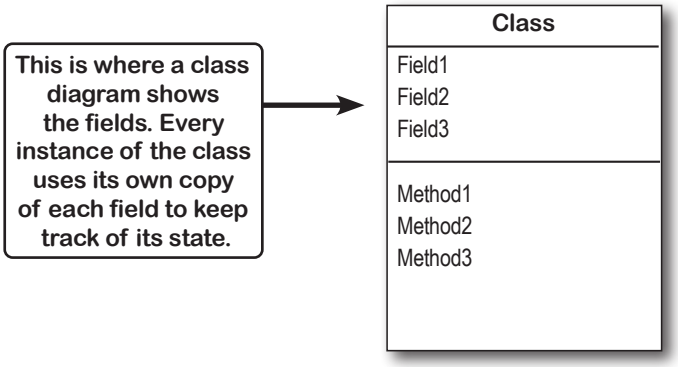
You've seen how classes can contain fields as well as methods. We just saw how you used the `static` keyword to declare a field in your CardPicker class:

```
static Random random = new Random();
```

What happens if you take away that `static` keyword? Then the field becomes an **instance field**, and every time you ***instantiate*** the class, the new instance that was created *gets its own copy* of that field.

Sometimes people think the word "instantiate" sounds a little weird, but it makes sense when you think about what it means: creating a new instance of a class.

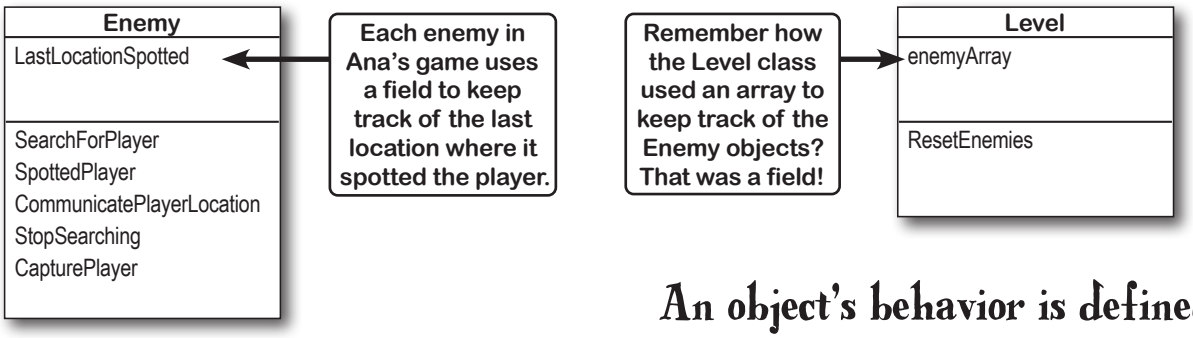
When we want to include fields a class diagram, we'll draw a horizontal line in the box. The fields go above the line, and methods go below the line.



Class diagrams typically list all of the fields and methods in the class. We call them the class members.

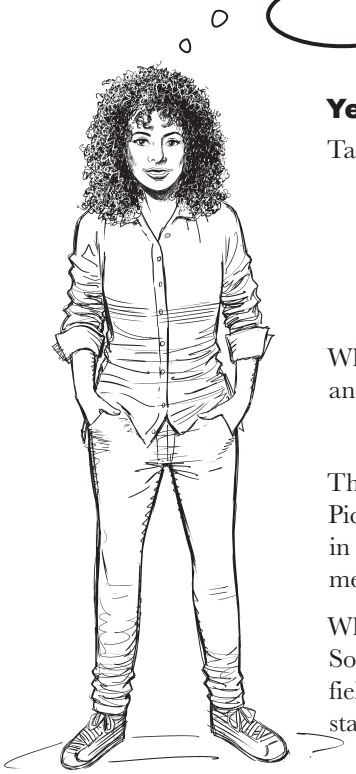
Methods are what an object does. Fields are what the object knows.

When Ana's prototype created three instances of her Enemy class, each of those objects was used to keep track of a different enemy in the game. Every instance keeps separate copies of the same data: setting a field on the enemy2 instance won't have any effect on the enemy1 or enemy3 instances.



An object's behavior is defined by its methods, and it uses fields to keep track of its state.

I USED THE **NEW** KEYWORD TO CREATE AN INSTANCE OF RANDOM, BUT I **NEVER CREATED A NEW INSTANCE** OF MY CARDPICKER CLASS. SO DOES THAT MEAN I CAN CALL METHODS WITHOUT CREATING OBJECTS?



Yes! That's why you used the `static` keyword in your declarations.

Take another look at the the method declarations in your CardPicker class:

```
public static string[] PickSomeCards(int numberOfCards)

private static string RandomValue()

private static string RandomSuit()
```

When you use the **`static`** keyword to declare a field or method in a class, you don't need an instance of that class to access it. You just called your method like this:

```
CardPicker.PickSomeCards(numberOfCards)
```

That's how you call static methods. If you take away the **`static`** keyword from the PickSomeCards method declaration, then you'll have to create an instance of CardPicker in order to call the method. Other than that distinction, static methods are just like object methods: they can take arguments, they can return values, and they live in classes.

When a field is static **there's only one copy of it, and it's shared by all instances**. So if you created multiple instances of CardPicker, they would all share the same *random* field. You can even mark your **whole class** as static, and then all of its members **must** be static too. If you try to add a nonstatic method to a static class, your program won't build.

there are no Dumb Questions

Q: When I think of something that's "static" I think of something that doesn't change. Does that mean nonstatic methods can change, but static methods don't? Do they behave differently?

A: No, both static and nonstatic methods act exactly the same. The only difference is that static methods don't require an instance, while nonstatic methods do.

Q: So I can't use my class until I create an instance of an object?

A: You can use its static methods, but if you have methods that aren't static, then you need an instance before you can use them.

Q: Then why would I want a method that needs an instance? Why wouldn't I make all my methods static?

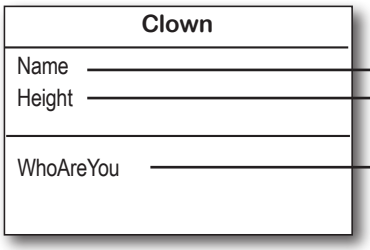
A: Because if you have an object that's keeping track of certain data—like Ana's instances of her Enemy class that each kept track of different enemies in her game—then you can use each instance's methods to work with that data. So when Ana's game calls the StopSearching method on the enemy2 instance, it only causes that one enemy to stop searching for the player. It doesn't affect the enemy1 or enemy3 objects, and they can keep searching. That's how Ana can create game prototypes with any number of enemies, and her programs can keep track of all of them at once.

When a field is static, there's only one copy of it shared by all instances.

Sharpen your pencil

Here's a console app that uses top-level statements and writes several lines to the console. It includes a class called Clown that has two fields, Name and Height, and a method called WhoAreYou that uses those fields to write a line to the console. Your job is to read the code and **write down the lines that are printed to the console**.

Here's the class diagram and code for the Clown class:




```
class Clown {
    public string Name;
    public int Height;

    public void WhoAreYou() {
        Console.WriteLine("My name is " + Name +
            " and I'm " + Height + " inches tall.");
    }
}
```

Here's are the contents of the Program.cs file. There are comments next to each of the calls to the WhoAreYou method, which prints a line to the console. Your job is to fill in the blanks in the comments so they match the output.

```
Clown oneClown = new Clown();
oneClown.Name = "Boffo";
oneClown.Height = 14;
oneClown.WhoAreYou();      // My name is _______ and I'm ____ inches tall.

Clown anotherClown = new Clown();
anotherClown.Name = "Biff";
anotherClown.Height = 16;
anotherClown.WhoAreYou();  // My name is _______ and I'm ____ inches tall.

Clown clown3 = new Clown();
clown3.Name = anotherClown.Name;
clown3.Height = oneClown.Height - 3;
clown3.WhoAreYou();        // My name is _______ and I'm ____ inches tall.

anotherClown.Height *= 2;
anotherClown.WhoAreYou();  // My name is _______ and I'm ____ inches tall.
```

The *= operator tells C# to take whatever's on the left of the operator and multiply it by whatever's on the right, so this will update the Height field.

Thanks for the memory

When your program creates an object, it lives in a part of the computer's memory called the **heap**. When your code creates an object with a `new` statement, C# immediately reserves space in the heap so it can store the data for that object.

Here's a picture of the heap before the project starts. Notice that it's empty.

When your program creates a new object, it gets added to the heap.

Sharpen your pencil Solution

Here's what the program prints to the console. It's worth taking a few minutes to create a new console app—make sure it uses top-level statements—add the Clown class, and make its Program.cs method the code below, Then step through it with the debugger so you can see exactly how it works.

```
Clown oneClown = new Clown();
oneClown.Name = "Boffo";
oneClown.Height = 14;
oneClown.WhoAreYou();      // My name is Boffo and I'm 14 inches tall.

Clown anotherClown = new Clown();
anotherClown.Name = "Biff";
anotherClown.Height = 16;
anotherClown.WhoAreYou();  // My name is Biff and I'm 16 inches tall.

Clown clown3 = new Clown();
clown3.Name = anotherClown.Name;
clown3.Height = oneClown.Height - 3;
clown3.WhoAreYou();        // My name is Biff and I'm 11 inches tall.

anotherClown.Height *= 2;
anotherClown.WhoAreYou();  // My name is Biff and I'm 32 inches tall.
```

What's on your app's mind

Let's take a closer look at the program in the "Sharpen your pencil" exercise, starting with the first line of the app. It's actually **two statements** combined into one:

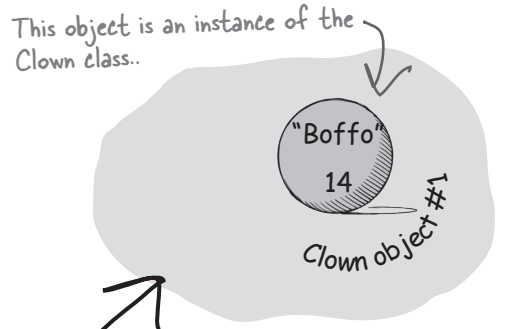
```
Clown oneClown = new Clown();
```

This is a statement that declares a variable called oneClown of type Clown.

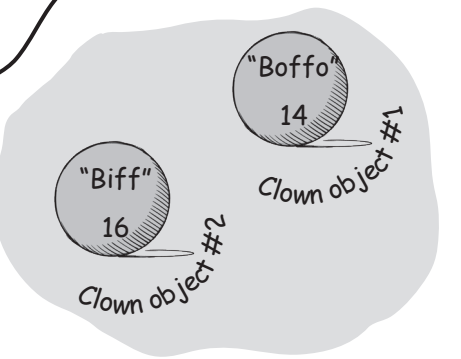
This statement creates a new object and assigns it to the oneClown variable.

Next, let's look closely at what the heap looks like after each group of statements is executed:

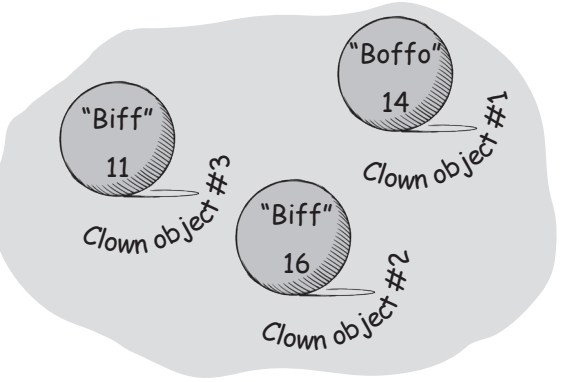
```
// These statements create an instance of
// Clown and then set its fields
Clown oneClown = new Clown();
oneClown.Name = "Boffo";
oneClown.Height = 14;
oneClown.WhoAreYou();
```



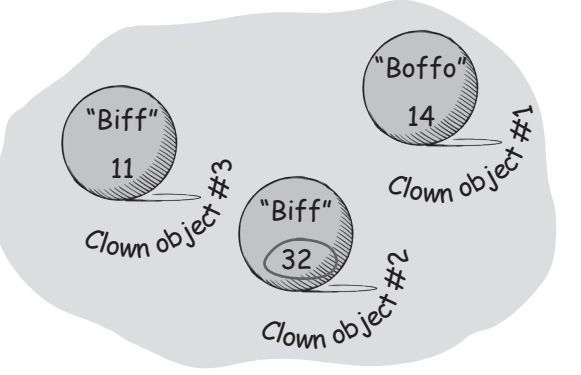

```
// These statements instantiate a second
// Clown object and fill it with data.
Clown anotherClown = new Clown();
anotherClown.Name = "Biff";
anotherClown.Height = 16;
anotherClown.WhoAreYou();
```




```
// Now we instantiate a third Clown object
// and use data from the other two
// instances to set its fields
Clown clown3 = new Clown();
clown3.Name = anotherClown.Name;
clown3.Height = oneClown.Height - 3;
clown3.WhoAreYou();
```




```
// Notice how there's no "new" statement
// here -- we're not creating a new object,
// just modifying one already in memory
anotherClown.Height *= 2;
anotherClown.WhoAreYou();
```



Sometimes code can be difficult to read

You may not realize it, but you're constantly making choices about how to structure your code. Do you use one method to do something? Do you split it into more than one? Do you even need a new method at all? The choices you make about methods can make your code much more intuitive—or if you're not careful, much more convoluted.

Here's a nice, compact chunk of code from a control program that runs a machine that makes candy bars:

```
int t = m.chkTemp();
if (t > 160) {
   T tb = new T();
   tb.clsTrpV(2);
   ics.Fill();
   ics.Vent();
   m.airsyschk();
}
```

Extremely compact code can be especially problematic

Take a second and look at that code. Can you figure out what it does? Don't feel bad if you can't—it's very difficult to read! Here are a few reasons why:

- We can see a few variable names: `tb`, `ics`, `m`. These are terrible names! We have no idea what they do. And what's that T class for?
- The chkTemp method returns an integer…but what does it do? We can guess maybe it has something to do with checking the temperature of…something?
- The clsTrpV method has one parameter. Do we know what that parameter is supposed to be? Why is it 2? What is that 160 number for?

C# and .NET are everywhere...and we mean everywhere.
Have you ever played with a Raspberry PI? It's a low-cost computer on a single board, and computers like it can be found inside all sorts of machinery. Thanks to Windows IoT (or Internet of Things), your C# code can run on them.

You can learn more about .NET IoT apps here: https://dotnet.microsoft.com/apps/iot

Micorosft even has a free Raspberry PI simulator that you can use to get started: https://azure-samples.github.io/raspberry-pi-web-simulator/#GetStarted

Most code doesn't come with a manual

Those statements don't give you any hints about why the code's doing what it's doing. In this case, the programmer was happy with the results because she was able to get it all into one method. But making your code as compact as possible isn't really useful! Let's break it up into methods to make it easier to read, and make sure the classes are given names that make sense.

We'll start by figuring out what the code is supposed to do. Luckily, we happen to know that this code is part of an **embedded system**, or a controller that's part of a larger electrical or mechanical system. And we happen to have documentation for this code—specifically, the manual that the programmers used when they originally built the system.

General Electronics Type 5 Candy Bar Maker Manual

The nougat temperature must be checked every 3 minutes by an automated system. If the temperature **exceeds 160°C**, the candy is too hot, and the system must **perform the candy isolation cooling system (CICS) vent procedure**:

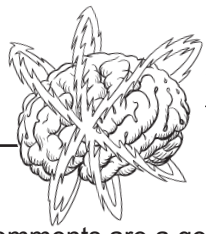
- Close the trip throttle valve on turbine #2.
- Fill the isolation cooling system with a solid stream of water.
- Vent the water.
- Initiate the automated check for air in the system.

How do you figure out what your code is supposed to do? Well, all code is written for a reason. So it's up to you to figure out that reason! In this case, we got lucky—we could look up the page in the manual that the developer followed.

We can compare the code with the manual that tells us what the code is supposed to do. Adding comments can definitely help us understand what it's supposed to do:

```
/* This code runs every 3 minutes to check the temperature.
 * If it exceeds 160C we need to vent the cooling system.
 */
int t = m.chkTemp();
if (t > 160) {
   // Get the controller system for the turbines
   T tb = new T();

   // Close throttle valve on turbine #2
   tb.clsTrpV(2);

   // Fill and vent the isolation cooling system
   ics.Fill();
   ics.Vent();

   // Initiate the air system check
   m.airsyschk();
}
```

Adding extra line breaks to your code in some places can make it easier to read.



Brain Power

Code comments are a good start. Can you think of a way to make this code even easier to understand?

Use intuitive class and method names

That page from the manual made it a lot easier to understand the code. It also gave us some great hints about how to make our code easier to understand. Let's take a look at the first two lines:

```
/* This code runs every 3 minutes to check the temperature.
 * If it exceeds 160C we need to vent the cooling system.
 */
int t = m.chkTemp();
if (t > 160) {
```

The comment we added explains a lot. Now we know why the conditional test checks the variable **t** against 160—the manual says that any temperature above 160°C means the nougat is too hot. It turns out that **m** is a class that controls the candy maker, with static methods to check the nougat temperature and check the air system.

So let's put the temperature check into a method, and choose names for the class and the methods that make their purpose obvious. We'll move these first two lines into their own method that returns a Boolean value, true if the nougat is too hot or false if it's OK:

```
/// <summary>
/// If the nougat temperature exceeds 160C it's too hot.
/// </summary>
public bool IsNougatTooHot() {
   int temp = CandyBarMaker.CheckNougatTemperature();
   if (temp > 160) {
     return true;
   } else {
     return false;
   }
}
```

When we rename the class "CandyBarMaker" and the method "CheckNougatTemperature" it starts to make the code easier to understand.

Notice how the C in CandyBarMaker is uppercase? If we always start class names with an uppercase letter and variables with lowercase ones, it's easier to tell when you're calling a static method versus using an instance.

Did you notice the special /// comments above the method? That's called an *XML Documentation Comment*. The IDE uses those comments to show you documentation for methods—like the documentation you saw when you used the IntelliSense window to figure out which method from the Random class to use.

IDE Tip: XML documentation for methods and fields

Visual Studio helps you add XML documentation. Put your cursor in the line above any method and type three slashes, and it will add an empty template for your documentation. If your method has parameters and a return type, it will add **<param>** and **<returns>** tags for them as well. Try going back to your CardPicker class and typing /// in the line above the PickSomeCards method—the IDE will add blank XML documentation. Fill it in and watch it show up in IntelliSense.

```
/// <summary>
/// Picks a number of cards and returns them.
/// </summary>
/// <param name="numberOfCards">The number of cards to pick.</param>
/// <returns>An array of strings that contain the card names.</returns>
```

You can create XML documentation for your fields, too. Try it out by going to the line just above any field and typing three slashes in the IDE. Anything you put after **<summary>** will show up in the IntelliSense window for the field.

What does the manual say to do if the nougat is too hot? It tells us to perform the candy isolation cooling system (or CICS) vent procedure. So let's make another method, and choose an obvious name for the T class (which turns out to control the turbine) and the ics class (which controls the isolation cooling system, and has two static methods to fill and vent the system), and cap it all off with some brief XML documentation:

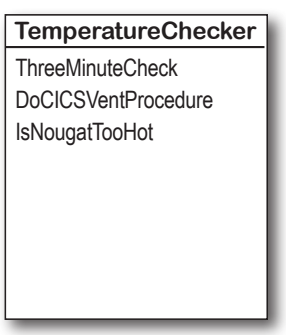
```
/// <summary>
/// Perform the Candy Isolation Cooling System (CICS) vent procedure.
/// </summary>
public void DoCICSVentProcedure() {
   TurbineController turbines = new TurbineController();
   turbines.CloseTripValve(2);
   IsolationCoolingSystem.Fill();
   IsolationCoolingSystem.Vent();
   Maker.CheckAirSystem();
}
```

When your method is declared with a void return type, that means it doesn't return a value and it doesn't need a `return` statement. All of the methods you wrote in the last chapter used the `void` keyword!

Now that we have the IsNougatTooHot and DoCICSVentProcedure methods, we can ***rewrite the original confusing code as a single method***—and we can give it a name that makes clear exactly what it does:

```
/// <summary>
/// This code runs every 3 minutes to check the temperature.
/// If it exceeds 160C we need to vent the cooling system.
/// </summary>
public void ThreeMinuteCheck() {
   if (IsNougatTooHot() == true) {
      DoCICSVentProcedure();
   }
}
```

We bundled these new methods into a class called TemperatureChecker. Here's its class diagram.



Now the code is a lot more intuitive! Even if you don't know that the CICS vent procedure needs to be run if the nougat is too hot, **it's a lot more obvious what this code is doing**.

Use class diagrams to plan out your classes

A class diagram is valuable tool for designing your code BEFORE you start writing it. Write the name of the class at the top of the diagram. Then write each method in the box at the bottom. Now you can see all of the parts of the class at a glance—and that's your first chance to spot problems that might make your code difficult to use or understand later.

That's right. When you change the structure of your code without altering its behavior, it's called refactoring.

Great developers write code that's as easy as possible to understand, even after they haven't looked at it for a long time. Comments can help, but nothing beats choosing intuitive names for your methods, classes, variables, and fields.

You can make your code easier to read and write by thinking about the problem your code was built to solve. If you choose names for your methods that make sense to someone who understands that problem, then your code will be a lot easier to decipher and develop. No matter how well we plan our code, we almost never get things exactly right the first time.

That's why ***great developers constantly refactor their code***. They'll move code into methods and give them names that make sense. They'll rename variables. Any time they see code that isn't 100% obvious, they'll take a few minutes to refactor it. They know it's worth taking the time to do it now, because it will make it easier to add more code in an hour (or a day, a month, or a year!).

Sharpen your pencil

Each of these classes has a serious design flaw. Write down what you think is wrong with each class, and how you'd fix it.

Class23

CandyBarWeight
PrintWrapper
GenerateReport
Go

This class is part of the candy manufacturing system from earlier.

.....

.....

.....

.....

DeliveryGuy

AddAPizza
PizzaDelivered
TotalCash
ReturnTime

DeliveryGirl

AddAPizza
PizzaDelivered
TotalCash
ReturnTime

These two classes are part of a system that a pizza parlor uses to track the pizza orders that are out for delivery.

.....

.....

.....

.....

CashRegister

MakeSale
NoSale
PumpGas
Refund
TotalCashInRegister
GetTransactionList
AddCash
RemoveCash

The CashRegister class is part of a program that's used by an automated convenience store checkout system.

.....

.....

.....

.....

Sharpen your pencil Solution

Here's how we improved the classes. We show just one possible way to fix the problems—but there are plenty of other ways you could design these classes depending on how they'll be used.

This class is part of the candy manufacturing system from earlier.

The class name doesn't describe what the class does. A programmer who sees a line of code that calls Class23.Go will have no idea what that line does. We'd also rename the method to something that's more descriptive—we chose MakeTheCandy, but it could be anything.

CandyMaker

CandyBarWeight
PrintWrapper
GenerateReport
MakeTheCandy

These two classes are part of a system that a pizza parlor uses to track the pizza orders that are out for delivery.

It looks like the DeliveryGuy class and the DeliveryGirl class both do the same thing—they represent a delivery person who's out delivering pizzas to customers. A better design would replace them with a single class that adds a field for gender.

DeliveryPerson

~~Gender~~

AddAPizza
PizzaDelivered
TotalCash
ReturnTime

We decided NOT to add a Gender field because there's actually no reason for this pizza delivery class to keep track of the gender of the people delivering pizza—and we should respect their privacy! Always look out for ways that bias can sneak into your code.

The CashRegister class is part of a program that's used by an automated convenience store checkout system.

All of the methods in the class do stuff that has to do with a cash register—making a sale, getting a list of transactions, adding cash...except for one: pumping gas. It's a good idea to pull that method out and stick it in another class.

CashRegister

MakeSale
NoSale
Refund
TotalCashInRegister
GetTransactionList
AddCash
RemoveCash

Code Tip: A few ideas for designing intuitive classes

We're about to jump back into writing code. You'll be writing code for the rest of this chapter, and a LOT of code throughout the book. That means you'll be ***creating a lot of classes***. Here are a few things to keep in mind when you make choices about how to design them:

- **You're building your program to solve a problem.**
  Spend some time thinking about that problem. Does it break down into pieces easily? How would you explain that problem to someone else? These are good things to think about when designing your classes.
- **What real-world things will your program use?**
  A program to help a zookeeper track her animals' feeding schedules might have classes for different kinds of food and types of animals.
- **Use descriptive names for classes and methods.**
  Someone should be able to figure out what your classes and methods do just by looking at their names.
- **Look for similarities between classes.**
  Sometimes two classes can be combined into one if they're really similar. The candy manufacturing system might have three or four turbines, but there's only one method for closing the trip valve that takes the turbine number as a parameter.

Relax

It's OK if you get stuck when you're writing code.
In fact, getting stuck can be a good thing!

Writing code is all about solving problems—and some of them can be tricky! But if you keep a few things in mind, it'll make the code exercises go more smoothly:

- *It's easy to get caught up in syntax problems, like missing parentheses or quotes. One missing bracket can cause many build errors.*
- *It's* **much better** *to look at the solution than to get frustrated with a problem. When you're frustrated, your brain doesn't like to learn.*
- *All of the code in this book is tested and definitely works! But it's easy to accidentally type things wrong (like typing a one instead of a lowercase L, or missing a comma or semicolon).*
- *If your solution just won't build, try downloading it from the GitHub repository for the book—it has working code for everything in the book: https://github.com/head-first-csharp/fifth-edition*

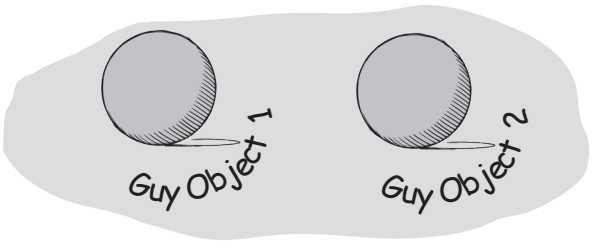
You can learn a lot from reading code. So if you run into a problem with a coding exercise, don't be afraid to peek at the solution. *It's not cheating!*

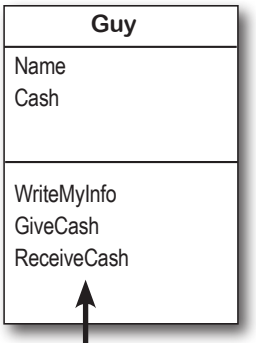
Build a class to work with some guys

Joe and Bob lend each other money all the time. Let's create a class to keep track of how much cash they each have. We'll start with an overview of what we'll build.

1 We'll create two instances of a "Guy" class.

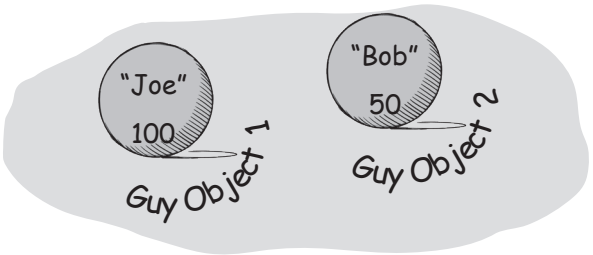
We'll use two Guy variables called `joe` and `bob` to keep track of each of our instances. Here's what the heap will look like after they're created:





2 We'll set each Guy object's Cash and Name fields.

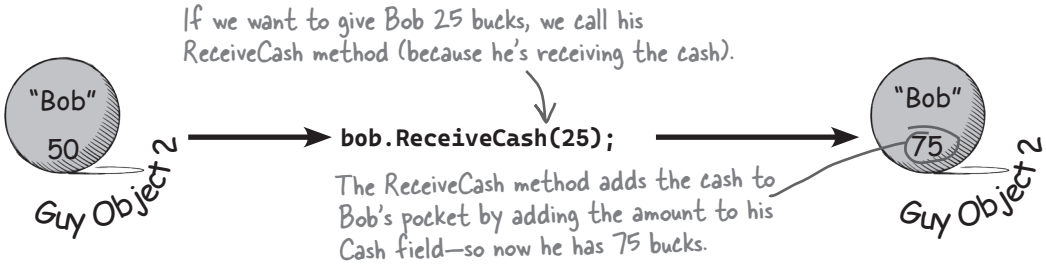
The two objects represent different guys, each with his own name and a different amount of cash in his pocket. Each guy has a Name field that keeps track of his name, and a Cash field that has the number of bucks in his pocket.



We chose names for the methods that make sense. You call a Guy object's GiveCash method to make him give up some of his cash, and his ReceiveCash method when you want to give cash to him (so he receives it).

3 We'll add methods to give and receive cash.

We'll make a guy give cash from his pocket (and reduce his Cash field) by calling his GiveCash method, which will return the amount of cash he gave. We'll make him receive cash and add it to his pocket (increasing his Cash field) by calling his ReceiveCash method.



```
internal class Guy
{
    public string? Name;
    public int Cash;

    /// <summary>
    /// Writes my name and the amount of cash I have to the console.
    /// </summary>
    public void WriteMyInfo()
    {
        Console.WriteLine(Name + " has " + Cash + " bucks.");
    }

    /// <summary>
    /// Gives some of my cash, removing it from my wallet (or printing
    /// a message to the console if I don't have enough cash).
    /// </summary>
    /// <param name="amount">Amount of cash to give.</param>
    /// <returns>
    /// The amount of cash removed from my wallet, or 0 if I don't
    /// have enough cash (or if the amount is invalid).
    /// </returns>
    public int GiveCash(int amount)
    {
        if (amount <= 0)
        {
            Console.WriteLine(Name + " says: " + amount + " isn't a valid amount");
            return 0;
        }
        if (amount > Cash)
        {
            Console.WriteLine(Name + " says: " +
                "I don't have enough cash to give you " + amount);
            return 0;
        }
        Cash -= amount;
        return amount;
    }

    /// <summary>
    /// Receive some cash, adding it to my wallet (or printing
    /// a message to the console if the amount is invalid).
    /// </summary>
    /// <param name="amount">Amount of cash to receive.</param>
    public void ReceiveCash(int amount)
    {
        if (amount <= 0)
        {
            Console.WriteLine(Name + " says: " + amount + " isn't an amount I'll take");
        }
        else
        {
            Cash += amount;
        }
    }
}
```

The Name and Cash fields keep track of the guy's name and how much cash he has in his pocket. Don't forget the question mark when you declare the **`string?`** field. We'll talk more about what that's about in the next chapter.

Sometimes you want to ask an object to perform a task, like printing a description of itself to the console.

The GiveCash and ReceiveCash methods verify that the amount they're being asked to give or receive is valid. That way you can't ask a guy to receive a negative number, which would cause him to lose cash.

Compare the comments in this code to the class diagrams and illustrations of the Guy objects. If something doesn't make sense at first, take the time to really understand it.

There's an easier way to initialize objects with C#

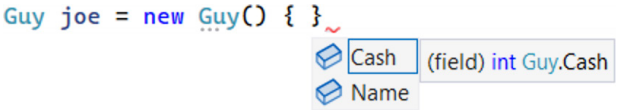
Almost every object that you create needs to be initialized in some way. The Guy object is no exception—it's useless until you set its Name and Cash fields. It's so common to have to initialize fields that C# gives you a shortcut for doing, it called an **object initializer**. The IDE's IntelliSense will help you do it.

You're about to do an exercise where you create two Guy objects. You ***could*** use one `new` statement and two more statements to set its fields:

```
joe = new Guy();
joe.Name = "Joe";
joe.Cash = 50;
```

Instead, type this: `Guy joe = new Guy() {`

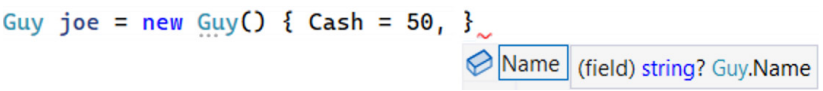
As soon as you add the left curly bracket, the IDE will pop up an IntelliSense window that shows all of the fields that you can initialize:



Choose the Cash field, set it to 50, and add a comma:

```
Guy joe = new Guy() { Cash = 50,
```

Now type a space—another IntelliSense window will pop up with the remaining field to set:

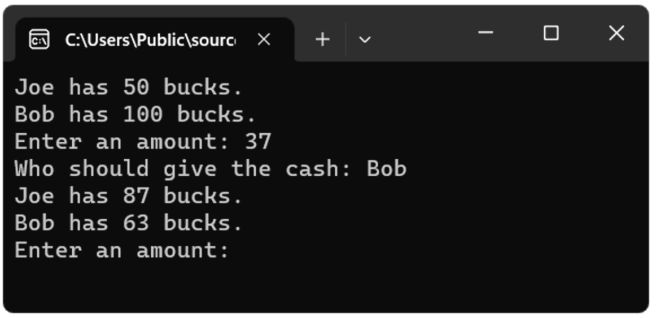


Set the Name field and add the semicolon. You now have a single statement that initializes your object:

```
Guy joe = new Guy() { Cash = 50, Name = "Joe" };
```

← This new declaration does the same thing as the three lines of code at the top of the page, but it's shorter and easier to read.

Now you have all of the pieces to build your console app that uses two instances of the Guy class. Here's what it will look like when it's running: →



Here's how it works. It calls each Guy object's WriteMyInfo method. It reads an amount from the input and asks who to give the cash to, then calls one Guy object's GiveCash method, then the other Guy object's ReceiveCash method. It keeps going until the user enters a blank line, then it calls return to exit the app.

Object initializers save you time and make your code more compact and easier to read...and the IDE helps you write them.

Exercise

This is part 1 of a two-part exercise.

Here are the top-level statements for a console app that makes Guy objects give cash to each other.

Step 1: **Create a new console app** that uses top-level statements. Name it Guys.

Step 2: Add a new class to your app called Guy. Since your project is called Guys, your new class will be in the namespace Guys. Carefully **add all of the code from the Guy class** that we just showed you.

Step 3: Here's the code that goes into your app's Program.cs file. Carefully enter it, then **replace the `comments` in with code**—read each comment and write code that does exactly what it says. When you're done, you'll have a program that looks like the screenshot on the previous page.

```
// Create a new Guy object in a variable called joe
// Set its Name field to "Joe"
// Set its Cash field to 50

// Create a new Guy object in a variable called bob
// Set its Name field to "Bob"
// Set its Cash field to 100

while (true)
{
    // Call the WriteMyInfo methods for each Guy object

    Console.Write("Enter an amount: ");
    string? howMuch = Console.ReadLine();
    if (howMuch == "") return;
    // Use int.TryParse to try to convert the howMuch string? to an int
    // if it was successful (just like you did earlier in the chapter)
    {
        Console.Write("Who should give the cash: ");
        string? whichGuy = Console.ReadLine();
        if (whichGuy == "Joe")
        {
            // Call the joe object's GiveCash method and save the results
            // Call the bob object's ReceiveCash method with the saved results
        }
        else if (whichGuy == "Bob")
        {
            // Call the bob object's GiveCash method and save the results
            // Call the joe object's ReceiveCash method with the saved results
        }
        else
        {
            Console.WriteLine("Please enter 'Joe' or 'Bob'");
        }
    }
    else
    {
        Console.WriteLine("Please enter an amount (or a blank line to exit).");
    }
}
```

Replace all of the comments with code that does what the comments describe.

Exercise Solution

Here are the top-level statements for your console app. It uses an infinite loop to keep asking the user how much cash to move between the Guy objects. If the user enters a blank line for an amount, the method executes a `return` statement, which causes Main to exit and the program to end.

```
Guy joe = new Guy() { Cash = 50, Name = "Joe" };
Guy bob = new Guy() { Cash = 100, Name = "Bob" };

while (true)
{
    joe.WriteMyInfo();
    bob.WriteMyInfo();
    Console.Write("Enter an amount: ");
    string? howMuch = Console.ReadLine();
    if (howMuch == "") return;
    if (int.TryParse(howMuch, out int amount))
    {
        Console.Write("Who should give the cash: ");
        string? whichGuy = Console.ReadLine();
        if (whichGuy == "Joe")
        {
            int cashGiven = joe.GiveCash(amount);
            bob.ReceiveCash(cashGiven);
        }
        else if (whichGuy == "Bob")
        {
            int cashGiven = bob.GiveCash(amount);
            joe.ReceiveCash(cashGiven);
        }
        else
        {
            Console.WriteLine("Please enter 'Joe' or 'Bob'");
        }
    }
    else
    {
        Console.WriteLine("Please enter an amount (or a blank line to exit).");
    }
}
```

When the app executes this return statement it ends the program, because console apps stop when the top-level statements finish running.

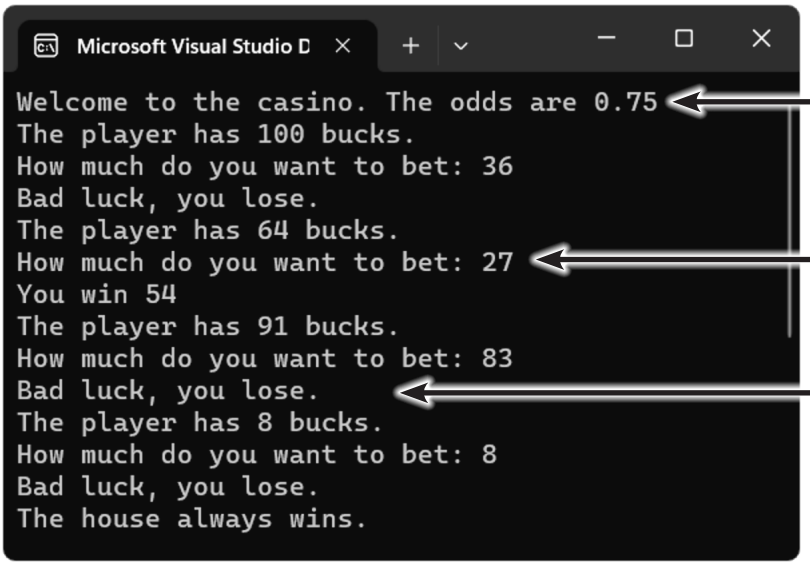
Here's the code where one Guy object gives cash from his pocket, and the other Guy object receives it.

Don't move on to the next part of the exercise until you have the first part working and you understand what's going on. It's worth taking a few minutes to use the debugger to step through the program and make sure you really get it.

Exercise

Here's the second part of the two-part exercise.

Now that you have your Guy class working, let's see if you can **reuse it in a betting game**. Look closely at this screenshot to see how it works and what it prints to the console.



These are the odds to beat.

The player makes a double-or-nothing bet each round.

The program picks a random double from 0 to 1. If the number is greater than the odds, the player wins twice their bet, otherwise the player loses.

Create a new console app that uses top-level statements, then add the Guy class from your Guys project. Make sure you **add a using statement to the top of your Program.cs file** so you can use the Guy class.

In your Program.cs, declare two variables:

- A double variable called **`odds`** that stores the odds to beat set to .75
- A Guy variable called **`player`** for an instance of Guy named "The player" with 100 bucks.

Your app should write a line to the console welcoming the player and printing the odds. Then it should run this loop:

1. Call the Guy object's WriteMyInfo method to write the amount of cash the player has to the console.
2. Write a line to the console asking the player how much money to bet.
3. Read the line from the console into a string variable called `howMuch`.
4. Try to parse it into an int variable called `amount`.
5. If it parses, the player gives the amount to an int variable called `pot`. Only do steps 6 through 9 if `pot` is greater than zero.
6. Multiply `pot` by two, because it's a double-or-nothing bet.
7. Use Random.Shared to pick a random double value between 0 and 1.
8. If the random value is greater than `odds`, the player receives the pot.
9. If not, the player loses the amount they bet.
10. The program keeps running while the player has cash.

In step 5, you'll call the Guy object's GiveCash method to give the amount to bet. The GiveCash method won't give more cash than the guy has, so you don't need to check if the player has enough money. The Guy class will write a message to the console if it doesn't have enough cash to place the bet, so your app doesn't have to. Checking if the pot variable is greater than zero makes sure the bet is valid and the player has enough cash.

The loop ends when the player runs out of money, then the app prints a message: "The house always wins."

Exercise Solution

Here's the working code for the top-level statements in the betting game. Can you think of ways to make it more fun? See if you can figure out how to add additional players, or give different options for odds, or maybe you can think of something more clever. ***This is a chance to get creative!***

...and to get some practice. Getting practice writing code is the best way to become a great developer.

```
using Guys;

double odds = .75;
Random random = new Random();

Guy player = new Guy() { Cash = 100, Name = "The player" };

Console.WriteLine("Welcome to the casino. The odds are " + odds);
while (player.Cash > 0)
{
    player.WriteMyInfo();
    Console.Write("How much do you want to bet: ");
    string? howMuch = Console.ReadLine();
    if (int.TryParse(howMuch, out int amount))
    {
        int pot = player.GiveCash(amount) * 2;
        if (pot > 0)
        {
            if (Random.Shared.NextDouble() > odds)
            {
                int winnings = pot;
                Console.WriteLine("You win " + winnings);
                player.ReceiveCash(winnings);
            } else
            {
                Console.WriteLine("Bad luck, you lose.");
            }
        }
    } else
    {
        Console.WriteLine("Please enter a valid number.");
    }

}
Console.WriteLine("The house always wins.");
```

Was your code a little different than ours? If it still works and produces the right output, that's OK! There are many different ways to write the same program.

...and as you get further along in the book and the exercise solutions get longer, your code will look more and more different from ours. Remember, it's always OK to look at our solution when you're working on an exercise!

Brain Power

Is Guy really the best name for the class? Why or why not? Can you think of a better name for it?

Sharpen your pencil

Here's an app that writes three lines to the console. Your job is to figure out what it writes, without using a computer. Start at the first line of the Main method and keep track of the values of each of the fields in the objects as it runs.

```
class Pizzazz
{
    public int Zippo;

    public void Bamboo(int eek)
    {
        Zippo += eek;
    }
}

class Abracadabra
{
    public int Vavavoom;

    public bool Lala(int floq)
    {
        if (floq < Vavavoom)
        {
            Vavavoom += floq;
            return true;
        }
        return false;
    }
}

Pizzazz foxtrot = new Pizzazz() { Zippo = 2 };
foxtrot.Bamboo(foxtrot.Zippo);

Pizzazz november = new Pizzazz() { Zippo = 3 };
Abracadabra tango = new Abracadabra() { Vavavoom = 4 };

while (tango.Lala(november.Zippo))
{
    november.Zippo *= -1;
    november.Bamboo(tango.Vavavoom);
    foxtrot.Bamboo(november.Zippo);
    tango.Vavavoom -= foxtrot.Zippo;
}

Console.WriteLine("november.Zippo = " + november.Zippo);
Console.WriteLine("foxtrot.Zippo = " + foxtrot.Zippo);
Console.WriteLine("tango.Vavavoom = " + tango.Vavavoom);
```

What does this program write to the console?

november.Zippo = ...

foxtrot.Zippo = ...

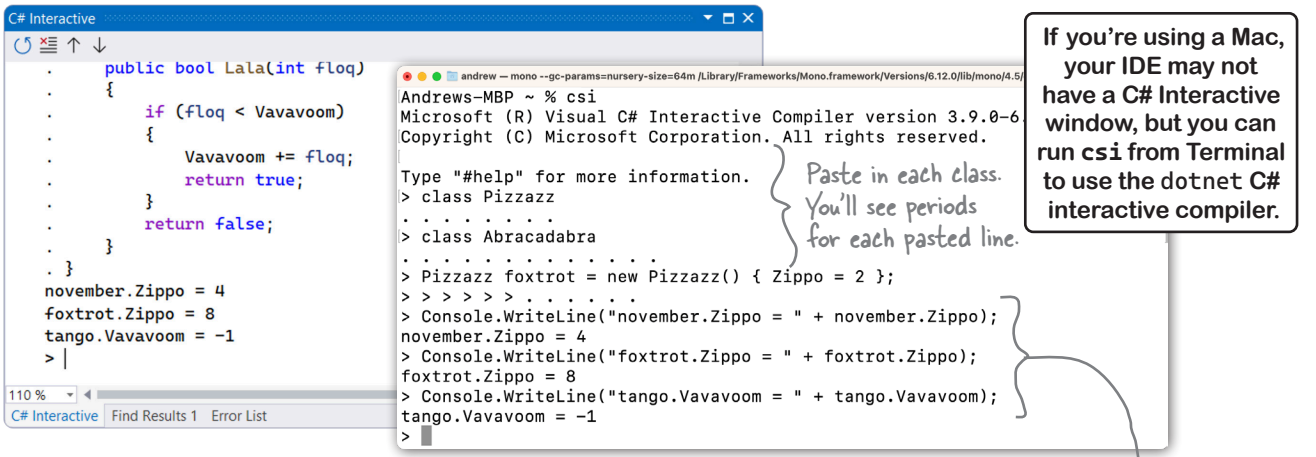
tango.Vavavoom = ...

To find the solution, enter the program into Visual Studio and run it. If you didn't get the answer right, step through the code line by line and add watches for each of the objects' fields.

If you don't want to type the whole thing in, you can download it from GitHub: https://github.com/head-first-csharp/fifth-edition.

Use the C# Interactive window or csi to run C# code

If you just want to run some C# code, you don't always need to create a new project in Visual Studio. Any C# code entered into the **C# Interactive window** is run immediately. You can open it by choosing View >> Other Windows >> C# Interactive. Try it now, and **paste in the code** from the exercise solution to see the output. You can call methods and enter other statements too.



You can also run an interactive C# session from the command line. On Windows, search the Start menu for `developer command prompt`, start it, and then type **`csi`**. On macOS, run **`csi`** from the Terminal. You can paste the Pizzazz, Abracadabra, and Program classes from the previous exercise directly into the prompt, then paste in the code that you want to run.

Bullet Points

- Use the **`new` keyword** to create instances of a class. A program can have many instances of the same class.
- Each **instance** has all of the methods from the class and gets its own copies of each of the fields.
- When you included `new Random();` in your code, you were creating an **instance of the Random class**.
- Use the **`static` keyword** to declare a field or method in a class as static. You don't need an instance of that class to access static methods or fields.
- When a field is **static**, there's only one copy of it shared by all instances. When you include the `static` keyword in a class declaration, all of its members must be static.
- Fields and methods of a class are called its **members**.
- If you remove the `static` keyword from a static field, it becomes an **instance field**.
- When your program creates an object, it lives in a part of the computer's memory called the **heap**.
- Visual Studio helps you add **XML documentation** to your fields and methods, and displays it in its IntelliSense window.
- **Class diagrams** help you plan out your classes and make them easier to work with.
- When you change the structure of your code without altering its behavior, it's called **refactoring**. Advanced developers constantly refactor their code.
- **Object initializers** save you time and make your code more compact and easier to read.

Managing your app's data

Data and objects are the building blocks of your apps.

What would your apps be without data? Think about it for a minute. Without data, your programs are…well, it's actually hard to imagine writing code without data. You need **information** from your users, and you use that to look up or produce new information to give back to them. In fact, almost everything you do in programming involves **working with data** in one way or another. In this chapter, you'll learn the ins and outs of C#'s **data types** and **references**, see how to work with data in your program, and even learn a few more things about **objects** (guess what…objects are data, too!).

Owen could use our help!

Owen is a game master—a really good one. He hosts a group that meets at his place every week to play different **role-playing games** (or **RPGs**), and like any good game master, he really works hard to keep things interesting for the players.

Storytelling, fantasy, and mechanics

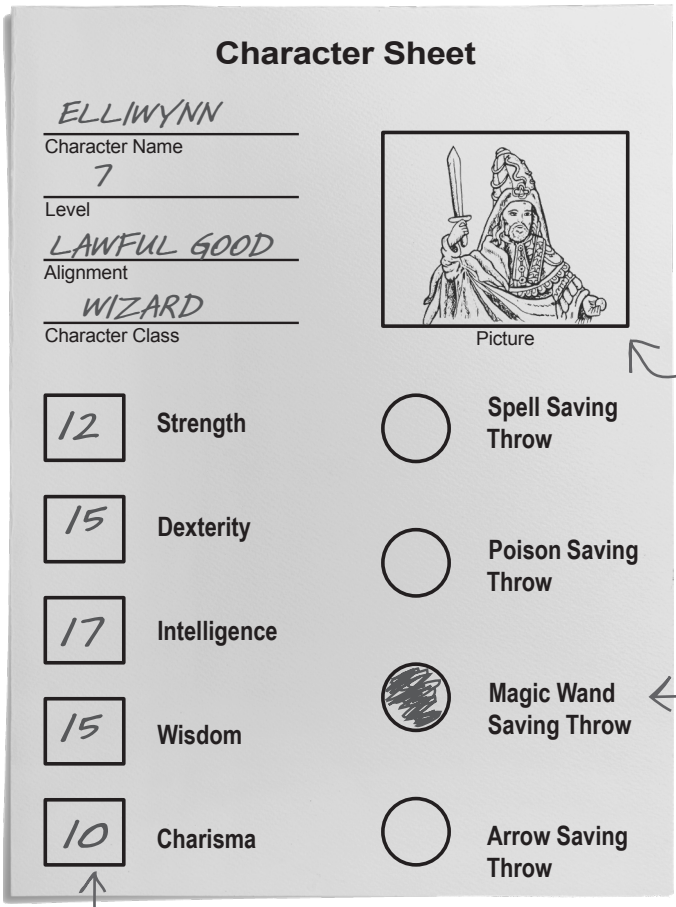
Owen is a particularly good storyteller. Over the last few months he's created an intricate fantasy world for his party, but he's not so happy with the mechanics of the game that they've been playing.

Can we find a way to help Owen improve his RPG?

Ability score (like strength, stamina, charisma, and intelligence) is an important mechanic in a lot of role-playing games. Players frequently roll dice and use a formula to determine their character's scores.

Character sheets store different types of data on paper

If you've ever played an RPG, you've seen character sheets: a page with details, statistics, background information, and any other notes you might see about a character. If you wanted to make a class to hold a character sheet, what types would you use for the fields?

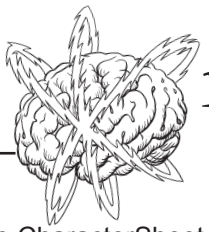


Players create characters by rolling dice for each of their ability scores, which they write in these boxes.

CharacterSheet
CharacterName Level PictureFilename Alignment CharacterClass Strength Dexterity Intelligence Wisdom Charisma SpellSavingThrow PoisonSavingThrow MagicWandSavingThrow ArrowSavingThrow
ClearSheet GenerateRandomScores

This box is for a picture of the character. If you were building a C# class for a character sheet, you could save that picture in an image file.

In the RPG that Owen plays, saving throws give players a chance to roll dice and avoid certain types of attacks. This character has a magic wand saving throw, so the player filled in this circle.



Brain Power

Look at the fields in the CharacterSheet class diagram. What type would you use for each field?

A variable's type determines what kind of data it can store

There are many **types** built into C#, and you'll use them to store many different kinds of data. You've already seen some of the most common ones, like int, string, bool, and float. There are a few others that you haven't seen, and they can really come in handy, too.

Here are some types you'll use a lot.

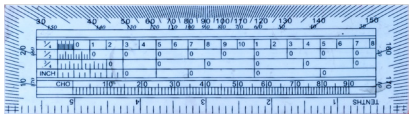
★ **bool** is a Boolean value—it's either true or false. You'll use it to represent anything that only has two options: it can either be one thing or another, but nothing else.

Better a witty fool, than a foolish wit.

★ **string** can hold text of any length (including the empty string `""`).

★ **int** can store any **integer** from –2,147,483,648 to 2,147,483,647. Integers don't have decimal points.

★ **double** can store **real** numbers from ±5.0 × 10−324 to ±1.7 × 10308 with up to 16 significant digits. It's a really common type when you're working with XAML properties.



★ **float** can store **real** numbers from ±1.5 × 10−45 to ±3.4 × 1038 with up to 8 significant digits.

Brain Power

Why do you think C# has more than one type for storing numbers that have a decimal point?

C# has several types for storing integers

C# has several different types for integers, as well as int. This may seem a little odd (pun intended). Why have so many types for numbers without decimals? For most of the programs in this book, it won't matter if you use an int or a long. If you're writing a program that has to keep track of millions and millions of integer values, then choosing a smaller integer type like byte instead of a bigger type like long can save you a lot of memory.

- ***byte*** can store any **integer** between 0 and 255.
- ***sbyte*** can store any **integer** from –128 to 127.
- ***short*** can store any **integer** from –32,768 to 32,767.
- ***long*** can store any **integer** from –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

Notice how we're saying "integer" and not "whole number"? We're trying to be really careful—our high school math teachers always told us that integers are any numbers that can be written without a fraction, while whole numbers are integers starting at 0, and do not include negative numbers.

byte only stores small whole numbers from 0 to 255.

If you need to store a larger number, you can use a short, which stores integers from –32,768 to 32,767.

Long also stores integers, but it can store huge values.

Did you notice that byte only stores positive numbers, while sbyte stores negative numbers? They both have 256 possible values. The difference is that, like short and long, sbyte can have a negative sign—which is why those are called **signed** types, (the "s" in sbyte stands for signed). Just like byte is the **unsigned** version of sbyte, there are unsigned versions of short, int, and long that start with "u":

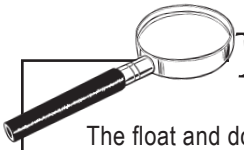
- ***ushort*** can store any **whole number** from 0 to 65,535.
- ***uint*** can store any **whole number** from 0 to 4,294,967,295.
- ***ulong*** can store any **whole number** from 0 to 18,446,744,073,709,551,615.

Types for storing really HUGE and really tiny numbers

Sometimes float just isn't precise enough. Believe it or not, sometimes 1038 isn't big enough and 10–45 isn't small enough. A lot of programs written for finance or scientific research run into these problems all the time, so C# gives us different **floating-point types** to handle huge and tiny values:

- ***float*** can store any number from ±1.5 × 10–45 to ±3.4 × 1038 with 6–9 significant digits.
- ***double*** can store any number from ±5.0 × 10–324 to ±1.7 × 10308 with 15–17 significant digits.
- ***decimal*** can store any number from ±1.0 × 10–28 to ±7.9 × 1028 with 28–29 significant digits. When your program **needs to deal with money or currency**, you always want to use a decimal to store the number.

The decimal type has a lot more precision (way more significant digits) which is why it's appropriate for financial calculations.



Floating Point Numbers Up Close

The float and double types are called "floating-point" because the decimal point can move (as opposed to a "fixed-point" number, which always has the same number of decimal places). That—and, in fact, a lot stuff that has to do with floating-point numbers, especially precision—may seem a little ***weird***, so let's dig into the explanation.

"Significant digits" represents the precision of the number: 1,048,415, 104.8415, and .0000001048415 all have 7 significant digits. So when we say a float can store real numbers as big as 3.4 × 1038 or as small as –1.5 × 10-45, that means it can store numbers as big as 8 digits followed by 30 zeros, or as small as 37 zeros followed by 8 digits.

If it's been a while since you've used exponents, 3.4x1038 means 34 followed by 37 zeros, and –1.5x10–45 is –.00...(40 more zeros)...0015.

The float and double types can also have special values, including both positive and negative zero, positive and negative infinity, and a special value called **NaN (not-a-number)** that represents, well, a value that isn't a number at all. They also have static methods that let you test for those special values. Try running this loop:

```
for (float f = 10; !float.IsInfinity(f); f *= f)
{
    Console.WriteLine(f);
}
```

Now try that same loop with double:

```
for (double d = 10; !double.IsInfinity(d); d *= d)
{
    Console.WriteLine(d);
}
```

Let's talk about strings

You've written code that works with **strings**. So what, exactly, is a string?

In any .NET app, a string is an object. Its full class name is System.String—in other words, the class name is String and it's in the System namespace (just like the Random class you used earlier). When you use the C# `string` keyword, you're working with System.String objects. In fact, you can replace `string` with `System.String` in any of the code you've written so far and it will still work! (The `string` keyword is called an *alias*—as far as your C# code is concerned, `string` and `System.String` mean the same thing.)

There are also two special values for strings: an empty string, "" (or a string with no characters), and a null string, or a string that isn't set to anything at all. We'll talk more about null later in the chapter.

Strings are made up of characters—specifically, Unicode characters (which you'll learn a lot more about later in the book). Sometimes you need to store a single character like `Q` or `j` or `$`, and when you do you'll use the **char** type. Literal values for char are always inside single quotes (`'x'`, `'3'`). You can include **escape sequences** in the quotes, too (`'\n'` is a line break, `'\t'` is a tab). You can write an escape sequence in your C# code using two characters, but your program stores each escape sequence as a single character in memory.

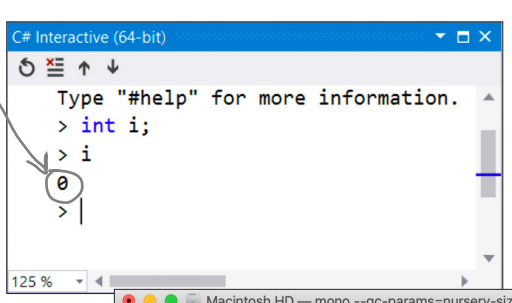
And finally, there's one more important type: **object**. If a variable has object as its type, ***you can assign any value to it***. The `object` keyword is also an alias—it's the same as `System.Object`.

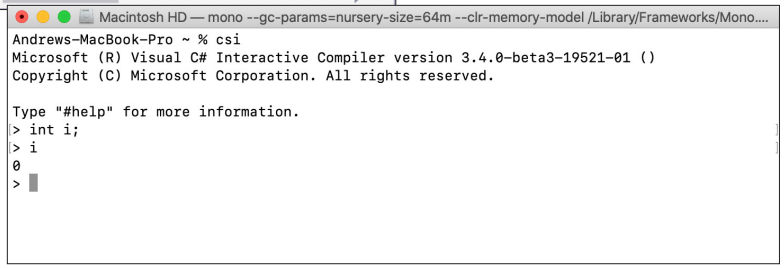
Sharpen your pencil

Sometimes you declare a variable and set its value in a single statement like this: `int i = 37;`—but you already know that you don't have to set a value. What happens if you use the variable without assigning a value? Let's find out! Use the **C# Interactive window** (or the **csi** if you're using a Mac) to declare a variable and check its value.

We wrote in the first answer for you.

```
0 ..... int i;
..... long l;
..... float f;
..... double d;
..... decimal m;
..... byte b;
..... char c;
..... string s;
..... bool t;
```





Start the C# Interactive window (from the View >> Other Windows menu) or run `csi` from the Mac Terminal. Declare each variable, then enter the variable name to see its default value. Write the default value for each type in the space provided.

A literal is a value written directly into your code

A **literal** is a number, string, or other fixed value that you include in your code. You've already used plenty of literals—here are some examples of numbers, strings, and other literals that you've used:

```
int number = 15;
string result = "the answer";
public bool GameOver = false;
Console.Write("Enter the number of cards to pick: ");
if (value == 1) return "Ace";
```

Can you spot all of the literals in these statements from code you've written in previous chapters? The last statement has two literals.

So when you type `int i = 5;`, the `5` is a literal.

Use suffixes to give your literals types

Go back to the first loop you wrote in the "Up Close" section and change 10 to 10**D**:

```
for (float f = 10D; float.IsFinite(f); f *= f)
```

Now your program won't build—you'll get an error that mentions a literal of type double. That's because **literals have types**. Every literal is automatically assigned a type, and C# has rules about how you can combine different types. You can see for yourself how that works. Add this line to any C# program:

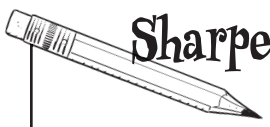
```
int wholeNumber = 14.7;
```

When you try to build your program, the IDE will show you this error in the Error List:

⊗	CS0266	Cannot implicitly convert type 'double' to 'int'. An explicit conversion exists (are you missing a cast?)

The IDE is telling you is that the literal 14.7 has a type—it's a double. You can use a suffix to change its type—try changing it to a float by sticking an F on the end (14.7F) or a decimal by adding M (14.7M—the M actually stands for "money"). The error message now says it can't convert float or decimal.

C# assumes that an integer literal without a suffix (like 371) is an int, and one with a decimal point (like 27.4) is a double.



Sharpen your pencil Solution

```
0       int i;
0       long l;
0       float f;
0       double d;
0       decimal m;
0       byte b;
'\0'    char c;
null    string s;
false   bool t;
```

If you used the C# command line on Mac or Unix, you might see '\x0' instead of '\0' as the default value for char. We'll take a deep dive into exactly what this means later in the book when we talk about Unicode.

Sharpen your pencil

C# has dozens of **reserved words called keywords**. They're words reserved by the C# compiler that you can't use for variable names. You've already learned many of them—here's a little review to help seal them into your brain. Write down what you think each of these keywords does in C#.

`namespace`

`for`

`class`

`else`

`new`

`using`

`if`

`while`

If you really want to use a reserved keyword as a variable name, put @ in front of it, but that's as close as the compiler will let you get to the reserved word. You can also do that with nonreserved names, if you want to.

Sharpen your pencil Solution

C# has dozens of **reserved words called keywords**. They're words reserved by the C# compiler that you can't use for variable names. You've already learned many of them—here's a little review to help seal them into your brain. Write down what you think each of these keywords does in C#.

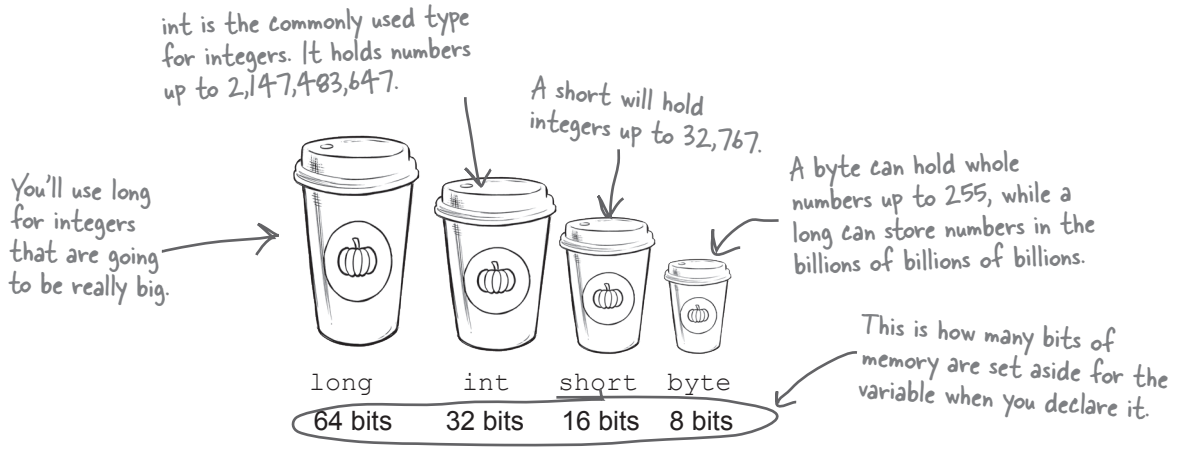
namespace	All of the classes and methods in a program are inside a namespace. Namespaces help make sure that the names you are using in your program don't clash with the ones in the .NET Framework or other classes.
for	This lets you do a loop that executes three statements. First it declares the variable it's going to use, then there's the statement that evaluates the variable against a condition. The third statement does something to the value.
class	Classes contain methods and fields, and you use them to instantiate objects. Fields are what objects know and methods are what they do.
else	A block of code that starts with else must immediately follow an if block, and will get executed if the if statement preceding it fails.
new	You use this to create a new instance of an object.
using	This is a way of listing off all of the namespaces you are using in your program. A using statement lets you use classes from various parts of the .NET Framework.
if	This is one way of setting up a conditional statement in a program. It says if one thing is true, do one thing; if not, do something else.
while	while loops are loops that keep on going as long as the condition at the beginning of the loop is true.

A variable is like a data to-go cup

All of your data takes up **space in memory**. (Remember the heap from the last chapter?) So part of your job is to think about how *much* space you're going to need whenever you use a string or a number in your program. That's one of the reasons you use variables. They let you set aside enough space in memory to store your data.

Not all data ends up on the heap. Value types usually keep their data in another part of memory called the stack. You'll learn all about that later in the book.

Think of a variable like a cup that you keep your data in. C# uses a bunch of different kinds of cups to hold different kinds of data. Just like the different sizes of cups at a coffee shop, there are different sizes of variables, too.



Use the Convert class to explore bits and bytes

Convert this!

You've always heard that programming is about 1s and 0s. .NET has a **static Convert class** that converts between different numeric data types. Let's use it to see an example of how bits and bytes work.

A bit is a single 1 or 0. A byte is 8 bits, so a byte variable holds an 8-bit number, which means it's a number that can be represented with up to 8 bits. What does that look like? Let's use the Convert class to convert some binary numbers to bytes:

```
Convert.ToByte("10111", 2)     // returns 23
Convert.ToByte("11111111", 2); // returns 255
```

The first argument to Convert.ToByte is the number to convert, and the second is its base. Binary numbers are base 2.

Bytes can hold numbers between 0 and 255 because they use 8 bits of memory—an 8-bit number is a binary number between 0 and 11111111 binary (or 0 and 255 decimal).

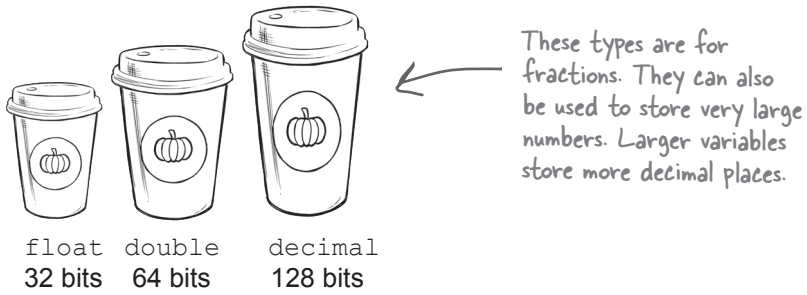
A short is a 16-bit value. Let's use Convert.ToInt16 to convert the binary value 111111111111111 (15 1s) to a short. An int is a 32-bit value, so we'll use Convert.ToInt32 to convert the 31 1s to an int:

```
Convert.ToInt16("111111111111111", 2);                 // returns 32767
Convert.ToInt32("11111111111111111111111111111111", 2); // returns 2147483647
```

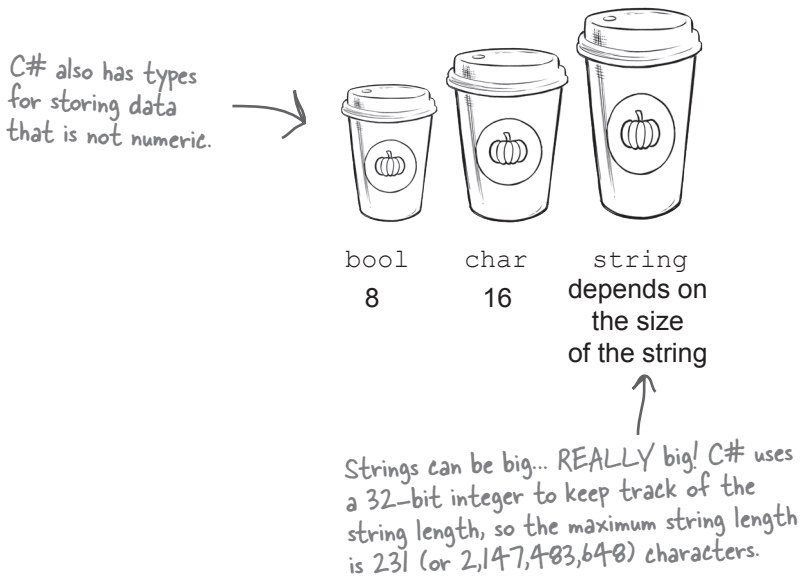
Other types come in different sizes, too

Numbers that have decimal places are stored differently than integers, and the different floating-point types take up different amounts of memory. You can handle most of your numbers that have decimal places using **float**, the smallest data type that stores decimals. If you need to be more precise, use a **double**. If you're writing a financial application where you'll be storing currency values, you'll always want to use the **decimal** type.

Oh, and one more thing: ***don't use double for money or currency, only use decimal.***



We've talked about strings, so you know that the C# compiler also can handle **characters and non-numeric types**. The char type holds one character, and string is used for lots of characters "strung" together. There's no set size for a string object—it expands to hold as much data as you need to store in it. The bool data type is used to store true or false values, like the ones you've used for your `if` statements.



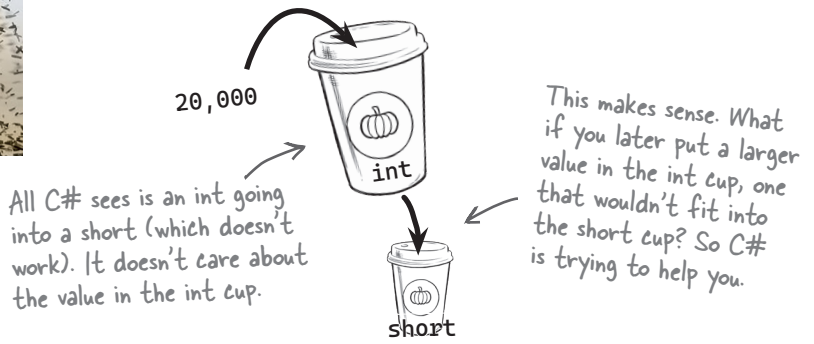
The different floating-point types take up different amounts of memory: float is smallest, and decimal is largest.

10 pounds of data in a 5-pound bag

When you declare your variable as one type, the C# compiler **allocates** (or reserves) all of the memory it would need to store the maximum value of that type. Even if the value is nowhere near the upper boundary of the type you've declared, the compiler will see the cup it's in, not the number inside. So this won't work:

```
int leaguesUnderTheSea = 20000;
short smallerLeagues = leaguesUnderTheSea;
```

20,000 would fit into a `short`, no problem. But because `leaguesUnderTheSea` is declared as an int, C# sees it as int-sized and considers it too big to put in a short container. The compiler won't make those translations for you on the fly. You need to make sure that you're using the right type for the data you're working with.



Sharpen your pencil

Three of these statements won't build, either because they're trying to cram too much data into a small variable or because they're putting the wrong type of data in. Circle them and write a brief explanation of what's wrong

```
int hours = 24;

short y = 78000;

bool isDone = yes;

short RPM = 33;

int balance = 345667 - 567;
```

```
string taunt = "your mother";

byte days = 365;

int radius = 3;

char initial = 'S';

string months = "12";
```

Casting lets you copy values that C# can't automatically convert to another type

Let's see what happens when you try to assign a decimal value to an int variable.

Do this!

1. Create a new Console App project and add this code to your Program.cs:

```
float myFloatValue = 10;
int myIntValue = myFloatValue;
Console.WriteLine("myIntValue is " + myIntValue);
```

Implicit conversion means C# has a way to automatically convert a value to another type without losing information.

2. Try building your program. You should get the same CS0266 error you saw earlier:

CS0266 Cannot implicitly convert type 'float' to 'int'. An explicit conversion exists (are you missing a cast?)

Look closely at the last few words of the error message: "are you missing a cast?" That's the C# compiler giving you a really useful hint about how to fix the problem.

3. Make the error go away by **casting** the decimal to an int. You do this by adding the type that you want to convert to in parentheses: **`(int)`**. Once you change the second line so it looks like this, your program will compile and run:

```
int myIntValue = (int) myFloatValue;
```

Here's where you cast the decimal value to an int.

When you cast a floating-point value to an int, it rounds the value down to the nearest integer.

So what happened?

The C# compiler won't let you assign a value to a variable if it's the wrong type—even if that variable can hold the value just fine! It turns out that a LOT of bugs are caused by type problems, and **the compiler is helping** by nudging you in the right direction. When you use casting, you're essentially saying to the compiler that you know the types are different, and promising that in this particular instance it's OK for C# to cram the data into the new variable.

Sharpen your pencil Solution

Three of these statements won't build, either because they're trying to cram too much data into a small variable or because they're putting the wrong type of data in. Circle them and write a brief explanation of what's wrong

```
short y = 78000;
```

The short type holds numbers from –32,767 to 32,768. This number's too big!

```
byte days = 365;
```

A byte can only hold a value between 0 and 255. You'll need a short for this.

```
bool isDone = yes;
```

You can only assign a value of "true" or "false" to a bool.

When you cast a value that's too big, C# adjusts it to fit its new container

You've already seen that a float can be cast to an int. It turns out that *any* number can be cast to *any other* number. That doesn't mean the ***value*** stays intact through the casting, though. Say you have an int variable set to 365. If you cast it to a byte variable (max value 255), instead of giving you an error, the value will just **wrap around**. 256 cast to a byte will have a value of 0, 257 will be converted to 1, 258 to 2, etc., up to 365, which will end up being **109**. Once you get back to 255 again, the conversion value "wraps" back to zero.

If you use + (or `*`, `/`, or `-`) with two different numeric types, the operator **automatically converts** the smaller type to the bigger one. Here's an example:

```
int myInt = 36;
float myFloat = 16.4F;
myFloat = myInt + myFloat;
```

Since an int can fit into a float but a float can't fit into an int, the + operator converts `myInt` to a float before adding it to `myFloat`.

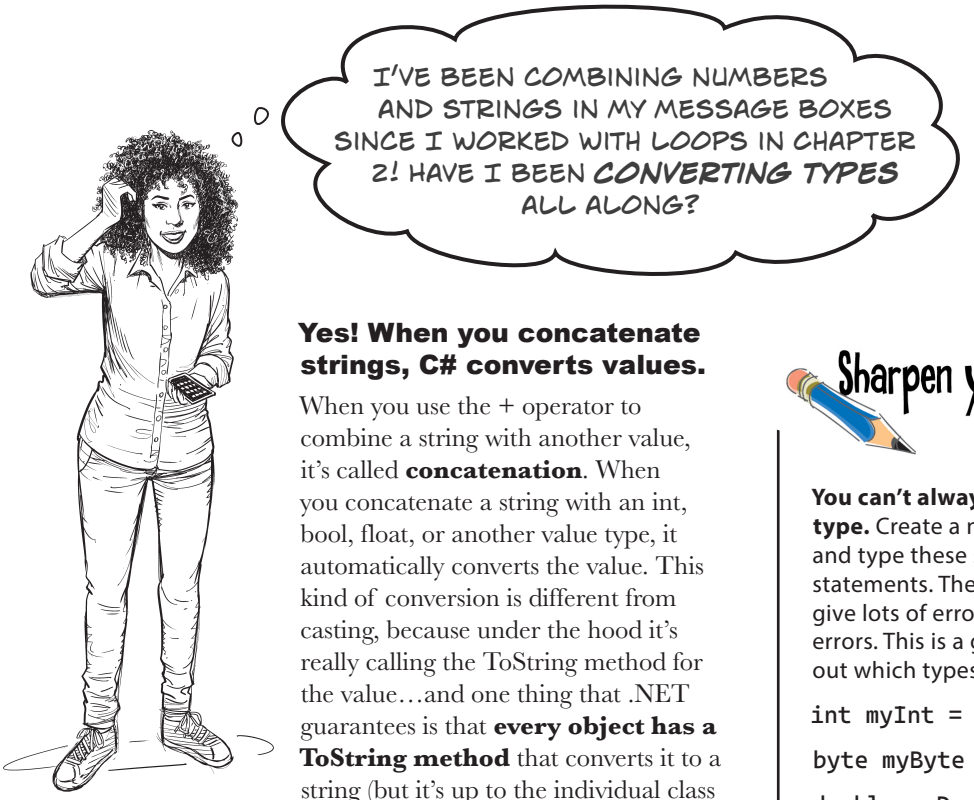
Sharpen your pencil

You can't always cast any type to any other type.

Create a new Console App project and type these statements into its top-level statements. Then build your program—it will give lots of errors. Cross out the ones that give errors. This is a great way to help you figure out which types can be cast, and which can't.

```
int myInt = 10;
byte myByte = (byte)myInt;
double myDouble = (double)myByte;
bool myBool = (bool)myDouble;
string myString = "false";
myBool = (bool)myString;
myString = (string)myInt;
myString = myInt.ToString();
myBool = (bool)myByte;
myByte = (byte)myBool;
short myShort = (short)myInt;
char myChar = 'x';
myString = (string)myChar;
long myLong = (long)myInt;
decimal myDecimal = (decimal)myLong;
myString = myString + myInt + myByte +
myDouble + myChar;
```

You can read a lot more about the different C# value types here—it's worth taking a look:
https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/value-types



Yes! When you concatenate strings, C# converts values.

When you use the + operator to combine a string with another value, it's called **concatenation**. When you concatenate a string with an int, bool, float, or another value type, it automatically converts the value. This kind of conversion is different from casting, because under the hood it's really calling the ToString method for the value…and one thing that .NET guarantees is that **every object has a ToString method** that converts it to a string (but it's up to the individual class to determine if that string makes sense).

Wrap it yourself!

There's no mystery to how casting "wraps" the numbers—you can do it yourself. Just open up any calculator app that has a Mod button (which does a modulus calculation—sometimes in a Scientific mode), and calculate 365 Mod 256.

Sharpen your pencil Solution

You can't always cast any type to any other type. Create a new Console App project and type these statements into its top-level statements. Then build your program—it will give lots of errors. Cross out the ones that give errors. This is a great way to help you figure out which types can be cast, and which can't.

```
int myInt = 10;
byte myByte = (byte)myInt;
double myDouble = (double)myByte;
bool myBool = (bool)myDouble;
string myString = "false";
myBool = (bool)myString;
myString = (string)myInt;
myString = myInt.ToString();
myBool = (bool)myByte;
myByte = (byte)myBool;
short myShort = (short)myInt;
char myChar = 'x';
myString = (string)myChar;
long myLong = (long)myInt;
decimal myDecimal = (decimal)
myLong;
myString = myString + myInt +
myByte + myDouble + myChar;
```

C# does some conversions automatically

There are two important conversions that don't require you to do casting. The first is the automatic conversion that happens any time you use arithmetic operators, like in this example:

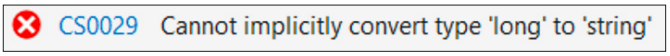
```
long l = 139401930;
short s = 516;
double d = l - s;
d = d / 123.456;
Console.WriteLine("The answer is " + d);
```

The - operator subtracted the short from the long, and the = operator converted the result to a double.

The other way C# converts types for you automatically is when you use the + operator to **concatenate** strings (which just means sticking one string on the end of another, like you've been doing with message boxes). When you use + to concatenate a string with something that's another type, it automatically converts the numbers to strings for you. Here's an example—try adding these lines to any C# program. The first two lines are fine, but the third one won't compile:

```
long number = 139401930;
string text = "Player score: " + number;
text = number;
```

The C# compiler gives you this error on the third line:



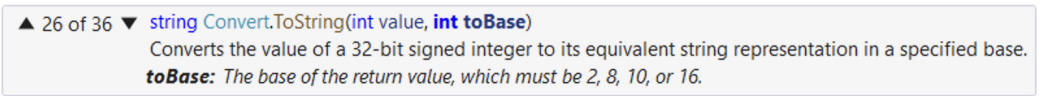
ScoreText.text is a string field, so when you used the + operator to concatenate a string it assigned the value just fine. But when you try to assign `x` to it directly, it doesn't have a way to automatically convert the long value to a string. You can convert it to a string by calling its ToString method.

there are no Dumb Questions

Q: You used the Convert.ToByte, Convert.ToInt32, and Convert.ToInt64 methods to convert strings with binary numbers into integer values. Can you convert integer values back to binary?

A: Yes. The Convert class has a **Convert.ToString method** that converts many different types of values to strings. The IntelliSense pop-up shows you how it works:

```
Console.WriteLine(Convert.ToString(8675309, 2));
```



So `Convert.ToString(255, 2)` returns the string "11111111", and `Convert.ToString(8675309, 2)` returns the string "100001000101111111101101"—try experimenting with it to get a feel for how binary numbers work.

When you call a method, the arguments need to be compatible with the types of the parameters

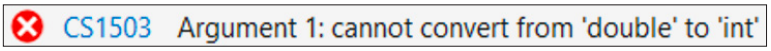
In the last chapter, you used the Random class to choose a random number from 1 up to (but not including) 5, which you used to pick a suit for a playing card:

```
int value = Random.Shared.Next(1, 5);
```

Try changing the first argument from `1` to `1.0`:

```
int value = Random.Shared.Next(1.0, 5);
```

You're passing a double literal to a method that's expecting an int value. So it shouldn't surprise you that the compiler won't build your program—instead, it shows an error:



Sometimes C# can do the conversion automatically. It doesn't know how to convert a double to an int (like converting 1.0 to 1), but it does know how to convert an int to a double (like converting 1 to 1.0). More specifically:

- The C# compiler knows how convert an integer to a floating-point type.
- And it knows how to convert an integer type to another integer type, or a floating-point type to another floating-point type.
- But it can only do those conversions if the type it's converting from is the same size as or smaller than the type it's converting to. So, it can convert an int to a long or a float to a double, but it can't convert a long to an int or a double to a float.

When the compiler gives you an "invalid argument" error, it means that you tried to call a method with variables whose types didn't match the method's parameters.

But Random.Shared.Next isn't the only method that will give you compiler errors if you try to pass it a variable whose type doesn't match the parameter. *All* methods will do that, ***even the ones you write yourself***. Add this method to a console app:

```
public int MyMethod(bool add3) {
   int value = 12;

   if (add3)
      value += 3;
   else
      value -= 2;

   return value;
}
```

Try passing it a string or long—you'll get one of those CS1503 errors telling you it can't convert the argument to a bool. Some folks have trouble remembering **the difference between a parameter and an argument**. So just to be clear:

A parameter is what you define in your method. An argument is what you pass to it. You can pass a byte argument to a method with an int parameter.

there are no Dumb Questions

Q: That last `if` statement only said `if (add3)`. Is that the same thing as `if (add3 == true)`?

A: Yes. Let's take another look at that `if/else` statement:

```
if (add3)
   value += 3;
else
   value -= 2;
```

An `if` statement always checks if something's true. So because the type of the `add3` variable is bool, it evaluates to either true or false, which means we didn't have to explicitly include `== true`.

You can also check if something's false using ! (an exclamation point, or the NOT operator). Writing `if (!add3)` is the same thing as writing `if (add3 == false)`.

In our code examples from now on, if we're using the conditional test to check a Boolean variable, you'll usually just see us write `if (add3)` or `if (!add3)`, and not use == to explicitly check to see if the Boolean is true or false.

Q: You didn't include curly braces in the `if` or `else` blocks, either. Does that mean they're optional?

A: Yes—but only if there's a single statement in the `if` or `else` block. We could leave out the `{` curly braces `}` because there was just one statement in the `if` block (`return 45;`) and one statement in the `else` block (`return 61;`). If we wanted to add another statement to one of those blocks, we'd have to use curly braces for it:

```
if (add3)
   value += 3;
else {
   Console.WriteLine("Subtracting 2");
   value -= 2;
}
```

Be careful when you leave out curly braces because it's easy to accidentally write code that doesn't do what you want it to do. It never hurts to add curly braces, but it's also good to get used to seeing `if` statements both with and without them.

Bullet Points

- There are value types for variables that hold different sizes of numbers. The biggest numbers should be of type long and the smallest ones (up to 255) can be declared as bytes.
- Every value type has a **size**, and you can't put a value of a bigger type into a smaller variable, no matter what the actual size of the data is.
- When you're using **literal** values, use the F suffix to indicate a float (15.6F) and M for a decimal (36.12M).
- Use the **decimal type for money and currency**. Floating-point precision is…well, it's a little weird.
- There are a few types that C# knows how to **convert** automatically (an implicit conversion), like short to int, int to double, or float to double.
- When the compiler won't let you set a variable equal to a value of a different type, that's when you need to cast it. To **cast** a value (an explicit conversion) to another type, put the target type in parentheses in front of the value.
- There are some keywords that are **reserved** by the language and you can't name your variables with them. They're words (like `for`, `while`, `using`, `new`, and others) that do specific things in the language.
- A **parameter** is what you define in your method. An **argument** is what you pass to it.
- When you build your code in the IDE, it uses the **C# compiler** to turn it into an executable program.
- You can use methods on the static **Convert class** to convert values between different types.

Owen is constantly improving his game...

Good game masters are dedicated to creating the best experience they can for their players. Owen's players are about to embark on a new campaign with a brand-new set of characters, and he thinks a few tweaks to the formula that they use for their ability scores could make things more interesting.

When players fill out their character sheets at the start of the game, they follow these steps to calculate each of the ability scores for their character.

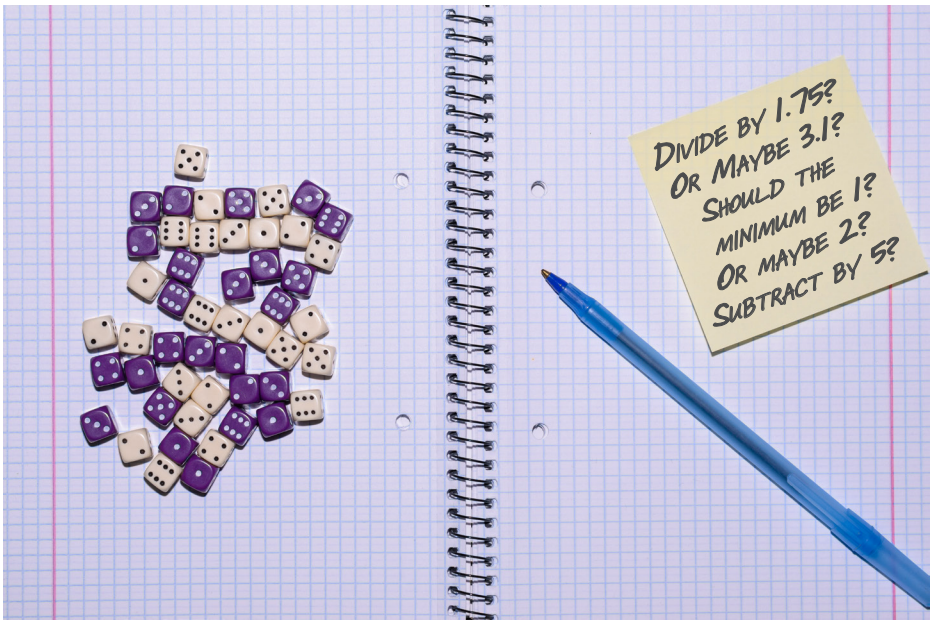
ABILITY SCORE FORMULA

* Start with a **4d6** roll to get a number between **4** and **24**
* Divide the roll result by **1.75**
* Add **2** to the result of that division
* Round down to the nearest whole number
* If the result is too small, use the minimum value of **3**

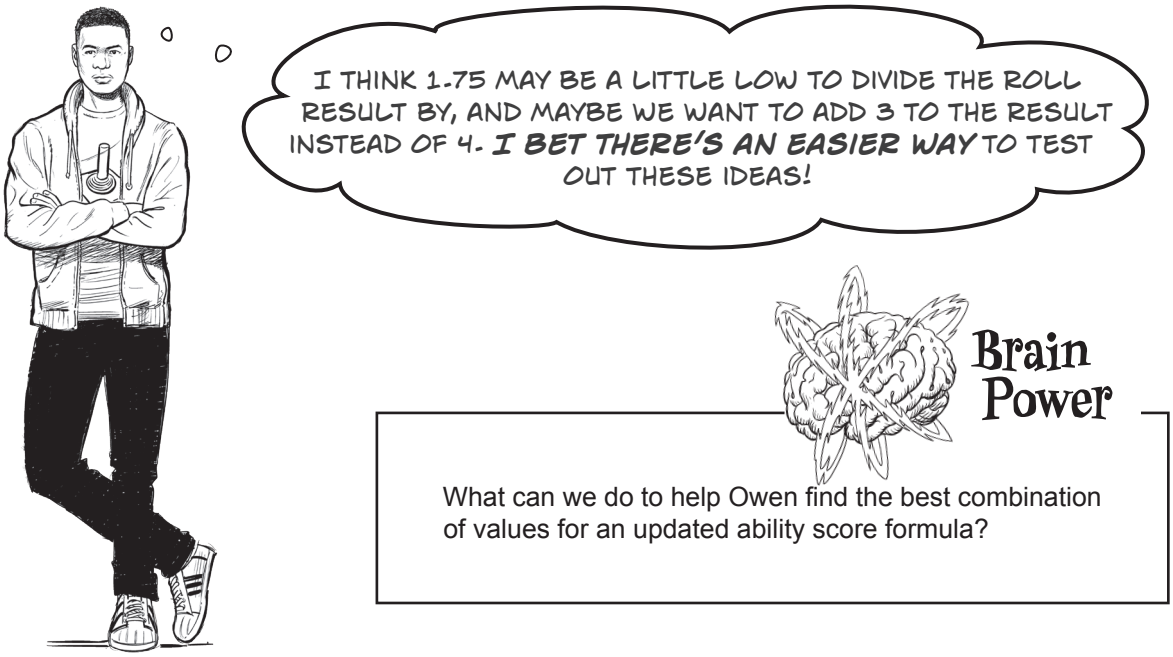
A "4d6 ROLL" means rolling four normal six-sided dice and adding up the results.

...but the trial and error can be time-consuming

Owen's been experimenting with ways to tweak the ability score calculation. He's pretty sure that he has the formula mostly right—but he'd really like to tweak the numbers.



Owen likes the overall formula: 4d6 roll, divide, subtract, round down, use a minimum value…but he's not sure that the actual numbers are right.



Brain Power

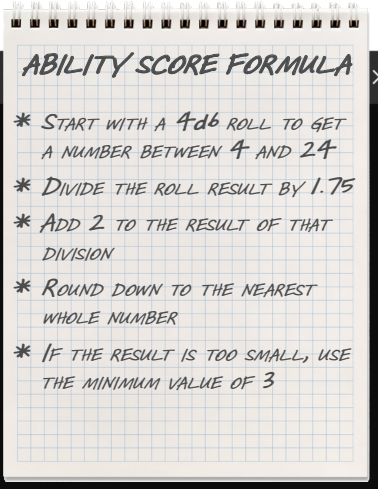
What can we do to help Owen find the best combination of values for an updated ability score formula?

Let's help Owen experiment with ability scores

In this next project, you'll build a .NET Core console app that Owen can use to test his ability score formula with different values to see how they affect the resulting score. The formula has **four inputs**: the *starting 4d6 roll*, the *divide by* value that the roll result is divided by, the *add amount* value to add to the result of that division, and the *minimum* to use if the result is too small.

Owen will enter each of the four inputs into the app, and it will calculate the ability score using those inputs. He'll probably want to test a bunch of different values, so we'll make the app easier by to use by asking for new values over and over again until he quits the app, keeping track of the values he used in each iteration and using those previous inputs as **default values** for the next iteration.

Here's the page from Owen's game master notebook with the ability score formula.



This is what it looks like when Owen runs the app:

This project is a little larger than the previous console apps that you've built, so we'll tackle it in a few steps. First you'll Sharpen your Pencil to understand the code to calculate the ability score. Then you'll do an Exercise to write the rest of the code for the app. And finally, you'll Sleuth out a bug in the code. Let's get started!

Sharpen your pencil

We've built a class to help Owen calculate ability scores. To use it, you'll set its RollResult, DivideBy, AddAmount, and Minimum fields—or just leave the values set in their declarations—and call its CalculateAbilityScore method.

Create a new Console App project called AbilityScore and **add a class called AbilityScoreCalculator**. Enter all of the code into the class. Uh-oh! There's one line of code that has a problem. **Circle the line of code** that causes a compiler error. Then **write down what you think you'll need to do to fix it**.

```
internal class AbilityScoreCalculator
{
    public int RollResult = 14;
    public double DivideBy = 1.75;
    public int AddAmount = 2;
    public int Minimum = 3;
    public int Score;

    public void CalculateAbilityScore()
    {
        // Divide the roll result by the DivideBy field
        double divided = RollResult / DivideBy;

        // Add AddAmount to the result and round down
        int added = AddAmount += divided;

        // If the result is too small, use Minimum
        if (added < Minimum)
        {
            Score = Minimum;
        } else
        {
            Score = added;
        }
    }
}
```

These fields are initialized with the values from the ability score formula. The app will use them to present default values to the user.

Here's a hint! Visual Studio will show you which line of code won't compile, and underline the specific part that has problems. Hover over that part and look closely at the error message.

After you **circle the line of code that won't compile**, write down what you need to do to fix the compiler error.

.....

.....

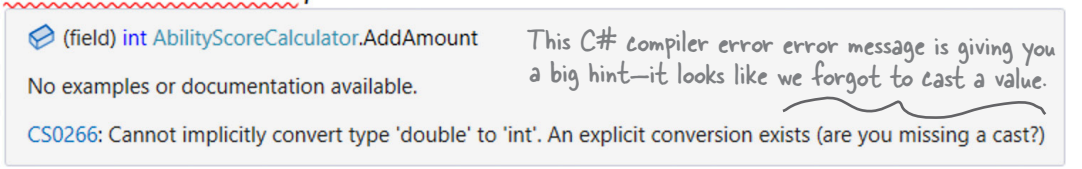
Fix the compiler error by adding a cast

Sharpen your pencil Solution

If you entered the code correctly, you should see a C# compiler error on this line of code:

```
int added = AddAmount += divided;
```

Here's the line of code to circle in the Sharpen Your Pencil exercise.



This C# compiler error error message is giving you a big hint—it looks like we forgot to cast a value.

Any time the C# compiler gives you an error, read it carefully. It often has a hint that can help you track down the problem. This error telling us exactly what went wrong: ***it can't convert a double to an int without a cast***. The `divided` variable is declared as a double, but C# won't allow you to add it to an int field like AddAmount because it doesn't know how to convert it. **So here's the answer** to the Sharpen Your Pencil question:

> After you **circle the line of code that has problems**, look at the error and write down what you need to do to fix it.
>
> The compiler error says it can't convert a double to an int, and asks if our code is missing a cast. To fix it, we need to use (int) to cast the double to an int so += will be able to add the values.

When the C# compiler asks "are you missing a cast?" it's giving you a huge hint that you need to **cast the double variable `divided`** before you can add it to the int field AddAmount.

But this isn't the whole answer! There's still something wrong with that line of code. Can you spot it?

Add a cast to get the AbilityScoreCalculator class to compile...

Now that you know what the problem is, you can **add a cast** to fix the problematic line of code in AbilityScoreCalculator. The line that caued the error because **`AddAmount += divided` *returns a double value***. When you try to store a double value in an int variable like `added` you'll get a "Cannot implicity convert type" error.

You can fix it by **casting `divided` to an int**, so adding it to AddAmount returns another int. Modify that line of code to change `divided` to `(int)divided`:

```
int added = AddAmount += (int)divided;
```

Cast this!

Adding that cast also addresses an important part of Owen's ability score formula:

* *Round down to the nearest whole number*

When you cast a double to an int C# rounds it down—so for example `(int)19.7431D` gives us `19`. By adding that cast, you're making sure the score is rounded down, like Owen's formula asks for.

...but there's still a bug!

We're not quite done yet! You fixed the compiler error, so now the project builds. But even though the C# compiler will accept it, ***there's still a bug in the code***. So let's go ahead and fix it! In the next exercise you'll use the AbilityScoreCalculator class as is, then you'll use it to sleuth out the bug.

Exercise

Finish building the console app that uses the AbilityScoreCalculator class. In this exercise, we'll give you the top-level statements for the console app. Your job is to write code for two methods: a method called ReadInt that reads user input and converts it to an int using int.TryParse, and a method called ReadDouble that does exactly the same thing except it parses doubles instead of int values.

Step 1: In this first step, you'll add top-level statements to your Program.cs file. Almost everything was used in previous projects. There's only one new thing—it calls the Console.ReadKey method:

```
char keyChar = Console.ReadKey(true).KeyChar;
```

Console.ReadKey reads a single key from the console. When you pass the argument **`true`** it intercepts the input so that it doesn't get printed to the console. Adding **`.KeyChar`** causes it to return the key pressed as a **`char`**.

Delete the "Hello, World!" line from your Program.cs file and **add these top-level statements**:

```
using AbilityScore;

AbilityScoreCalculator calculator = new AbilityScoreCalculator();
while (true)
{
    calculator.RollResult = ReadInt(calculator.RollResult, "Starting 4d6 roll");
    calculator.DivideBy = ReadDouble(calculator.DivideBy, "Divide by");
    calculator.AddAmount = ReadInt(calculator.AddAmount, "Add amount");
    calculator.Minimum = ReadInt(calculator.Minimum, "Minimum");
    calculator.CalculateAbilityScore();
    Console.WriteLine("Calculated ability score: " + calculator.Score);
    Console.WriteLine("Press Q to quit, any other key to continue");
    char keyChar = Console.ReadKey(true).KeyChar;
    if ((keyChar == 'Q') || (keyChar == 'q')) return;
}
```

Step 2: The code you wrote calls a method called ReadInt, so **add a static ReadInt method**. The ReadInt method takes two parameters: a string called **prompt** to display to the user, and an int called **defaultValue**. It writes the prompt to the console, followed by the default value in square brackets. Then it reads a line from the console and attempts to parse it with int.TryParse. If that returns true, return that value; otherwise, return the default value.

Here's the declaration:

```
static int ReadInt(int defaultValue, string prompt)
```

Calling `ReadInt("37", "What's the magic number?")` will cause the following prompt to be printed:

```
What's the magic number? [37]
```

There's a space at the end of that prompt. The user then types in a value and presses enter. The method reads that line from the console and calls int.TryParse to try to parse it. If int.TryParse returns true, the method returns the result. If it returns false, the method returns defaultValue—in this case, 37.

Step 3: Generate and implement the ReadDouble method. ReadDouble is exactly like ReadInt, except that **it uses double.TryParse** instead of int.TryParse. The double.TryParse method works exactly like int.TryParse, except its **`out`** variable needs to be a double, not an int.

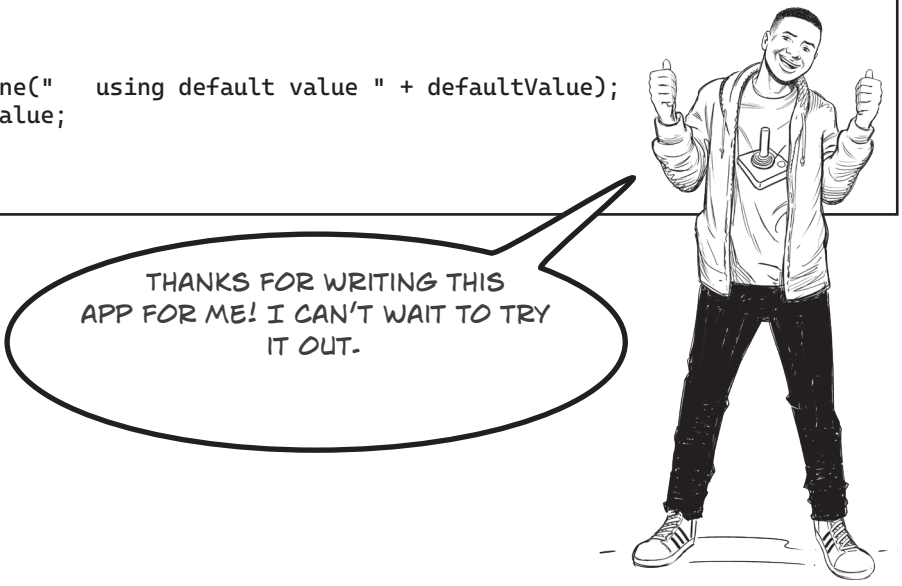
```
static double ReadDouble(double defaultValue, string prompt)
```

Exercise Solution

Here are the ReadInt and ReadDouble methods that display a prompt that includes the default value, read a line from the console, try to convert it to an int or a double, and either use the converted value or the default value, writing a message to the console with the value returned.

Really take some time to understand how each iteration of the `while` loop in the top-level statements uses fields to save the values that the user entered, then uses them for the default values in the next iteration.

```
static int ReadInt(int defaultValue, string prompt)
{
    Console.Write(prompt + " [" + defaultValue + "]: ");
    string? line = Console.ReadLine();
    if (int.TryParse(line, out int value))
    {
        Console.WriteLine("   using value " + value);
        return value;
    } else
    {
        Console.WriteLine("   using default value " + defaultValue);
        return defaultValue;
    }
}

static double ReadDouble(double defaultValue, string prompt)
{
    Console.Write(prompt + " [" + defaultValue + "]: ");
    string? line = Console.ReadLine();
    if (double.TryParse(line, out double value))
    {
        Console.WriteLine("   using value " + value);
        return value;
    }
    else
    {
        Console.WriteLine("   using default value " + defaultValue);
        return defaultValue;
    }
}
```



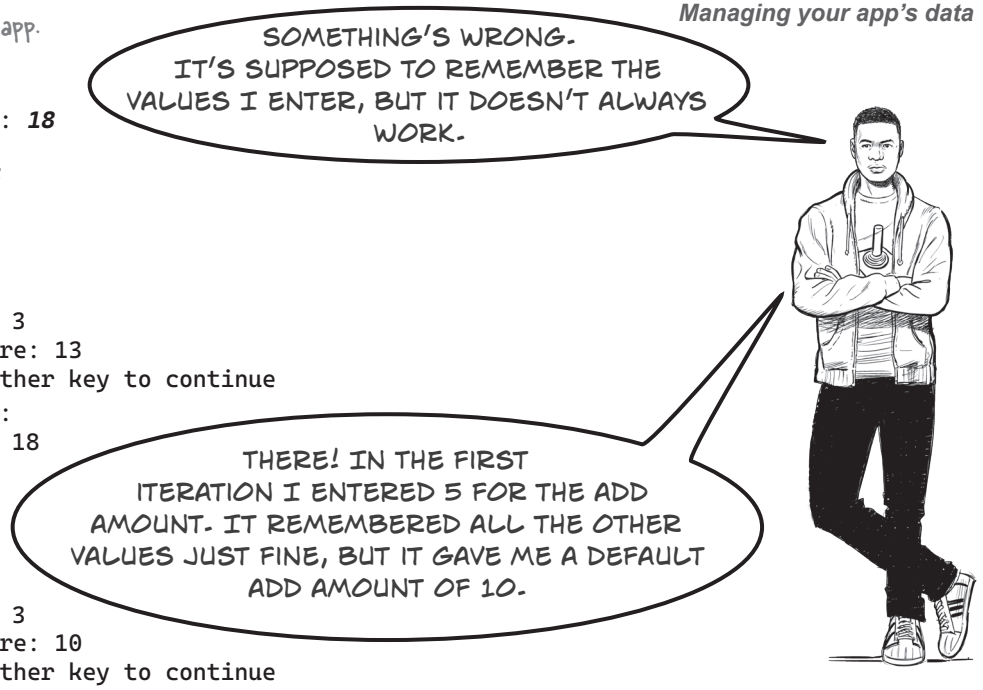
Here's the output from the app.

```
Starting 4d6 roll [14]: 18
   using value 18
Divide by [1.75]: 2.15
   using value 2.15
Add amount [2]: 5
   using value 5
Minimum [3]:
   using default value 3
Calculated ability score: 13
Press Q to quit, any other key to continue
Starting 4d6 roll [18]:
   using default value 18
Divide by [2.15]: 3.5
   using value 3.5
Add amount [13]: 5
   using value 5
Minimum [3]:
   using default value 3
Calculated ability score: 10
Press Q to quit, any other key to continue
Starting 4d6 roll [18]:
   using default value 18
Divide by [3.5]:
   using default value 3.5
Add amount [10]: 7
   using value 7
Minimum [3]:
   using default value 3
Calculated ability score: 12
Press Q to quit, any other key to continue
Starting 4d6 roll [18]:
   using default value 18
Divide by [3.5]:
   using default value 3.5
Add amount [12]: 4
   using value 4
Minimum [3]:
   using default value 3
Calculated ability score: 9
Press Q to quit, any other key to continue
Starting 4d6 roll [18]:
   using default value 18
Divide by [3.5]:
   using default value 3.5
Add amount [9]:
   using default value 9
Minimum [3]:
   using default value 3
Calculated ability score: 14
Press Q to quit, any other key to continue
```

That's strange. Owen entered 5 for the previous add amount, but the program is giving him 10 as a default option.

Again, the last amount Owen entered was 7, but it's giving 12 as a default option. Weird.

Where did this 9 number come from? Did we see it before? Can that give us a hint about what's causing this bug?

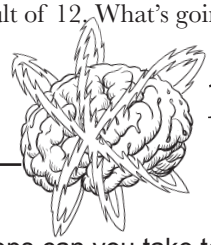


You're right, Owen. There's a bug in the code.

Owen wants to try out different values to use in his ability score formula, so we used a loop to make the app ask for those values over and over again.

To make it easier for Owen to just change one value at a time, we included a feature in the app that remembers the last values he entered and presents them as default options. We implemented that feature by keeping an instance of the AbilityScoreCalculator class in memory, and updating its fields in each iteration of the `while` loop.

But something's gone wrong with the app. It remembers most of the values just fine, but it remembers the wrong number for the "add amount" default value. In the first iteration Owen entered 5, but it gave him 10 as a default option. Then he entered 7, but it gave a default of 12. What's going on?



Brain Power

What steps can you take to track down the bug in the ability score calculator app?

The Case of the Operator Oddity

The debugger is like a detective's magnifying glass. It helps you spot even the smallest clues.

Let's do an investigation and see if we can apprehend the culprit, Sherlock Holmes style. ***Something*** is causing the bug, so let's use the debugger to identify suspects and retrace their steps.

The problem seems to be isolated to the "add amount" value, so let's start by looking for any line of code that touches the AddAmount field. Here's the line that uses the AddAmount field—put a breakpoint on it:

```
39   calculator.DivideBy = ReadDouble(calculator.DivideBy, "Divide by");
40   calculator.AddAmount = ReadInt(calculator.AddAmount, "Add amount");
41   calculator.Minimum = ReadInt(calculator.Minimum, "Minimum");
```

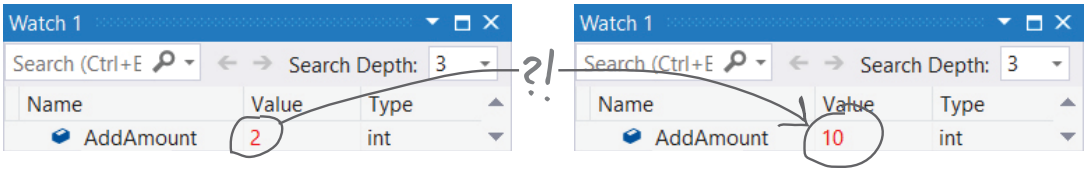
Here's another one in the AbilityScoreCalculator.CalculateAbilityScore method—breakpoint that suspect, too:

```
20   // Add to the result
21   int added = AddAmount += (int)divided;
```

The trap is set. Let's see who springs it.

Now run your program. When your code hits the breakpoint, **select calculator.AddAmount and add a watch** (if you just right-click on AddAmount and choose "Add Watch" from the menu, it will only add a watch for AddAmount and not calculator.AddAmount). Does anything look weird there? We're not seeing anything unusual. It seems to read the value and update it just fine—that's probably not the issue. You can delete that breakpoint.

Continue running your program. When the breakpoint in AbilityScoreCalculator.CalculateAbilityScore is hit, **add a watch for AddAmount**. According to Owen's formula, this line of code is supposed to add AddAmount to the result of dividing the roll result. Now **step over** the statement and...



Wait, what?! AddAmount changed. But...but that's not supposed to happen—it's impossible! Right? As Sherlock Holmes said, "When you have eliminated the impossible, whatever remains, however improbable, must be the truth."

It looks like we've sleuthed out the source of the problem. That statement is supposed to cast `divided` to an int to round it down to an integer, then add it to AddAmount and store the result in `added`. It also has an unexpected side effect: it's updating AddAmount with the sum because **the statement uses the += operator**, which returns the sum but assigns the sum to AddAmount.

Now we can finally fix Owen's bug—and get the REAL "Sharpen" answer

Now that you know what's happening, you can **fix the bug**—and it turns out to be a pretty small change. You just need to change the statement to use + instead of +=:

```
int added = AddAmount + (int)divided;
```

Change the += to a + to keep this line of code from updating the "added" variable and fix the bug.

And we can finally have the ***real*** answer to the "Sharpen Your Pencil" question in the first part of this project.

After you **circle the line of code that has problems**, look at the error and write down what you need to do to fix it.

First, it won't compile because AddAmount += divided is a double, so a cast needs to happen to assign it to an int. Second, it uses += and not +, which causes the line to update AddAmount.

there are no Dumb Questions

Q: I'm still not clear on the difference between the + operator and the += operator. How do they work, and why would I use one and not the other?

A: There are several operators that you can combine with an equals sign. They include += for adding, -= for subtracting, /= for dividing, *= for multiplying, and %= for remainder. Operators like + that combine two values are called **binary operators**. Some people find this name a little confusing, but "binary" refers to the fact that the operator combines two values—"binary" means "involving two things"—not that it somehow operates only on binary numbers.

With binary operators, you can do something called **compound assignment**, which means instead of this:

```
a = a + c;
```

you can do this:

The += operator tells C# to add a + c and then store the result in a.

```
a += c;
```

and it means the same thing. The compound assignment `x op= y` is equivalent to `x = x op y` (that's the technical way of explaining it). They do exactly the same thing.

Operators like += or *= that combine a binary operator with an equals sign are called compound assignment operators.

Q: But then how did the added variable get updated?

A: What caused confusion in the score calculator is that the **assignment operator = also returns a value**. You can do this:

```
int q = (a = b + c)
```

which will calculate `a = b + c` as usual. The = operator ***returns*** a value, so **it will update q with the result** as well. So:

```
int added = AddAmount += divided;
```

is just like doing this:

```
int added = (AddAmount = AddAmount +
divided);
```

which causes `AddAmount` to be increased by `divided`, but stores that result in `added` as well.

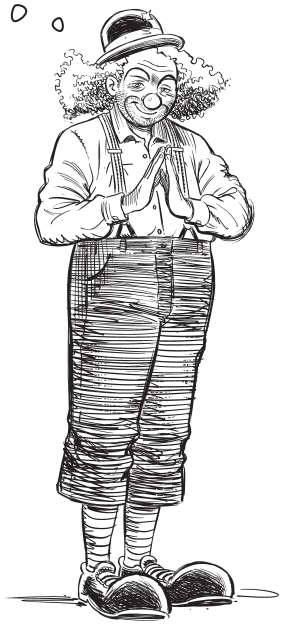
Q: Wait, what? The equals operator returns a value?

A: Yes, = returns the value being set. So in this code:

```
int first;
int second = (first = 4);
```

both `first` and `second` will end up equal to 4. Open up a console app and use the debugger to test this. It really works!

HEY, KID! WANNA SEE SOMETHING **WEIRD?**



Try this!

Try adding this `if/else` statement to a console app and build the solution:

```
if (0.1M + 0.2M == 0.3M) Console.WriteLine("They're equal");
else Console.WriteLine("They aren't equal");
```

You'll see a green squiggle under the second `Console`—it's an **`Unreachable code detected`** warning. The C# compiler knows that 0.1 + 0.2 is always equal to 0.3, so the code will never reach the `else` part of the statement. Run the code—it prints `They're equal` to the console.

Next, **change the float literals to doubles** (remember, literals like 0.1 default to double):

```
if (0.1 + 0.2 == 0.3) Console.WriteLine("They're equal");
else Console.WriteLine("They aren't equal");
```

That's really strange. The warning moved to the first line of the `if` statement. Try running the program. Hold on, that can't be right! It printed `They aren't equal` to the console. How is 0.1 + 0.2 not equal to 0.3?

Now do one more thing. Change 0.3 to 0.30000000000000004 (with 15 zeros between the 3 and 4). Now it prints `They're equal` again. So apparently 0.1D plus 0.2D equals 0.30000000000000004D.

Wait, what?!

SO IS THAT WHY I SHOULD ONLY USE THE ***DECIMAL TYPE FOR MONEY***, AND NEVER USE DOUBLE?

Exactly. Decimal has a lot more precision than double or float, so it avoids the 0.30000000000000004 problem.

Some floating-point types—not just in C#, but in most programming languages!—can give you ***rare*** weird errors. This is so strange! How can 0.1 + 0.2 be 0.30000000000000004?

It turns out that there are some numbers that just can't be exactly represented as a double—it has to do with how they're stored as binary data (0s and 1s in memory). For example, .1D is not *exactly* .1. Try multiplying `.1D * .1D`—you get 0.010000000000000002, not 0.01. But `.1M * .1M` gives you the right answer. That's why floats and doubles are really useful for a lot of things (like positioning a GameObject in Unity). If you need more rigid precision—like for a financial app that deals with money—decimal is the way to go.

Q: I'm still not clear on the difference between conversion and casting. Can you explain it a little more clearly?

A: Conversion is a general, all-purpose term for converting data from one type to another. Casting is a much more specific operation, with explicit rules about which types can be cast to other types, and what to do when the data for the value from one doesn't quite match the type it's being cast to. You just saw an example of one of those rules—when a floating-point number is cast to an int, it's rounded down by dropping any decimal value. You saw another rule earlier about wrapping for integer types, where a number that's too big to fit into the type it's being cast to is wrapped using the remainder operator.

Q: Hold on a minute. Earlier you had me "wrap" numbers myself using the mod function on my calculator app. Now you're talking about remainders. What's the difference?

A: Mod and remainder are very similar operations. For positive numbers they're exactly the same: A % B is the remainder when you divide B into A, so: 5 % 2 is the remainder of 5 ÷ 2, or 1. (If you're trying to remember how long division works, that just means that 5 ÷ 2 is equal to 2 × 2 + 1, so the rounded quotient is 2 and the remainder is 1.) But when you start dealing with negative numbers, there's a difference between mod (or modulus) and remainder. You can see for yourself: your calculator will tell you that –397 mod 17 = 11, but if you use the C# remainder operator you'll get –397 % 17 = –6.

there are no Dumb Questions

Q: Owen's formula had me dividing two values and then rounding the result down to the nearest integer. How does that fit in with casting?

A: Let's say you have some floating-point values:

```
float f1 = 185.26F;
double d2 = .0000316D;
decimal m3 = 37.26M;
```

and you want to cast them to int values so you can assign them to int variables `i1`, `i2`, and `i3`. We know that those int variables can only hold integers, so your program needs to do *something* to the decimal part of the number.

So C# has a consistent rule: it drops the decimal and rounds down: `f1` becomes 185, `d2` becomes 0, and `m3` becomes 37. But don't take our word for it—write your own C# code that casts those three floating-point values to int to see what happens.

There's a whole web page dedicated to the 0.30000000000000004 problem! Check out https://0.30000000000000004.com to see examples in a lot of different languages.

The 0.1D + 0.2D != 0.3D example is an edge case, or a problem or situation that only happens under certain rare conditions, usually when a parameter is at one of its extremes (like a very big or very small number). If you want to learn more about it, there's a great article by Jon Skeet about how floating-point numbers are stored in memory in .NET. You can read it here: https://csharpindepth.com/Articles/FloatingPoint.

Jon gave us some amazing technical review feedback for the very first edition of this book, and that made a huge difference for us. Thanks so much, Jon!

Use reference variables to access your objects

When you create a new object, you use a `new` statement to instantiate it, like `new Guy()` in your program at the end of the last chapter—the `new` statement created a new Guy object on the heap. You still needed a way to *access* that object, and that's where a variable like `joe` came in: `Guy joe = new Guy()`. Let's dig a little deeper into exactly what's going on there.

The `new` statement creates the instance, but just creating that instance isn't enough. ***You need a reference to the object***. So you created a **reference variable**: a variable of type Guy with a name, like `joe`. So `joe` is a reference to the new Guy object you created. Any time you want to use that particular Guy, you can reference it with the reference variable called `joe`.

When you have a variable that's an object type, it's a reference variable: a reference to a particular object. Let's just make sure we get the terminology right since we'll be using it a lot. We'll use the first two lines of the "Joe and Bob" program from the last chapter:

Here's the heap before your code runs. Nothing there.

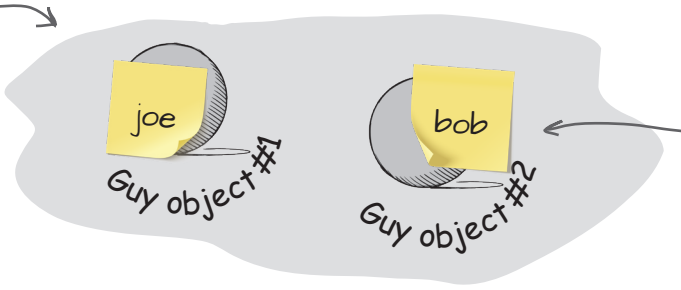
Creating a reference is like writing a name on a sticky note and sticking it to the object. You're using it to label an object so you can refer to it later.

```
Guy joe = new Guy() { Cash = 50, Name = "Joe" };
Guy bob = new Guy() { Cash = 100, Name = "Bob" };
```

This is the reference variable.

This creates the object that it will refer to.

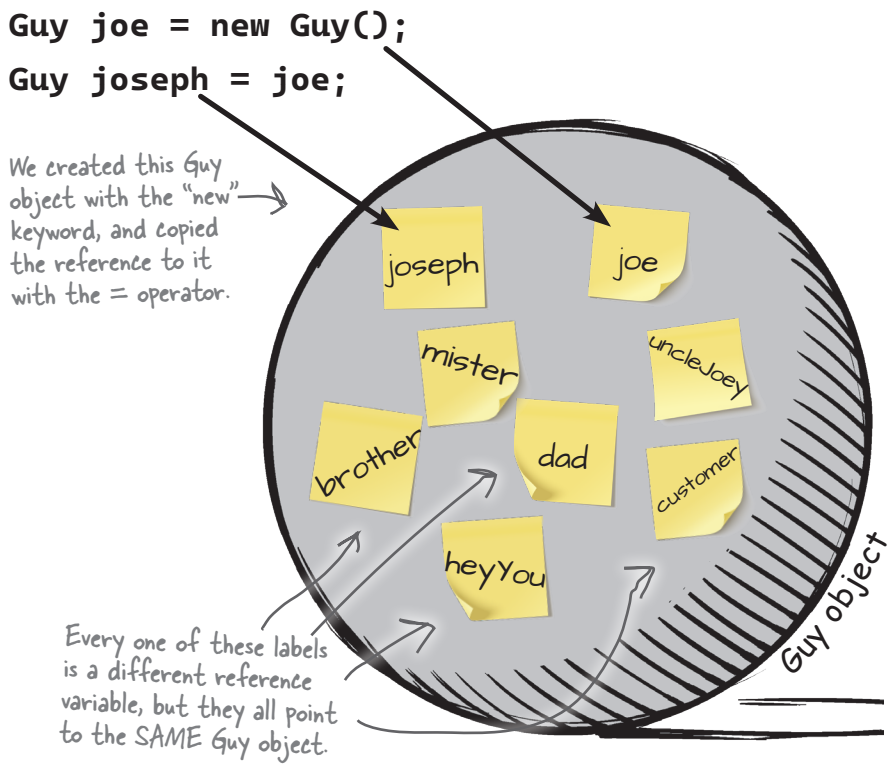
And here's the heap after this code runs. It has two objects, with the variable "joe" referring to one object and the variable "bob" referring to the other one.



The ONLY way to reference this Guy object is through the reference variable called "bob".

References are like sticky notes for your objects

In your kitchen, you probably have containers of salt and sugar. If you switched their labels, it would make for a pretty disgusting meal—even though you changed the labels, the contents of the containers stayed the same. ***References are like labels.*** You can move labels around and point them at different things, but it's the **object** that dictates what methods and data are available, not the reference itself—and you can **copy references** just like you copy values.



A reference is like a label that your code uses to talk about a specific object. You use it to access fields and call methods on an object that it points to.

We stuck a lot of sticky notes on that object! In this particular case, there are a lot of different references to this same Guy object—because a lot of different methods use it for different things. Each reference has a different name that makes sense in its context.

That's why it can be really useful to have ***multiple references pointing to the same instance***. So you could say `Guy dad = joe`, and then call `dad.GiveCash()` (that's what Joe's kid does every day). If you want to write code that works with an object, you need a reference to that object. If you don't have that reference, you have no way to access the object.

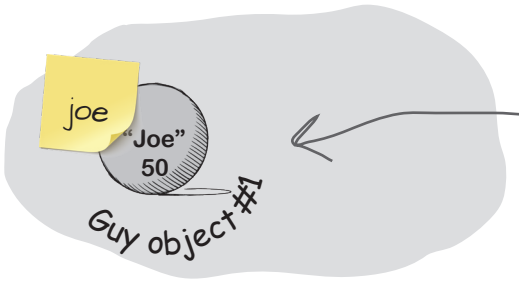
If there aren't any more references, your object gets garbage-collected

If all of the labels come off of an object, programs can no longer access that object. That means C# can mark the object for **garbage collection**. That's when C# gets rid of any unreferenced objects and reclaims the memory those objects took up for your program's use.

1 **Here's some code that creates an object.**

Just to recap what we've been talking about: when you use the `new` statement, you're telling C# to create an object. When you take a reference variable like `joe` and assign it to that object, it's like you're slapping a new sticky note on it.

```
Guy joe = new Guy() { Cash = 50, Name = "Joe" };
```

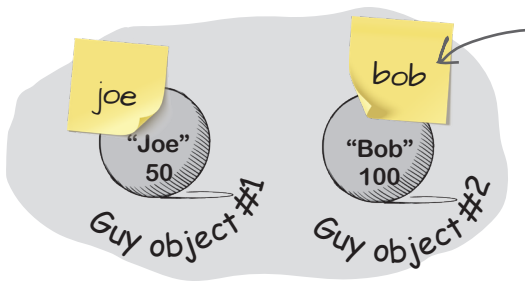


We used an object initializer to create this Guy object. Its Name field has the string "Joe", its Cash field has the int 50, and we put a reference to the object in a variable called "joe".

2 **Now let's create our second object.**

Once we do this we'll have two Guy object instances and two reference variables: one variable (`joe`) for the first Guy object, and another variable (`bob`) for the second.

```
Guy bob = new Guy() { Cash = 100, Name = "Bob" };
```



We created another Guy object and created a variable called "bob" that points to it. Variables are like sticky notes—they're just labels that you can "stick" to any object.

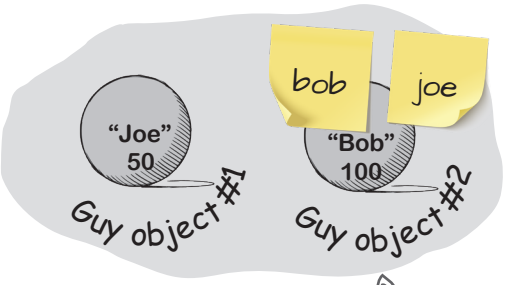
3 Let's take the reference to the first Guy object and change it to point to the second Guy object.

Take a really close look at what you're doing when you create a new Guy object. You're taking a variable and using the = assignment operator to set it—in this case, to a reference that's returned by the `new` statement. That assignment works because **you can copy a reference just like you copy a value**.

So let's go ahead and copy that value:

```
joe = bob;
```

That tells C# to take make `joe` point to the same object that `bob` does. Now the `joe` and `bob` variables **both point to the same object**.

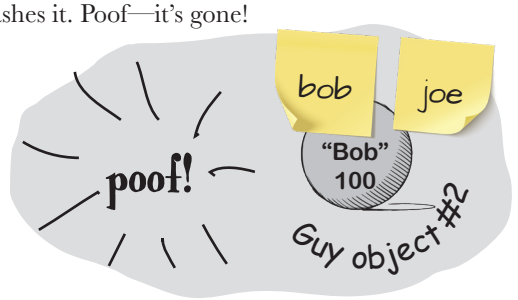


After the CLR (coming up in "Garbage Collection Exposed" interview!) removes the last reference to the object, it marks it for garbage collection.

4 There's no longer a reference to the first Guy object...so it gets garbage-collected.

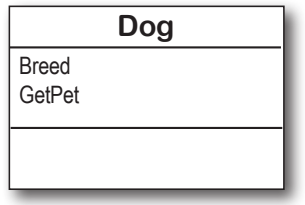
Now that `joe` is pointing to the same object as `bob`, there's no longer a reference to the Guy object it used to point to. So what happens? C# marks the object for garbage collection, and ***eventually*** trashes it. Poof—it's gone!

The CLR keeps track of all of the references to each object, and when the last reference disappears it marks it for removal. But it might have other things to do right now, so the object could stick around for a few milliseconds—or even longer!



For an object to stay in the heap, it has to be referenced. Some time after the last reference to the object disappears, so does the object.

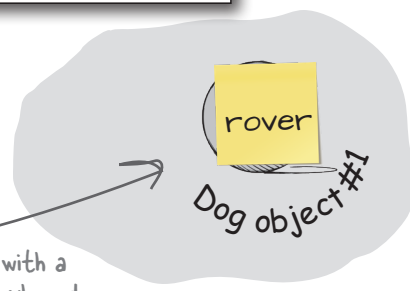
```
public partial class Dog {
  public void GetPet() {
    Console.WriteLine("Woof!");
  }
}
```



Multiple references and their side effects

You've got to be careful when you start moving reference variables around. Lots of times, it might seem like you're simply pointing a variable to a different object. You could end up removing all references to another object in the process. That's not a bad thing, but it may not be what you intended. Take a look:

1
```
Dog rover = new Dog();
rover.Breed = "Greyhound";
```

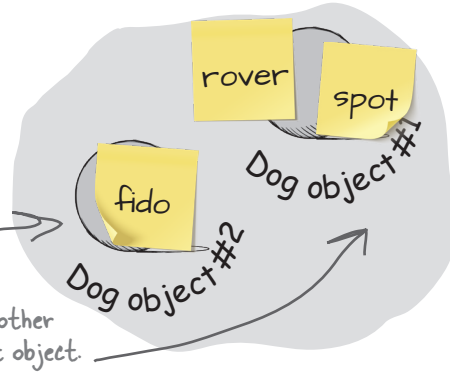


rover is a Dog object with a Breed field set to Greyhound.

Objects: 1

References: 1

2
```
Dog fido = new Dog();
fido.Breed = "Beagle";
Dog spot = rover;
```

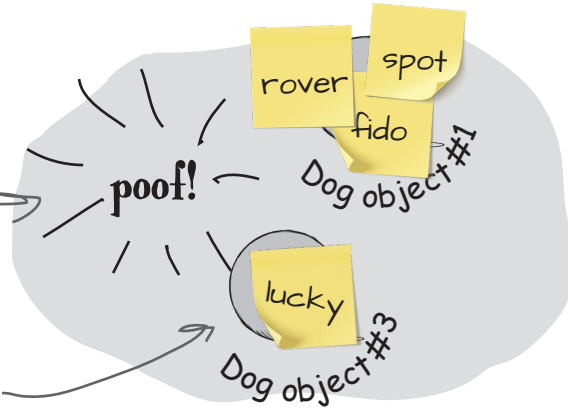


fido is another Dog object. spot is just another reference to the first object.

Objects: 2

References: 3

3
```
Dog lucky = new Dog();
lucky.Breed = "Dachshund";
fido = rover;
```



lucky is a third object. fido is now pointing to object #1. So, object #2 has no references. It's done as far as the program is concerned.

Objects: 2

References: 4

Sharpen your pencil

Now it's your turn. Here's one long block of code. Figure out how many objects and references there are at each stage. On the right-hand side, draw a picture of the objects and sticky notes in the heap.

1
```
Dog rover = new Dog();
rover.Breed = "Greyhound";
Dog rinTinTin = new Dog();
Dog fido = new Dog();
Dog greta = fido;
```

Objects:_____

References:_____

2
```
Dog spot = new Dog();
spot.Breed = "Dachshund";
spot = rover;
```

Objects:_____

References:_____

3
```
Dog lucky = new Dog();
lucky.Breed = "Beagle";
Dog charlie = fido;
fido = rover;
```

Objects:_____

References:_____

4
```
rinTinTin = lucky;
Dog laverne = new Dog();
laverne.Breed = "pug";
```

Objects:_____

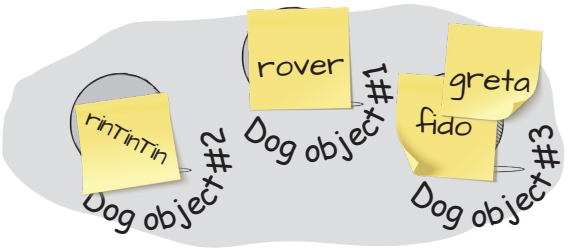
References:_____

5
```
charlie = laverne;
lucky = rinTinTin;
```

Objects:_____

References:_____

Sharpen your pencil Solution



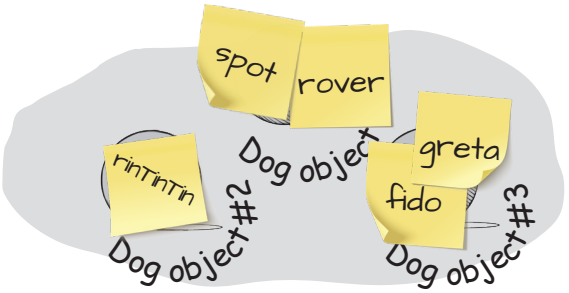
1

```
Dog rover = new Dog();
rover.Breed = "Greyhound";
Dog rinTinTin = new Dog();
Dog fido = new Dog();
Dog greta = fido;
```

Objects: 3

References: 4

One new Dog object is created, but spot is the only reference to it. When spot is set to rover, that object goes away.



2

```
Dog spot = new Dog();
spot.Breed = "Dachshund";
spot = rover;
```

Objects: 3

References: 5

3

```
Dog lucky = new Dog();
lucky.Breed = "Beagle";
Dog charlie = fido;
fido = rover;
```

Objects: 4

References: 7

charlie was set to fido when fido was still on object #3. Then, after that, fido moved to object #1, leaving charlie behind.

Dog #2 lost its last reference, and it went away.

4

```
rinTinTin = lucky;
Dog laverne = new Dog();
laverne.Breed = "pug";
```

Objects: 4

References: 8

When rinTinTin moved to lucky's object, the old rinTinTin object disappeared.

5

```
charlie = laverne;
lucky = rinTinTin;
```

Objects: 4

References: 8

Here the references move around, but no new objects are created. Setting lucky to rinTinTin did nothing because they already pointed to the same object.

Garbage Collection Exposed

This week's interview: The .NET Common Language Runtime

Head First: So, we understand that you do a pretty important job for us. Can you tell us a little more about what you do?

Common Language Runtime (CLR): In a lot of ways, it's pretty simple. I run your code. Any time you're using a .NET app, I'm making it work.

Head First: What do you mean by making it work?

CLR: I take care of the low-level "stuff" for you by doing a sort of "translation" between your program and the computer running it. When you talk about instantiating objects or doing garbage collection, I'm the one that's managing all of those things.

Head First: So how does that work, exactly?

CLR: Well, when you run a program on Windows, macOS, Linux, or most other operating systems, the OS loads machine language from a **binary**.

Head First: I'm going to stop you right there. Can you back up and tell us what machine language is?

CLR: Sure. A program written in machine language is made up of code that's executed directly by the CPU—and it's a whole lot less readable than C#.

Head First: If the CPU is executing the actual machine code, what does the OS do?

CLR: The OS makes sure each program gets its own process, respects the security rules, and provides APIs.

Head First: And for our readers who don't know what an API is?

CLR: An **API**—or **application programming interface**—is a set of methods provided by an OS, library, or program. OS APIs help you do things like work with the filesystem and interact with hardware. But they're often pretty difficult to use—especially for memory management—and they vary from OS to OS.

Head First: So back to your job. You mentioned a binary. What exactly is that?

CLR: A binary is a file that's (usually) created by a **compiler**, a program whose job it is to convert high-level language into low-level code like machine code. Windows binaries usually end with *.exe* or *.dll*.

Head First: But I'm guessing that there's a twist here. You said "low-level code like machine code"—does that mean there are other kinds of low-level code?

CLR: Exactly. I don't run the same machine language as the CPU. When you build your C# code, Visual Studio asks the C# compiler to create **Common Intermediate Language (CIL)**. That's what I run. C# code is turned into CIL, which I read and execute.

Head First: You talked about Windows binaries. But you also work on macOS. How does that work?

CLR: If you look in the folders created for your Visual Studio for Mac projects, you'll see lots of files that end with *.dll*. These are **managed .NET DLL files**, and they contain CIL code for the app. You can **run those apps from the command line** anywhere I'm installed! Try it out yourself. Open a console window, go to the folder with the PickRandomCards project from Chapter 3, find the folder under `bin/` that has files that end with *.dll*, and run this: `dotnet PickRandomCards.dll`

Head First: You mentioned managing memory. Is that where garbage collection fits into all of this?

CLR: Yes! One useful thing that I do for you is manage your computer's memory by figuring out when your app is done with certain objects. When it is, I get rid of them for you to free up that memory. That's something programmers used to have to do themselves—but thanks to me, it's something that you don't have to be bothered with. You might not have known it at the time, but I've been making your job of learning C# a whole lot easier.

You can run your console apps from the command line. Find the DLL file underneath the bin/ folder and run it like this: `dotnet ProjectName.dll` – and this will work on any OS you can install .NET on—even Linux!

Exercise

Create a program with an Elephant class. Instantiate two Elephant instances and then swap the reference values that point to them, ***without*** getting any Elephant instances garbage-collected. Here's what it will look like when your program runs.

You're going to build a new console app that has a class called Elephant.

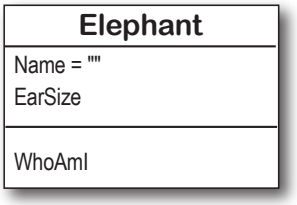
Here's an example of the output of the program:

```
Press 1 for Lloyd, 2 for Lucinda, 3 to swap
You pressed 1
Calling lloyd.WhoAmI()
My name is Lloyd.
My ears are 40 inches tall.

You pressed 2
Calling lucinda.WhoAmI()
My name is Lucinda.
My ears are 33 inches tall.

You pressed 3
References have been swapped

You pressed 1
Calling lloyd.WhoAmI()
My name is Lucinda.
My ears are 33 inches tall.

You pressed 2
Calling lucinda.WhoAmI()
My name is Lloyd.
My ears are 40 inches tall.

You pressed 3
References have been swapped

You pressed 1
Calling lloyd.WhoAmI()
My name is Lloyd.
My ears are 40 inches tall.

You pressed 2
Calling lucinda.WhoAmI()
My name is Lucinda.
My ears are 33 inches tall.
```

The Elephant class has a WhoAmI method that writes these two lines to the console to display the values in the Name and EarSize fields.

Swapping the references causes the lloyd variable to call the Lucinda object's method, and vice versa.

Here's the class diagram for the Elephant class you'll need to create.



Swapping them again returns things to the way they were when the program started.

When you create your Elephant class, declare your Name field like this:

```
public string Name = "";
```

This sets the Name field to an empty string. Why do you think we're asking you to do that?

The CLR garbage-collects any object with no references to it. So here's a hint for this exercise: if you want to pour a cup of coffee into another cup that's currently full of tea, you'll need a third glass to pour the tea into...

Exercise

Your job is to create a .NET Core console app with an Elephant class that matches the class diagram and uses its fields and methods to generate output that matches the example output.

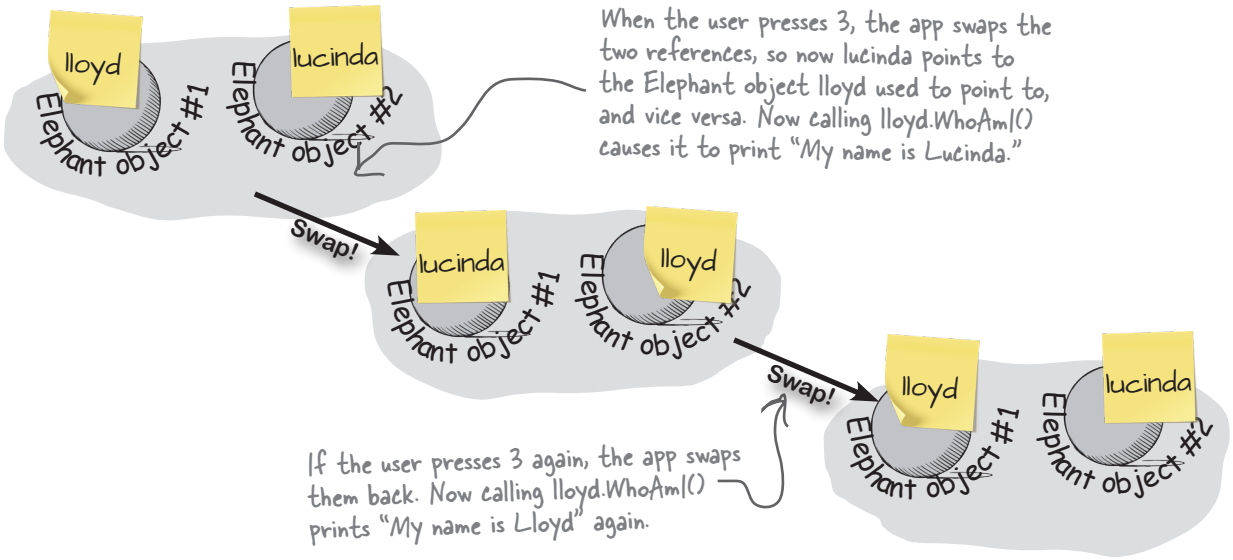
1. **Create a new .NET Core console app and add the Elephant class.**
Add an Elephant class to the project. Have a look at the Elephant class diagram—you'll need an int field called EarSize and a string field called Name. Add them, and make sure both are public. Then add a method called WhoAmI that writes two lines to the console to tell you the name and ear size of the elephant. Look at the example output to see exactly what it's supposed to print.

2. **Create two Elephant instances and a reference.**
Use object initializers to instantiate two Elephant objects:

```
Elephant lucinda = new Elephant() { Name = "Lucinda", EarSize = 33 };
Elephant lloyd = new Elephant() { Name = "Lloyd", EarSize = 40 };
```

3. **Call their WhoAmI methods.**
When the user presses 1, call lloyd.WhoAmI. When the user presses 2, call lucinda.WhoAmI. Make sure that the output matches the example.

4. **Now for the fun part: swap the references.**
Here's the interesting part of this exercise. When the user presses 3, make the app execute code that ***exchanges the two references***. You'll need to write that method. After you swap references, pressing 1 should write Lucinda's message to the console, and pressing 2 should write Lloyd's message. If you swap the references again, everything should go back to normal.

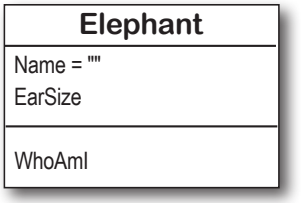


Exercise Solution

Create a program with an Elephant class. Instantiate two Elephant instances and then swap the reference values that point to them, ***without*** getting any Elephant instances garbage-collected.

Here's the Elephant class:

We asked you to initialize the Name field like this.




```
class Elephant
{
    public int EarSize;
    public string Name = "";
    public void WhoAmI()
    {
        Console.WriteLine("My name is " + Name + ".");
        Console.WriteLine("My ears are " + EarSize + " inches tall.");
    }
}
```

Here are the top-level statements for your Program.cs file:

```
Elephant lucinda = new Elephant() { Name = "Lucinda", EarSize = 33 };
Elephant lloyd = new Elephant() { Name = "Lloyd", EarSize = 40 };

Console.WriteLine("Press 1 for Lloyd, 2 for Lucinda, 3 to swap");
while (true)
{
    char input = Console.ReadKey(true).KeyChar;
    Console.WriteLine("You pressed " + input);
    if (input == '1')
    {
        Console.WriteLine("Calling lloyd.WhoAmI()");
        lloyd.WhoAmI();
    } else if (input == '2')
    {
        Console.WriteLine("Calling lucinda.WhoAmI()");
        lucinda.WhoAmI();
    } else if (input == '3')
    {
        Elephant holder;
        holder = lloyd;
        lloyd = lucinda;
        lucinda = holder;
        Console.WriteLine("References have been swapped");
    }
    else return;
    Console.WriteLine();
}
```

If you just point Lloyd to Lucinda, there won't be any more references pointing to Lloyd, and his object will be lost. That's why you need to have an extra variable (we called it "holder") to keep track of the Lloyd object reference until Lucinda can get there.

There's no "new" statement when we declare the "holder" variable because we don't want to create another instance of Elephant.

Two references mean TWO variables that can change the same object's data

Besides losing all the references to an object, when you have multiple references to an object, you can unintentionally change the object. In other words, one reference to an object may ***change*** that object, while another reference to that object has ***no idea*** that something has changed. Let's see how that works.

Add one more "else if" block to your top-level statements. Can you guess what will happen once it runs?

```
else if (input == '3')
{
    Elephant holder;
    holder = lloyd;
    lloyd = lucinda;
    lucinda = holder;
    Console.WriteLine("References have been swapped");
}
else if (input == '4')
{
    lloyd = lucinda;
    lloyd.EarSize = 4321;
    lloyd.WhoAmI();
}
else
{
    return;
}
```

After this statement, both the `lloyd` and `lucinda` variables reference the SAME Elephant object.

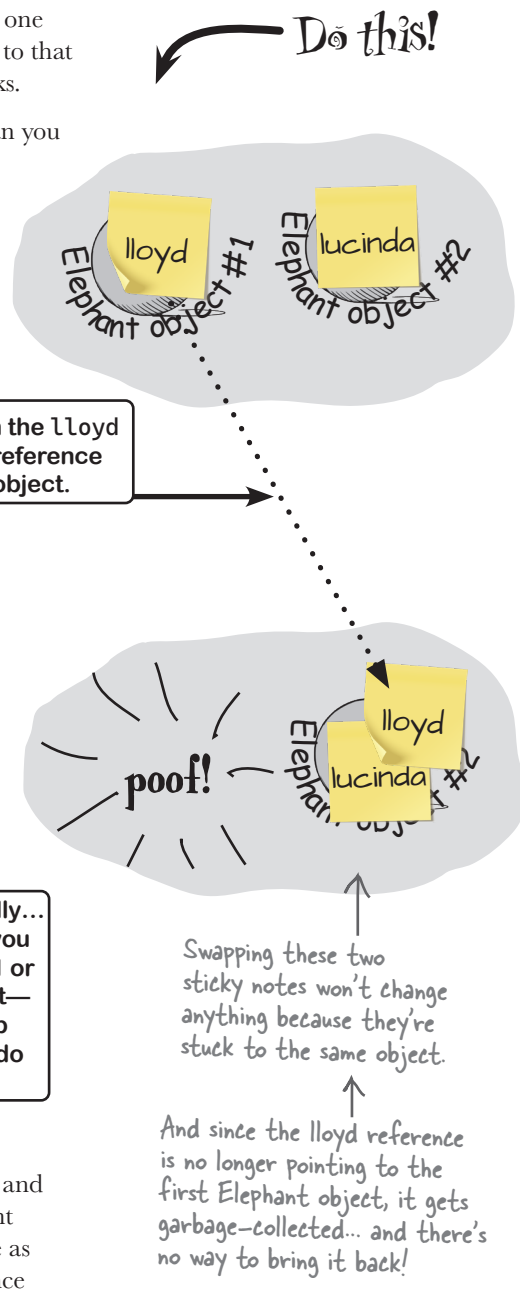
This statement says to set EarSize to 4321 on whatever object the reference stored in the `lloyd` variable happens to point to.

Now go ahead and run your program. Here's what you'll see:

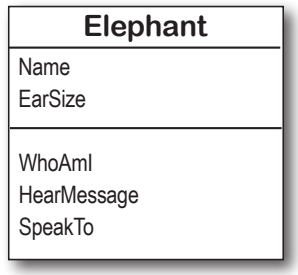
```
You pressed 4
My name is Lucinda
My ears are 4321 inches tall.

You pressed 1
Calling lloyd.WhoAmI()
My name is Lucinda
My ears are 4321 inches tall.

You pressed 2
Calling lucinda.WhoAmI()
My name is Lucinda
My ears are 4321 inches tall.
```

The program acts normally… until you press 4. Once you do that, pressing either 1 or 2 prints the same output—and pressing 3 to swap the references doesn't do anything anymore.

After you press 4 and run the new code that you added, both the **`lloyd`** and lucinda variables **contain the same reference** to the second Elephant object. Pressing 1 to call lloyd.WhoAmI prints exactly the same message as pressing 2 to call **`lucinda`**.WhoAmI. Swapping them makes no difference because you're swapping two identical references.



Objects use references to talk to each other



So far, you've seen forms talk to objects by using reference variables to call their methods and check their fields. Objects can call one another's methods using references, too. In fact, there's nothing that a form can do that your objects can't do, because your form is just another object. When objects talk to each other, one useful keyword that they have is **`this`**. Any time an object uses the **`this`** keyword, it's referring to itself—it's a reference that points to the object that calls it. Let's see what that looks like by modifying the Elephant class so instances can call each other's methods.

1 Add a method that lets an Elephant hear a message.

Let's add a method to the Elephant class. Its first parameter is a message from another Elephant object. Its second parameter is the Elephant object that sent the message:


```
public void HearMessage(string message, Elephant whoSaidIt) {
    Console.WriteLine(Name + " heard a message");
    Console.WriteLine(whoSaidIt.Name + " said this: " + message);
}
```

Here's what it looks like when it's called:

```
lloyd.HearMessage("Hi", lucinda);
```

We called **`lloyd's`** HearMessage method, and passed it two parameters: the string **`"Hi"`** and a reference to Lucinda's object. The method uses its **`whoSaidIt`** parameter to access the Name field of whatever elephant was passed in.

2 Add a method that lets an Elephant send a message.

Now let's add a SpeakTo method to the Elephant class. It uses a special keyword: **`this`**. That's a reference that lets an object get a reference to itself.

```
public void SpeakTo(Elephant whoToTalkTo, string message) {
    whoToTalkTo.HearMessage(message, this);
}
```

An Elephant's SpeakTo method uses the "this" keyword to send a reference to itself to another Elephant.

Let's take a closer look at what's going on.

When we call the Lucinda object's SpeakTo method:

```
lucinda.SpeakTo(lloyd, "Hi, Lloyd!");
```

It calls the Lloyd object's HearMessage method like this:

```
whoToTalkTo.HearMessage("Hi, Lloyd!", this);
```

Lucinda uses `whoToTalkTo` (which has a reference to Lloyd) to call `HearMessage`.

`this` is replaced with a reference to Lucinda's object.

```
[a reference to Lloyd].HearMessage("Hi, Lloyd!", [a reference to Lucinda]);
```

3 Call the new methods.

Add one more **`else if`** block to the top-level statements to make the Lucinda object send a message to the Lloyd object:

```
else if (input == '4')
{
    lloyd = lucinda;
    lloyd.EarSize = 4321;
    lloyd.WhoAmI();
}
else if (input == '5')
{
    lucinda.SpeakTo(lloyd, "Hi, Lloyd!");
}
else
{
    return;
}
```

The "this" keyword lets an object get a reference to itself.

Now run your program and press 5. You should see this output:

```
You pressed 5
Lloyd heard a message
Lucinda said this: Hi, Lloyd!
```

4 Use the debugger to understand what's going on.

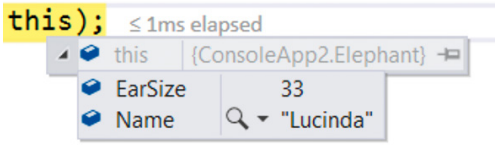
Place a breakpoint on the statement that you just added:

Run your program and press 5.

When it hits the breakpoint, use Debug >> Step Into (F11) to step into the SpeakTo method.

Add a watch for Name to show you which Elephant object you're inside. You're currently inside the Lucinda object—which makes sense because the the app just called lucinda.SpeakTo.

Hover over the **`this`** keyword at the end of the line and expand it. It's a reference to the Lucinda object.



Hover over **`whoToTalkTo`** and expand it—it's a reference to the Lloyd object.

The SpeakTo method has one statement—it calls whoToTalkTo.HearMessage. Step into it.

You should now be inside the HearMessage method. Check your watch again—now the value of the Name field is "Lloyd"—the Lucinda object called the Lloyd object's HearMessage method.

Hover over **`whoSaidIt`** and expand it. It's a reference to the Lucinda object.

Finish stepping through the code. Take a few minutes to really understand what's going on.

Arrays hold multiple values

Strings and arrays are different from the other data types you've seen in this chapter because they're the only ones without a set size (think about that for a bit).

If you have to keep track of a lot of data of the same type, like a list of prices or a group of dogs, you can do it in an **array**. What makes an array special is that it's a **group of variables** that's treated as one object. An array gives you a way of storing and changing more than one piece of data without having to keep track of each variable individually. When you create an array, you declare it just like any other variable, with a name and a type—except **the type is followed by square brackets**:

We saw arrays of strings in Chapter 3. Now let's take a deeper dive into how arrays work.

```
bool[] myArray;
```

Use the `new` keyword to create an array. Let's create an array with 15 bool elements:

```
myArray = new bool[15];
```

You use the `new` keyword to create an array because it's an object—so an array variable is a kind of reference variable. In C#, arrays are zero-based, which means the first element has index 0.

Use square brackets to set one of the values in the array. This statement sets the value of the fifth element of `myArray` to `false` by using square brackets and specifying the **index** 4. It's the fifth one because the first is `myArray[0]`, the second is `myArray[1]`, etc.:

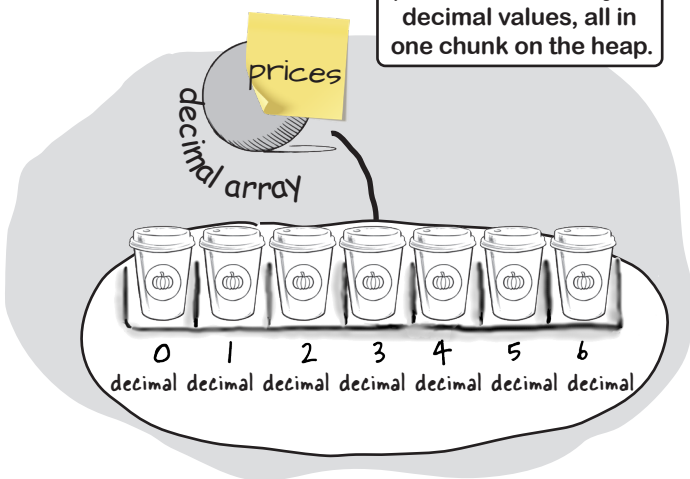
```
myArray[4] = false;
```

Use each element in an array like it's a normal variable

When you use an array, first you need to **declare a reference variable** that points to the array. Then you need to **create the array object** using the `new` statement, specifying how big you want the array to be. Then you can **set the elements** in the array. Here's an example of code that declares and fills up an array—and what's happening in the heap when you do it. The first element in the array has an **index** of 0.

The `prices` variable is a reference, just like any other object reference. The object it points to is an array of decimal values, all in one chunk on the heap.

```
// declare a new 7-element decimal array
decimal[] prices = new decimal[7];
prices[0] = 12.37M;
prices[1] = 6_193.70M;

// we didn't set the element
// at index 2, it remains
// the default value of 0

prices[3] = 1193.60M;
prices[4] = 58_000_000_000M;
prices[5] = 72.19M;
prices[6] = 74.8M;
```



Arrays can contain reference variables

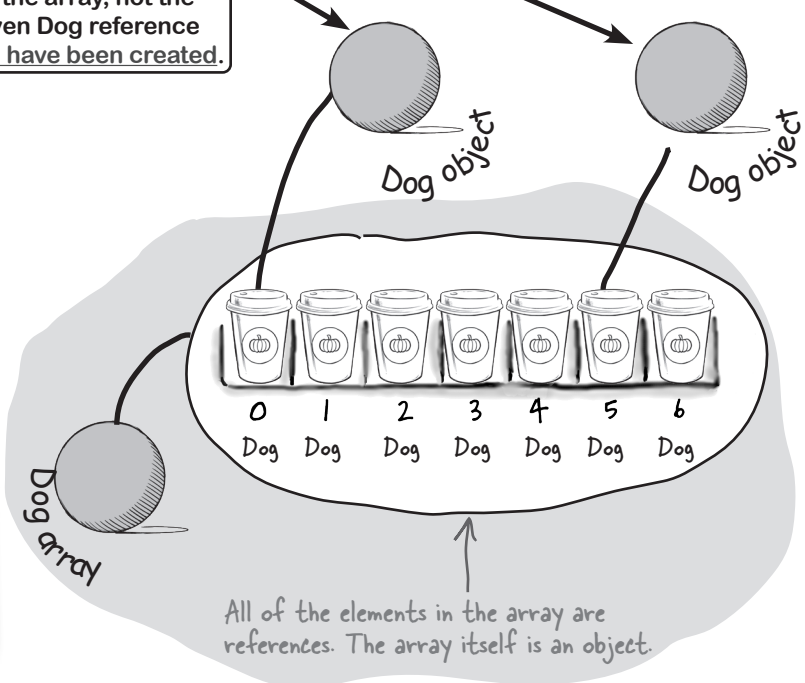
You can create an **array of object references** just like you create an array of numbers or strings. Arrays don't care what type of variable they store; it's up to you. So you can have an array of ints, or an array of Duck objects, with no problem.

Here's code that creates an array of seven Dog variables. The line that initializes the array only creates reference variables. Since there are only two `new Dog()` lines, only two actual instances of the Dog class are created.

```
// Declare a variable that holds an
// array of references to Dog objects
Dog[] dogs = new Dog[7];

// Create two new instances of Dog
// and put them at indexes 0 and 5
dogs[5] = new Dog();
dogs[0] = new Dog();
```

When you set or retrieve an element from an array, the number inside the brackets is called the index. The first element in the array has an index of **0**.

The first line of code only created the array, not the instances. The array is a list of seven Dog reference variables—but only two Dog objects have been created.



An array's length

You can find out how many elements are in an array using its Length property. So if you've got an array called "prices", then you can use prices.Length to find out how long it is. If there are seven elements in the array, that'll give you 7—which means the array elements are numbered 0 to 6.

there are no Dumb Questions

Q: I'm still not sure I get how references work.

A: References are the way you use all of the methods and fields in an object. If you create a reference to a Dog object, you can then use that reference to access any methods you've created for the Dog object. If the Dog class has (nonstatic) methods called Bark and Fetch, you can create a reference called `spot`, and then you can use that to call spot.Bark() or spot.Fetch(). You can also change information in the fields for the object using the reference (so you could change a Breed field using spot.Breed).

Q: Then doesn't that mean that every time I change a value through a reference I'm changing it for all of the other references to that object, too?

A: Yes. If the `rover` variable contains a reference to the same object as `spot`, changing rover.Breed to "beagle" would make it so that spot.Breed was "beagle".

Q: Remind me again—what does `this` do?

A: `this` is a special variable that you can only use inside an object. When you're inside a class, you use `this` to refer to any field or method of that particular instance. It's especially useful when you're working with a class whose methods call other classes. One object can use it to send a reference to itself to another object. So if `spot` calls one of `rover`'s methods passing `this` as a parameter, he's giving `rover` a reference to the `spot` object.

Any time you've got code in an object that's going to be instantiated, the instance can use the special this variable that has a reference to itself.

Q: You keep talking about garbage-collecting, but what's actually doing the collecting?

A: Every .NET app runs inside the **Common Language Runtime** (or the Mono Runtime if you're running your apps on macOS, Linux, or using Mono on Windows). The CLR does a lot of stuff, but there are two *really important things* the CLR does that we're concerned about right now. First, it **executes your code**—specifically, the output produced by the C# compiler. Second, it manages the memory that your program uses. That means it keeps track of all of your objects, figures out when the last reference to an object disappears, and frees up the memory that it was using. The .NET team at Microsoft and the Mono team at Xamarin (which was a separate company for many years, but is now a part of Microsoft) have done an enormous amount of work making sure that it's fast and efficient.

Q: I still don't get that stuff about different types holding different-sized values. Can you go over that one more time?

A: Sure. The thing about variables is they assign a size to your number no matter how big its value is. So if you name a variable and give it a long type even though the number is really small (like, say, 5), the CLR sets aside enough memory for it to get really big. When you think about it, that's really useful. After all, they're called variables because they change all the time.

The CLR assumes you know what you're doing and you're not going to give a variable a type bigger than it needs. So even though the number might not be big now, there's a chance that after some math happens, it'll change. The CLR gives it enough memory to handle the largest value that type can accommodate.

Sharpen your pencil

Here's an array of Elephant objects and a loop that will go through it and find the one with the biggest ears. What's the value of biggestEars.EarSize **after** each iteration of the `for` loop?

We're creating an array of seven Elephant references.

```
Elephant[] elephants = new Elephant[7];
elephants[0] = new Elephant() { Name = "Lloyd", EarSize = 40 };
elephants[1] = new Elephant() { Name = "Lucinda", EarSize = 33 };
elephants[2] = new Elephant() { Name = "Larry", EarSize = 42 };
elephants[3] = new Elephant() { Name = "Lucille", EarSize = 32 };
elephants[4] = new Elephant() { Name = "Lars", EarSize = 44 };
elephants[5] = new Elephant() { Name = "Linda", EarSize = 37 };
elephants[6] = new Elephant() { Name = "Humphrey", EarSize = 45 };
```

Arrays start with index 0, so the first Elephant in the array is elephants[0].

```
Elephant biggestEars = elephants[0];
for (int i = 1; i < elephants.Length; i++)
{
    Console.WriteLine("Iteration #" + i);

    if (elephants[i].EarSize > biggestEars.EarSize)
    {
        biggestEars = elephants[i];
    }

    Console.WriteLine(biggestEars.EarSize.ToString());
}
```

This sets the biggestEars reference to the object that elephants[i] points to.

Be careful—this loop starts with the second element of the array (at index 1) and iterates six times until "i" is equal to the length of the array.

Fill in these values.

Iteration #1 biggestEars.EarSize = _____

Iteration #2 biggestEars.EarSize = _____

Iteration #3 biggestEars.EarSize = _____

Iteration #4 biggestEars.EarSize = _____

Iteration #5 biggestEars.EarSize = _____

Iteration #6 biggestEars.EarSize = _____

Sharpen your pencil Solution

Here's an array of Elephant objects and a loop that will go through it and find the one with the biggest ears. What's the value of biggestEars.EarSize **after** each iteration of the `for` loop?

The for loop starts with the second Elephant and compares it to whatever Elephant biggestEars points to. If its ears are bigger, it points biggestEars at that Elephant instead. Then it moves to the next one, then the next one...by the end of the loop, biggestEars points to the one with the biggest ears.

```
Elephant[] elephants = new Elephant[7];
elephants[0] = new Elephant() { Name = "Lloyd", EarSize = 40 };
elephants[1] = new Elephant() { Name = "Lucinda", EarSize = 33 };
elephants[2] = new Elephant() { Name = "Larry", EarSize = 42 };
elephants[3] = new Elephant() { Name = "Lucille", EarSize = 32 };
elephants[4] = new Elephant() { Name = "Lars", EarSize = 44 };
elephants[5] = new Elephant() { Name = "Linda", EarSize = 37 };
elephants[6] = new Elephant() { Name = "Humphrey", EarSize = 45 };

Elephant biggestEars = elephants[0];
for (int i = 1; i < elephants.Length; i++)
{
    Console.WriteLine("Iteration #" + i);

    if (elephants[i].EarSize > biggestEars.EarSize)
    {
        biggestEars = elephants[i];
    }

    Console.WriteLine(biggestEars.EarSize.ToString());
}
```

Did you remember that the loop starts with the second element of the array? Why do you think that is?

The biggestEars reference keeps track of which Elephant we've seen so far has the biggest ears. Use the debugger to check this! Put your breakpoint here and watch biggestEars.EarSize.

Iteration #1 biggestEars.EarSize = 40

Iteration #2 biggestEars.EarSize = 42

Iteration #3 biggestEars.EarSize = 42

Iteration #4 biggestEars.EarSize = 44

Iteration #5 biggestEars.EarSize = 44

Iteration #6 biggestEars.EarSize = 45

null means a reference points to nothing

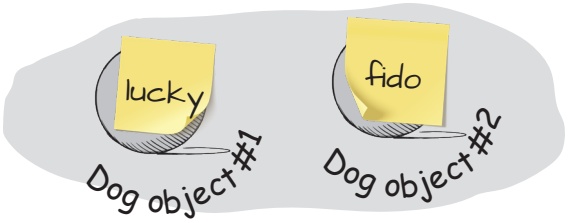
There's another important keyword that you'll use with objects. When you create a new reference and don't set it to anything, it has a value. It starts off set to `null`, which means **it's not pointing to any object at all**. Let's have a closer look at this:

The default value for any reference variable is `null`. Since we haven't assigned a value to `fido`, it's set to `null`.

```
Dog fido;
Dog lucky = new Dog();
```

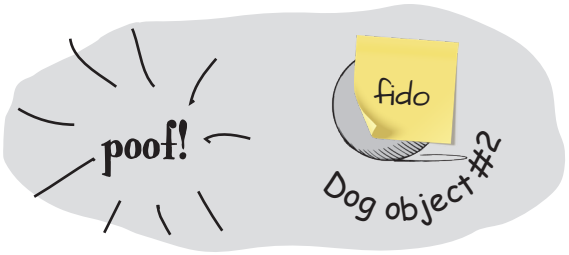
Now `fido` is set to a reference to another object, so it's not equal to `null` anymore.

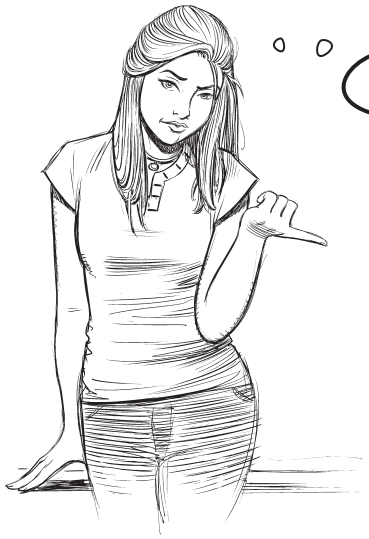
```
fido = new Dog();
```



Once we set `lucky` to `null` it no longer points to its object, so it gets marked for garbage collection.

```
lucky = null;
```





WOULD I EVER *REALLY* USE NULL IN A PROGRAM?

Yes. The null keyword can be useful.

There are a few ways you see `null` used in typical programs. The most common way is making sure a reference points to an object:

```
if (lloyd == null) {
```

That test will return `true` if the `lloyd` reference is set to `null`.

Another way you'll see the `null` keyword used is when you want your object to get garbage-collected. If you've got a reference to an object and you're finished with the object, setting the reference to `null` will immediately mark it for collection (unless there's another reference to it somewhere).

I THINK I'VE SEEN THE WORD **NULL** BEFORE. DIDN'T THE IDE TELL ME SOMETHING ABOUT IT WHEN I WAS USING CONSOLE.READLINE?

Yes! Console.ReadLine can return a null value.

Back at the beginning of Chapter 3, you hovered over Console.ReadLine so you could learn more about it from the description that the IntelliSense quick info pop-up. Let's take another look at it:

```
string? line = Console.ReadLine();
```

```
string? Console.ReadLine()
Reads the next line of characters from the standard input stream.

Returns:
  The next line of characters from the input stream, or null if no more lines are available.

Exceptions:
  IOException
  OutOfMemoryException
  ArgumentOutOfRangeException

GitHub Examples and Documentation
```

Console.ReadLine will return the next line of characters that it reads. If there are no more lines, then it returns null.

Console.ReadLine returns a null when there are no lines available

You've been running your apps in Visual Studio and typing input using the keyboard. But you can also run them from the command line. In Windows, there's an executable in the bin\Debug folder. And on Windows or Mac, you can use this command to run your app from the project folder:

```
C:\Users\Public\source\repos\ConsoleApp1\ConsoleApp1>dotnet run
Hello, World!
```

Make sure you run from inside the project folder that has the .csproj file, not the solution folder that contains it.

You can also use your operating system's the pipe commands like << or < or | to send input to your app from a file or the output of another console app. When you do this, Console.ReadLine needs a way to tell your app that it hit the end of the file—and that's when it returns null.

But there's stil one issue: ***what does your app do when Console.ReadLine returns null?***

Use the string? type when a string might be null

You've been using two different (but related!) types to hold text values. First, there's the **`string` type**, like you used for the Name field in the Elephant class:

```
public string Name = "";
```

Then there's the **`string?` type**, like the type returned by Console.ReadLine or which int.TryParse takes as its first parameter, like you used in Owen's ability score calculator app:

```
string? line = Console.ReadLine();
if (int.TryParse(line, out int value))
```

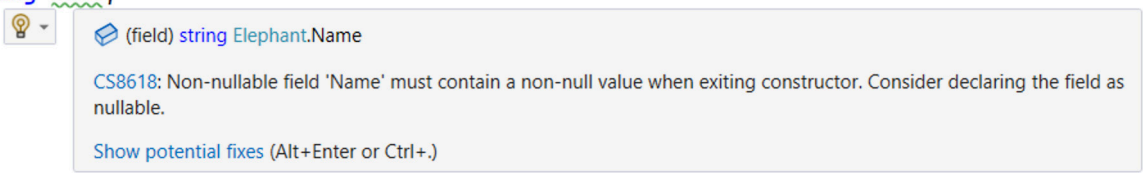
The difference is that in the Elephant class ***the Name field is never null***. That's why we asked you to initialize the Name field in your Elephant class.

What do you think would happen if you didn't initialize the Name field in the Elephant class?

Do this!

Change the field declaration in the Elephant class so it doesn't initialize it to an empty string:

```
public string Name;
```



Visual Studio gives you a warning that has to do with null values, and asks you to consider declaring the field as nullable. That's what the string? type is—a **nullable string**.

You can make the error disappear by changing the Name field to a nullable string? instead of a string:

```
public string? Name;
```

Now your app builds again, and runs exactly the same way as it did before.

Use the debugger to step through the app, and add a watch for the holder variable. Its value is null until it runs this statement: `holder = lloyd;`

int.TryParse takes a string? parameter

So what does your app do if Console.ReadLine returns null?

Luckily, int.TryParse also takes a string? value, so if your app gets to the end of the input and Console.ReadLine returns null, int.TryParse will just return false—so the app will work just fine, and when it gets a null value it will treat it the way it treats any other value that can't be parsed.

Visual Studio is smart enough to check for possible places where a value can be null. You can avoid that problem by making sure all of your reference variables are initialized.

Tabletop Games

Game design... and beyond

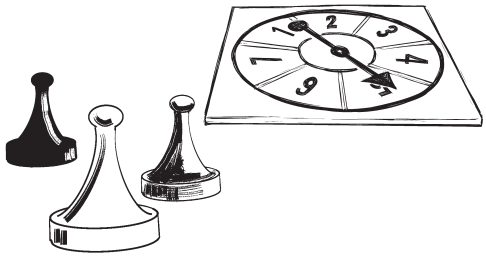
There's a rich history to tabletop games—and, as it turns out, a long history of tabletop games influencing video games, at least as early as the very first commercial role-playing game.

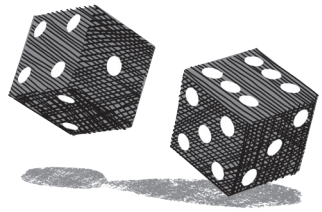
- The first edition of Dungeons and Dragons (D&D) was released in 1974, and that same year games with names like "dungeon" and "dnd" started popping up on university mainframe computers.
- You've used the Random class to create numbers. The idea of games based on random numbers has a long history—for example, tabletop games that use dice, cards, spinners, and other sources of randomness.
- We saw in the last chapter how a paper prototype can be a valuable first step in designing a video game. Paper prototypes have a strong resemblance to tabletop games. In fact, you can often turn the paper prototype of a video game into a playable tabletop game, and use it to test some game mechanics.
- You can use tabletop games—especially card games and board games—as learning tools to understand the more general concept of game mechanics. Dealing, shuffling, dice rolling, rules for moving pieces around the board, use of a sand timer, and rules for cooperative play are all examples of mechanics.
- The mechanics of Go Fish include dealing cards, asking another player for a card, saying "Go Fish" when asked for a card you don't have, determining the winner, etc. We're going to actually build a Go Fish game later in the book, so take a minute and read the rules here: https://en.wikipedia.org/wiki/Go_Fish#The_game.

If you've never played Go Fish, take a few minutes and read the rules. We'll use them later in the book!

Even if we're not writing code for video games, there's a lot we can learn from tabletop games.

A lot of our programs depend on **random numbers**. For example, you've already used the Random class to create random numbers for several of your apps. Most of us don't actually have a lot of real-world experience with genuine random numbers… except when we play games. Rolling dice, shuffling cards, spinning spinners, flipping coins…these are all great examples of **random number generators**. The Random class is .NET's random number generator—you'll use it in many of your programs, and your experience using random numbers when playing tabletop games will make it a lot easier for you to understand what it does.





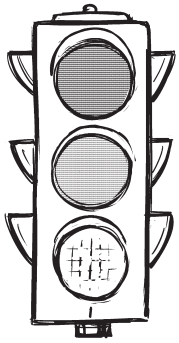
A Random Test Drive

You'll be using the **Random class** throughout the book, so let's get to know it better by kicking its tires and taking it for a spin. Fire up Visual Studio and follow along—and make sure you run your code multiple times, since you'll get different random numbers each time.

1. **Create a new console app**—all of this code will go in the top-level statements. Start by using Random.Shared to genrate a random int and write it to the console:

```
int randomInt = Random.Shared.Next();
Console.WriteLine(randomInt);
```

Specify a **maximum value** to get random numbers from 0 up to—but not including—the maximum value. A maximum of 10 generates random numbers from 0 to 9:



```
int zeroToNine = Random.Shared.Next(10);
Console.WriteLine(zeroToNine);
```

2. Now **simulate the roll of a die**. You can specify a minimum and maximum value. A minimum of 1 and maximum of 7 generates random numbers from 1 to 6:

```
int dieRoll = Random.Shared.Next(1, 7);
Console.WriteLine(dieRoll);
```

3. The **NextDouble method** generates random double values. Hover over the method name to see a tooltip—it generates a floating-point number from 0.0 up to 1.0:

```
double randomDouble = Random.Shared.NextDouble();
```

```
double Random.NextDouble()
Returns a random floating-point number that is greater than or equal to 0.0, and less than 1.0.
```

You can use **multiply a random double** to generate much larger random numbers. So if you want a random double value from 1 to 100, multiply the random double by 100:

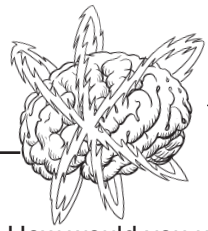
```
Console.WriteLine(randomDouble * 100);
```

Use **casting** to convert the random double to other types. Try running this code a bunch of times—you'll see tiny precision differences in the float and decimal values.

```
Console.WriteLine((float)randomDouble * 100F);
Console.WriteLine((decimal)randomDouble * 100M);
```

4. Use a maximum value of 2 to **simulate a coin toss**. That generates a random value of either 0 or 1. Use the **Convert class**, which has a static ToBoolean method that will convert it to a Boolean value:

```
int zeroOrOne = Random.Shared.Next(2);
bool coinFlip = Convert.ToBoolean(zeroOrOne);
Console.WriteLine(coinFlip);
```



Brain Power

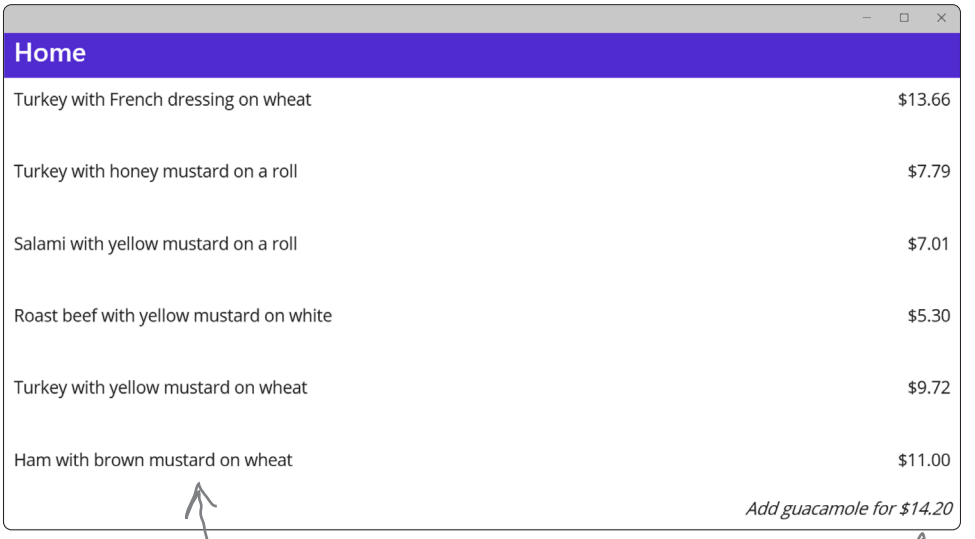
How would you use Random to choose a random string from an array of strings?

Welcome to Sloppy Joe's Budget House o' Discount Sandwiches!

Sloppy Joe has a pile of meat, a whole lotta bread, and more condiments than you can shake a stick at. What he doesn't have is a menu! Can you build a program that makes a new random menu for him every day? You definitely can…with a **new MAUI app**, some arrays, your handy random number generator, and a couple of new, useful tools. Let's get started!

Here's the app you'll build. It creates a menu with six random sandwiches. Each sandwich has a protein, a condiment, and a bread, all chosen at random from a list. Every sandwich is gets a random price, and there's a special random price at the bottom to add guacamole on the side.

Sloppy Joe needs a new menu every day. Your app will generate random sandwiches and prices for him.



Each sandwich is generated by choosing a random protein, random condiment, and random bread from arrays.

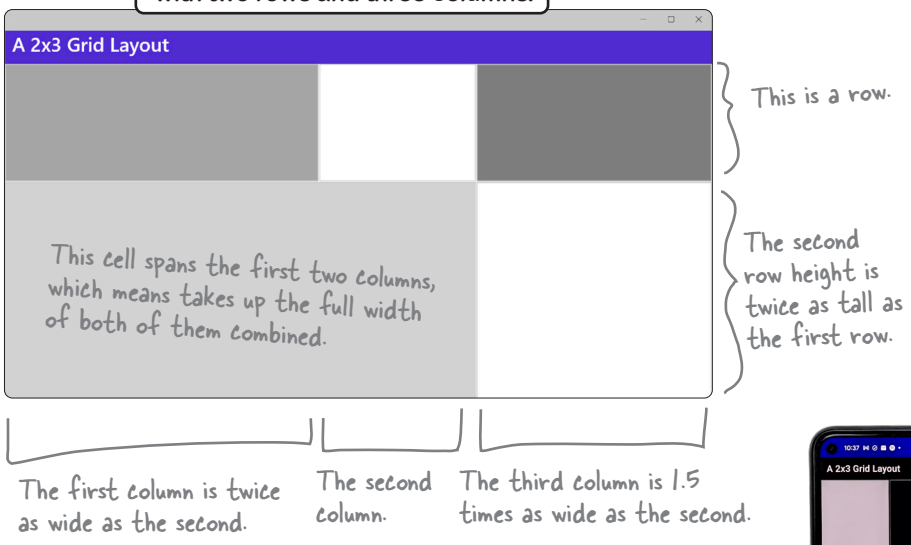
The prices are random numbers between 5.00 and 14.99.

Sloppy Joe's menu app uses a Grid layout

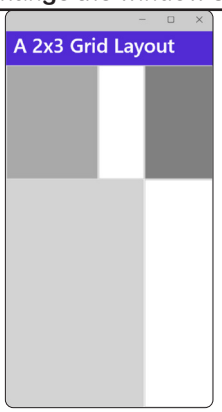
A **Grid control** contains other controls, and defines a set of rows and columns to lay out those controls.

You've used other layout controls: you've used VerticalStackLayout controls to stack Button, Label, and other controls in your apps on top of each other. You used a HorizontalStackLayout control in Chapter 2 for your bird picker. And in the Animal Matching Game project your VerticalStackLayout contained a FlexLayout that arranged the buttons so they stacked horizontally, flowing into rows if as the window size changed.

Here's an example of a Grid layout with two rows and three columns.



The layout adjusts as you change the window size.



The Grid preserves the row and column proportions when you change the size and shape of the page, which is really useful when you want your app to run on devices with different screen sizes.

A Grid control is for layouts, not data.

When most of us see something that contains "rows" and "columns" we think of tables of data, like spreadsheets or HTML tables. That's not what a Grid control is all about.

*The Grid control is for **laying out content**. Its job is to contain other controls, and give you a way to design more interesting or intricate layouts than you get with stack panels, in a way that works well with different window sizes or on mobile devices.*

Grid controls

The Grid control contains other controls, and works just like the other layout controls to contain **child controls** (the other controls nested inside it). There's an opening <Grid> tag and a closing </Grid> tag, and the tags for all of the child controls are between them.

Cells in a grid are invisible—their only purpose is to determine where the child controls are displayed on the page. We used **Border controls** to make the grid visible. A Border control draws a border around a child control nested inside it:

```
<Border>
    <Label Text="I have a border!"/>
</Border>
```

A Border can only contain one child control. In the app below we didn't nest any controls inside the Borders—we just took advantage of the fact that each Border fills up the entire cell. We used the Border control's BackgroundColor property to make some of the cells in the grid darker.

Use Grid properties to put a control in a cell

The rows sand columns in a Grid are numbered starting with 0. To put a child control in a specific row and column, use the Grid.Row and Grid.Column properties. For example, putting `<Border Grid.Row="1" Grid.Column="2" />` between Grid tags will make the Grid place the border in the second row and third column. You can also make a control span multiple rows or columns using the Grid.RowSpan and Grid.ColumnSpan properties.

Define the rows and columns for a Grid

The Grid control XAML has sections to define rows and columns. Each row or column can either have proportional sizes—for example, column 3 is twice as wide as column 2 and three times as wide as column 1—or absolute sizes in device-independent pixels.

The row and column definitions are in special sections inside the <Grid> tag. The row definitions are inside a <Grid.RowDefinitions> section, and the column definitions are inside a <Grid.ColumnDefinitions> section.

Here's the complete XAML for the app that we've been showing you. **Create a .NET MAUI app** called *GridExample* and add this XAML code (and delete the OnCounterClicked method in MainPage.xaml.cs).

Do this!

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="GridExample.MainPage">

    <ScrollView>

        <Grid>

            <Grid.RowDefinitions>
                <RowDefinition/>
                <RowDefinition Height="2*"/>
            </Grid.RowDefinitions>

            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="2*"/>
                <ColumnDefinition/>
                <ColumnDefinition Width="1.5*"/>
            </Grid.ColumnDefinitions>

            <Border BackgroundColor="DarkGray"/>
            <Border Grid.Column="1" />
            <Border Grid.Column="2" BackgroundColor="Gray"/>

            <Border Grid.Row="1" Grid.ColumnSpan="2" BackgroundColor="LightGray"/>
            <Border Grid.Row="1" Grid.Column="2" />

        </Grid>

    </ScrollView>

</ContentPage>
```

The app has two rows, so the Grid.RowDefinitions section contains two RowDefinition tags. The second row height is twice as tall as the first row, so we added the `Height="2*"` property to the second RowDefinition tag to make it twice as tall.

The Grid.ColumnDefinitions section has three ColumnDefinition tags, one for each of the three columns. The first column is twice as wide as the second, so it has `Width="2*"`. The third column is 1.5 times as wide, so it has `Width="1.5*"`.

Here are the child Border controls we just showed you.

Row heights and column widths

When you use a value like 2* in a RowDefinition.Height or ColumnDefinition.Width property, you're choosing a proportional width, which means they're proportional to each other. You'll get the same results setting the first row to 6* and the second row to 12* because the proportions are still the same: the second row is still twice as big as the first row.

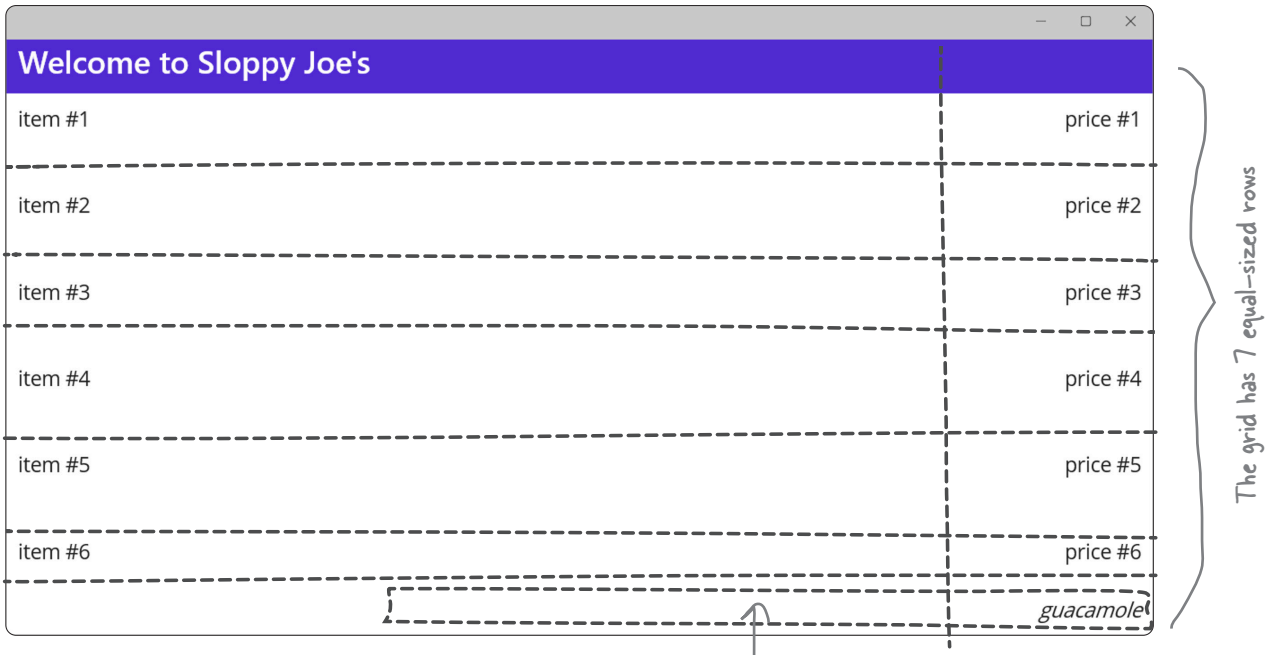
You can also set a row width or column height to an absolute value like 100, which will cause it to be sized in device-independent pixels. If all the rows and columns are proportional, the grid will fill up the page. If you set an absolute width or height, it could end up larger than the page, which is why it's a good idea to nest the Grid inside a ScrollView.

Create the Sloppy Joe's menu app and set up the grid

Create a new .NET MAUI app and name it SloppyJoe. The first thing you'll do is create the XAML for the app. Here's how it will work:

```
<ContentPage>
    <ScrollView>
        <Grid Margin="10">
```

The grid has two columns. Column 1 is 5 times wider than column 2.




```
        </Grid>
    </ScrollView>
</ContentPage>
```

Each of the cells in the grid contains a Label control...

...except for the Label with the guacamole price, which fills up the whole row by spanning two cells

We'll give you all of the XAML for the app. But before we do, try editing the MainPage.xaml file and creating the XAML for the page on your own. Can you use the app we just gave you as an example to create the row and column definitions yourself?

See how far you can get, then comapre it with our XAML.

Here's the XAML for the app.

Take your time and go through it line by line to make sure you understand how its grid works.

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="SloppyJoe.MainPage">

    <ScrollView>

        <Grid Margin="10">
            <Grid.RowDefinitions>
                <RowDefinition/>
                <RowDefinition/>
                <RowDefinition/>
                <RowDefinition/>
                <RowDefinition/>
                <RowDefinition/>
            </Grid.RowDefinitions>

            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="5*"/>
                <ColumnDefinition/>
            </Grid.ColumnDefinitions>

            <Label x:Name="item1" FontSize="18" Text="item #1" />
            <Label x:Name="price1" FontSize="18" HorizontalOptions="End"
                   Grid.Column="1" Text="price #1"/>

            <Label x:Name="item2" FontSize="18" Grid.Row="1" Text="item #2"/>
            <Label x:Name="price2" FontSize="18" HorizontalOptions="End"
                   Grid.Row="1" Grid.Column="1" Text="price #2"/>

            <Label x:Name="item3" FontSize="18" Grid.Row="2" Text="item #3" />
            <Label x:Name="price3" FontSize="18" HorizontalOptions="End"
                   Grid.Row="2" Grid.Column="1" Text="price #3"/>

            <Label x:Name="item4" FontSize="18" Grid.Row="3" Text="item #4" />
            <Label x:Name="price4" FontSize="18" HorizontalOptions="End"
                   Grid.Row="3" Grid.Column="1" Text="price #4"/>

            <Label x:Name="item5" FontSize="18" Grid.Row="4" Text="item #5" />
            <Label x:Name="price5" FontSize="18" HorizontalOptions="End"
                   Grid.Row="4" Grid.Column="1" Text="price #5"/>

            <Label x:Name="item6" FontSize="18" Grid.Row="5" Text="item #6" />
            <Label x:Name="price6" FontSize="18" HorizontalOptions="End"
                   Grid.Row="5" Grid.Column="1" Text="price #6"/>

            <Label x:Name="guacamole" FontSize="18" FontAttributes="Italic" Text="guacamole"
                   Grid.Row="6" Grid.ColumnSpan="2" HorizontalOptions="End" VerticalOptions="End" />

        </Grid>
    </ScrollView>

</ContentPage>
```

If you used a different app name, you'll see a different namespace here.

The 10 pixel margin around the grid adds a little space between the Labels and the edge of the window.

The grid has six rows that are all the same height.

After you add this code to your MainPage.xaml file, don't foget to go to the MainPage.xaml.cs file and delete the OnCounterClicked method and count field.

The grid has two columns. The first column is five times wider than the second.

Each of these Label controls goes in a different cell. We gave each of them text like "item #1" or "price #3" to make it easier to see how the grid is laid out when you run the app.

Each price has its HorizontalOptions set to "End" so it gets aligned all the way to the right of the window.

This Label spans both columns in the bottom row, so it stretches across two cells. Try removing the Grid.ColumnSpan property—what happens?

The Label with the guacamole price has both its Horizontal and Vertical options set to "End" to align it to the bottom right corner of the cell.

The C# code for the main page

Here's the C# code for the main page of your Sloppy Joe app. We're about to give you an exercise to build a class called MenuItem that generates random sandwiches and prices. As soon as the page loads, it calls a method called MakeTheMenu that uses an array of MenuItem objects to fill in all of the prices, and one last MenuItem object to get the price for the guacamole.

Your namespace will be different if you chose a different name for your app.

```
namespace SloppyJoe;

public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();
        MakeTheMenu();
    }

    private void MakeTheMenu()
    {
        MenuItem[] menuItems = new MenuItem[6];

        for (int i = 0; i < 6; i++)
        {
            menuItems[i] = new MenuItem();
            menuItems[i].Generate();
        }

        price1.Text = menuItems[0].Price;
        item1.Text = menuItems[0].Description;
        price2.Text = menuItems[1].Price;
        item2.Text = menuItems[1].Description;
        price3.Text = menuItems[2].Price;
        item3.Text = menuItems[2].Description;
        price4.Text = menuItems[3].Price;
        item4.Text = menuItems[3].Description;
        price5.Text = menuItems[4].Price;
        item5.Text = menuItems[4].Description;
        price6.Text = menuItems[5].Price;
        item6.Text = menuItems[5].Description;

        MenuItem guacamoleMenuItem = new MenuItem();
        guacamoleMenuItem.Generate();
        guacamole.Text = "Add guacamole for " + guacamoleMenuItem.Price;
    }
}
```

Call the MakeTheMenu method as soon as the page loads.

This array will hold 6 references to MenuItem objects.

You'll use a for loop to create each MenuItem and call its Generate method. You'll write that Generate method in the next exercise.

Now that you have the MenuItem objects with sandwich descriptions and prices, you can use them to set the text in the Label controls.

Create one more MenuItem object to generate a random price for the guacamole. You won't use its Description field.

Exercise

MenuItem
Proteins Condiments Breads Description Price
Generate

Create the MenuItem class for your menu app.

Start bhy looking closely at the class diagram. It has five fields: three arrays to hold the various sandwich parts, a description, and a price. The array fields use collection initializers, which let you define the items in an array by putting them inside curly braces.

Add the MenuItem class to your project. Here's the code for the fields:

```
class MenuItem
{
    public string[] Proteins = {
                "Roast beef", "Salami", "Turkey",
                "Ham", "Pastrami", "Tofu"
    };

    public string[] Condiments = {
                "yellow mustard", "brown mustard",
                "honey mustard", "mayo", "relish", "French dressing"
    };

    public string[] Breads = { "rye", "white", "wheat", "pumpernickel", "a roll" };

    public string Description = "";
    public string Price = "";

    public void Generate()
    {
        // You'll fill in this method
    }
}
```

The Generate method uses Random.Shared to choose random prices between 5.00 and 14.99 by creating a random decimal value out of two ints. We gave you the last line of code for the method:

```
Price = price.ToString("c");
```

The parameter to the ToString method is a format. In this case, the "c" format tells ToString to format the value with the local currency: if you're in the United States you'll see a $; in the UK you'll get a £, in the EU you'll see €, etc. If the values don't make sense in your currency, choose different random numbers!

Your job is to fill in the Generate method. It does the following:

- Picks a random protein from the Proteins array
- Picks a random condiment from the Condiments array
- Picks a random bread from the Breads array
- Sets the description field like this: `protein + " with " + condiment + " on " + bread`
- Sets the Price field to a random price that's at least 5.00 and less than 15.00. Pick a random int that's at least 5 and less than 15. Then pick a second random in that's at least 0 and less than 100. Multiply the second number by .01M to get a decimal value that's at least .00 and less than 1.00, and add it to the first value, and store it in a variable called `price`. Then set the Price field like this: `Price = price.ToString("c");`

Sharpen your pencil

Can you write a single line of code that sets Price to a random value between 5.00 and 14.99? Here's a hint: if the NextDouble method returns a value between 0 and 1. Try multiplying it by 10. What do you get?

Exercise Solution

```
public void Generate()
{
    string protein = Proteins[Random.Shared.Next(Proteins.Length)];
    string condiment = Condiments[Random.Shared.Next(Condiments.Length)];
    string bread = Breads[Random.Shared.Next(Breads.Length)];
    Description = protein + " with " + condiment + " on " + bread;

    int bucks = Random.Shared.Next(5, 15);
    int cents = Random.Shared.Next(0, 100);
    decimal price = bucks + (cents * .01M);
    Price = price.ToString("c");
}
```

Can you write a single line of code that sets Price to a random value between 5.00 and 14.99? Here's a hint: if the NextDouble method returns a value between 0 and 1. Try multiplying it by 10. What do you get?

```
Price = (Random.Shared.NextDouble() * 10 + 5).ToString("c");
```

You're right! This is a great time to improve accesibility.

Sloppy Joe has a wheelchair ramp and braille versions of all of his menus, because he wants to make sure everyone has a chance to eat his discount budget-friendly sandwiches. So let's make sure our menu app is accessible, too!

Start your operating system's screen reader and read the menu page.

Do this!

Windows Narrator

Start Windows Narrator (Ctrl+⊞+N). Narrator will scroll through the contents of any window when you hold down the Narrator key (insert) and press the left or right arrows. Navigate to your app, then navigate through all the controls and listen to what Narrator says.

macOS VoiceOver

Start VoiceOver (⌘+F5). VoiceOver will read the contents of any window when you hold down the VoiceOver activation key (^ control + ⌥ optoin) and pressing A. Navigate to your app and press VO+A (or ^ ⌥A), and listen to. Press the either ^ or ⌥ to stop reading.

Can we make the app more accessible?

When a screen reader narrates a window, it navigates from item to item, reading each item aloud and drawing a rectangle around it. What did you hear when you listened to the screen reader narrate your app? What did you see? Try having it read the menu while you have your eyes closed. Did you still understand everything that you needed to? It's pretty good! But accessibility is all about making things better for all of our users. Can we make it better?

Set the main header so the screen reader narrates it

You may have noted that the first thing it said was "Home" – and if you watched carefully, you saw that was narrating the title bar. **Modify AppShell.xaml to change "Home" to "Sloppy Joe's menu"** and have the screen reader narrate the page again. When you're

It would be great to have the narrator tell the user that they're looking at items on a menu. Let's try adding a SemanticProperties.Description to the <Grid> tag:

```
<Grid Margin="10"
      SemanticProperties.Description="Here are the items on the menu.">
```

Now try using the screen reader to narrate the window. It sounds fine in Windows, but if you're using macOS there's a problem: the screen reader won't read the items or prices. That's because if you set the SemanticProperties.Description on a that has children, the screen reader can't reach those children anymore. This is important even if you're building software for Windows, because your MAUI apps are cross-platform, and you want your app to be accessible anywhere.

Try setting the item1 label's SemanticProperties.Description instead

Ok, let's try something else. Remove the SemanticProperties.Description property from the <Grid> tag. Then try setting the SemanticProperties.Description on the first label:

```
<Label x:Name="item1" FontSize="18" Text="item #1"
       SemanticProperties.Description="Here are the items on the menu.">
```

Try using the screen reader again. ***It's still not right!*** When you have a Label, you always want the screen reader to read the contents of the label. Setting the SemanticProperties.Description causes the screen reader to read that description instead of the label text.

Go ahead and delete the SemanticDescription property from the item1 Label control (and also from the Grid, if you haven't done it already).

Brain Power

What do you think you'll do to make the screen reader say "Here are the items on the menu" followed by the randomly generated sandwich in the item1 Label control's Text property?

Use the SetValue method to change a control's semantic properties

Let's find a different way to make the screen reader say "Here are the items on the menu." before it reads the menu items. We'll still use the SemanticProperties.Description for the first menu item, but instead of using a XAML tag, we'll use C# to make sure it preserves the text.

Add this line of code to the end of your MainPage method:

If you type "item1." into Visual Studio, you won't see SemanticProperties in the IntelliSense pop-up. That's why you need to use the SetValue method to set it instead.

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

    item1.SetValue(SemanticProperties.DescriptionProperty,
        "Here are the items on the menu. " + item1.Text);
}
```

This code sets the SemanticProperties.Description property—in this case, it's setting it to the text "Here are the items on the menu" followed by the random sandwich generated by MenuItem. Try the screen reader one more time—now the page includes that text, and works on all operating systems.

Bullet Points

- The **new keyword** returns a reference to an object that you can store in a reference variable.
- You can have **multiple references** to the same object. You can change an object with one reference and access the results of that change with another.
- For an object to stay in the heap, it **has to be referenced**. Once the last reference to an object disappears, it eventually gets **garbage-collected** and the memory it used is reclaimed.
- Your .NET apps run in the **Common Language Runtime** (CLR), a "layer" between the OS and your program. The C# compiler builds your code into **Common Intermediate Language** (CIL), which the CLR executes.
- Declare **array variables** by putting square brackets after the type in the variable declaration (like bool[] trueFalseValues or Dog[] kennel).
- Use the **new keyword to create a new array**, specifying the array length in square brackets (like new bool[15] or new Dog[3]).The **this keyword** lets an object get a reference to itself.
- Use the **Length method** on an array to get its length (like kennel.Length).
- Access an array value using its **index** in square brackets (like bool[3] or Dog[0]). Array indexes **start at 0**.
- null means a reference **points to nothing**. The compiler will warn you when a variable can **potentially be null**.
- Use the **string? type** to hold a string that's allowed be null. Console.ReadLine can return null strings.
- Use **collection initializers** to initialize an array by setting the array equal to the new keyword followed by the array type followed by a comma-delimited list in curly braces (like new int[] { 8, 6, 7, 5, 3, 0, 9 }). The array type is optional when setting a variable or field value in the same statement where it's declared.
- You can pass a **format parameter** to an object or value's ToString method. If you're calling a numeric type's ToString method, passing it a value of "c" formats the value as a local currency.
- Use a control's **SetValue method** to set its semantic properties in code, so the screen reader can include text that's generated when the app runs.

Unity Lab #2

Write C# Code for Unity

Unity isn't *just* a powerful, cross-platform engine and editor for building 2D and 3D games and simulations. It's also a **great way to get practice writing C# code**.

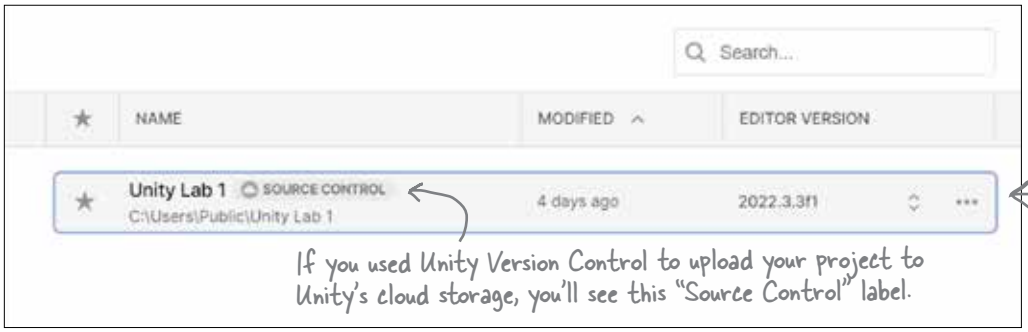
In the last Unity Lab, you learned how to navigate around Unity and your 3D space, and started to create and explore GameObjects. Now it's time to write some code to take control of your GameObjects. The whole goal of that lab was to get you oriented in the Unity editor (and give you an easy way to remind yourself of how to navigate around it if you need it).

In this Unity Lab, you'll start writing code to control your GameObjects. You'll write C# code to explore concepts you'll use in the rest of the Unity Labs, starting with adding a method that rotates the 8 Ball GameObject that you created in the last Unity Lab. You'll also start using the Visual Studio debugger with Unity to sleuth out problems in your games.

C# scripts add behavior to your GameObjects

Now that you can add a GameObject to your scene, you need a way to make it, well, do stuff. That's where your C# skills come in. Unity uses **C# scripts** to define the behavior of everything in the game.

This Unity Lab will introduce tools that you'll use to work with C# and Unity. You're going to build a simple "game" that's really just a little bit of visual eye candy: you'll make your 8 ball fly around the scene. Start by going to Unity Hub and **opening the same project** that you created in the first Unity Lab.



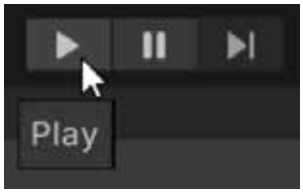
This Unity Lab picks up where the first one left off, so go to Unity Hub and open the project you created in the last lab.

Here's what you'll do in this Unity Lab:

1. **Attach a C# script to your GameObject.** You'll add a Script component to your Sphere GameObject. When you add it, Unity will create a class for you. You'll modify that class so that it drives the 8 ball sphere's behavior.

2. **Use Visual Studio to edit the script.** Remember how you set the Unity editor's preferences to make Visual Studio the script editor? That means you can just double-click on the script in the Unity editor and it will open up in Visual Studio.

3. **Play your game in Unity.** There's a Play button at the top of the screen. When you press it, it starts executing all of the scripts attached to the GameObjects in your scene. You'll use that button to run the script that you added to the sphere.



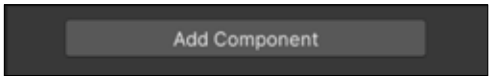
The Play button does not save your game! So make sure you save early and save often. A lot of people get in the habit of saving the scene every time they run the game.

4. **Use Unity and Visual Studio together to debug your script.** You've already seen how valuable the Visual Studio debugger is when you're trying to track down problems in your C# code. Unity and Visual Studio work together seamlessly so you can add breakpoints, use the Locals window, and work with the other familiar tools in the Visual Studio debugger while your game is running.

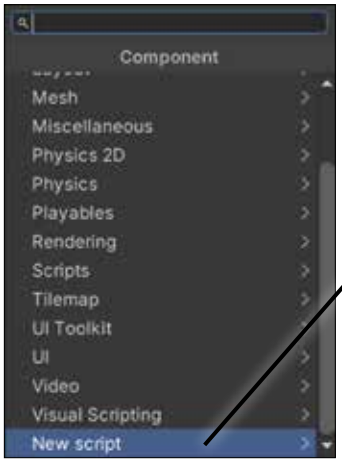
Add a C# script to your GameObject

Unity is more than an amazing platform for building 2D and 3D games. Many people use it for artistic work, data visualization, augmented reality, and more. It's especially valuable to you, as a C# learner, because you can write code to control everything that you see in a Unity game. That makes Unity ***a great tool for learning and exploring C#***.

Let's start using C# and Unity right now. Make sure the Sphere GameObject is selected, then **click the Add Component button** at the bottom of the Inspector window.



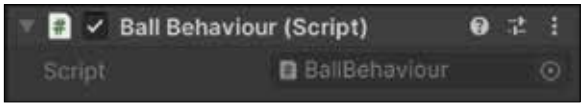
When you click it, Unity pops up a window with all of the different kinds of components that you can add—and there are ***a lot*** of them. **Choose "New script"** to add a new C# script to your Sphere GameObject. You'll be prompted for a name. **Name your script `BallBehaviour`**.



Unity code uses British spelling.

*If you're American (like us), or if you're used to the US spelling of the word **behavior**, you'll need to be careful when you work with Unity scripts because the class names often feature the British spelling **behaviour**.*

Click the "Create and Add" button to add the script. You'll see a component called *Ball Behaviour (Script)* appear in the Inspector window.



You'll also see the C# script in the Project window.

The Project window gives you a folder-based view of your project. Your Unity project is made up of files: media files, data files, C# scripts, textures, and more. Unity calls these files assets. The Project window was displaying a folder called Assets when you right-clicked inside it to import your texture, so Unity added it to that folder.

Did you notice a folder called Materials appeared in the Project window as soon as you dragged the 8 ball texture onto your sphere?

Write C# code to rotate your sphere

In the first lab, you told Unity to use Visual Studio as its external script editor. So go ahead and **double-click your new C# script in the Assets window**. When you do, ***Unity will open your script in Visual Studio***. Your C# script contains a class called BallBehaviour with two empty methods called Start and Update:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BallBehaviour : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

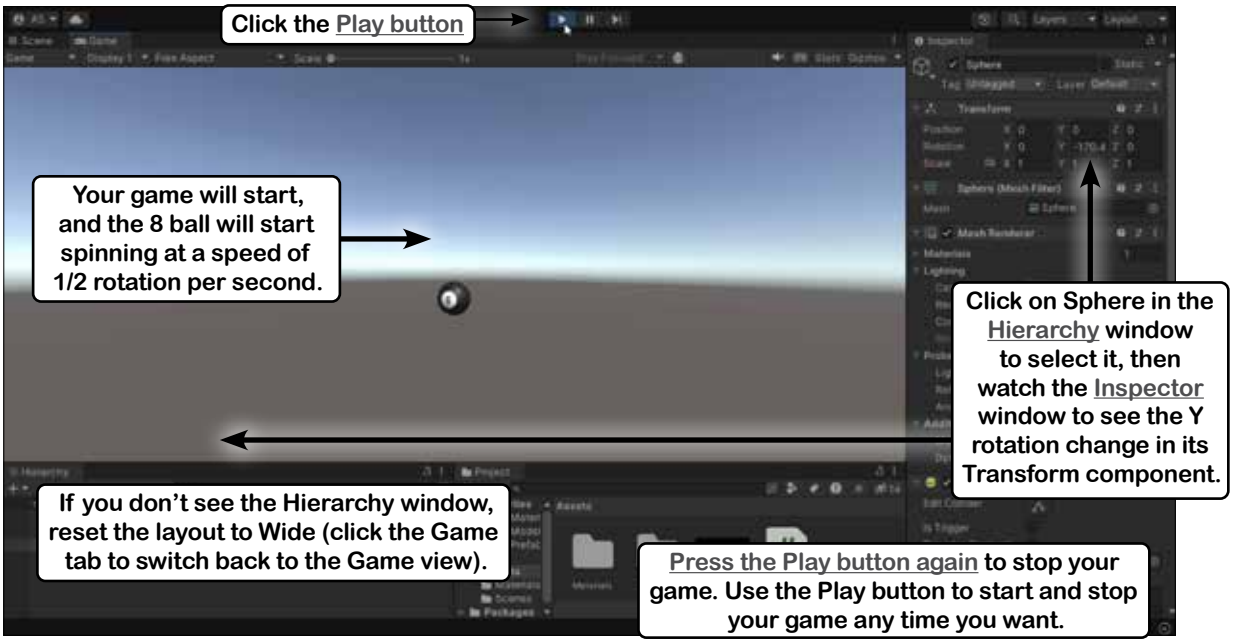
You opened your C# script in Visual Studio by clicking on it in the Hierarchy window, which shows you a list of every GameObject in the current scene. When Unity created your project, it added a scene called SampleScene with a camera and a light. You added a sphere to it, so your Hierarchy window will show all of those things.

If Unity didn't launch Visual Studio and open your C# script in it, go back to the beginning of Unity Lab 1 and make sure you followed the steps to set the External Tools preferences.

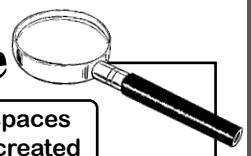
Here's a line of code that will rotate your sphere. **Add it to your Update method**:

```
transform.Rotate(Vector3.up, 180 * Time.deltaTime);
```

Now **go back to the Unity editor** and click the Play button in the toolbar to start your game:



Your Unity Code Up Close



```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BallBehaviour : MonoBehaviour
{
  // Start is called before the first frame update
  void Start()
  {

  }

  // Update is called once per frame
  void Update()
  {
     transform.Rotate(Vector3.up, 180 * Time.deltaTime);
  }
}
```

You learned about namespaces in Chapter 3. When Unity created the file with the C# script, it added `using` lines so it can use code in the UnityEngine namespace and two other common namespaces.

A frame is a fundamental concept of animation. Unity draws one still frame, then draws the next one very quickly, and your eye interprets changes in these frames as movement. Unity calls the Update method for every GameObject before each frame so it can move, rotate, or make any other changes that it needs to make. A faster computer will run at a higher frame rate—or number of frames per second (FPS)—than a slower one.

The transform.Rotate method causes a GameObject to rotate. The first parameter is the axis to rotate around. In this case, your code used Vector3.up, which tells it to rotate around the Y axis. The second parameter is the number of degrees to rotate.

Different computers will run your game at different frame rates. If it's running at 30 FPS, we want one rotation every 60 frames. If it's running at 120 FPS, it should rotate once every 240 frames. Your game's frame rate may even change if it needs to run more or less complex code.

That's where the Time.deltaTime value comes in handy. Every time the Unity engine calls a GameObject's Update method—once per frame—it sets Time.deltaTime to the fraction of a second since the last frame. Since we want our ball to do a full rotation every two seconds, or 180 degrees per second, all we need to do is multiply it by Time.deltaTime to make sure that it rotates exactly as much as it needs to for that frame.

Inside your Update method, multiplying any value by Time.deltaTime turns it into that value per second.

Time.deltaTime is static—and like we saw in Chapter 3, you don't need an instance of the Time class to use it.

Add a breakpoint and debug your game

Let's debug your Unity game. First **stop your game** if it's still running (by pressing the Play button again). Then switch over to Visual Studio, and **add a breakpoint** on the line that you added to the Update method.

```
13      // Update is called once per frame
        Unity Message | 0 references
14      void Update()
15      {
16          transform.Rotate(Vector3.up, 180 * Time.deltaTime);
17      }
```

Now find the button at the top of Visual Studio that starts the debugger:

- In Windows it looks like this 

—or choose Debug >> Start Debugging (F5) from the menu
- In macOS it looks like this 

—or choose Debug >> Start Debugging (⌘↵)

Click that button to **start the debugger**. Now switch back to the Unity editor. If this is the first time you're debugging this project, the Unity editor will pop up a dialog window with these buttons:

Press the "Enable debugging for this session" button (or if you want to keep that pop-up from appearing again, press "Enable debugging for all projects"). Visual Studio is now ***attached*** to Unity, which means it can debug your game.

Now **press the Play button in Unity** to start your game. Since Visual Studio is attached to Unity, it ***breaks immediately*** on the breakpoint that you added, just like with any other breakpoint you've set.

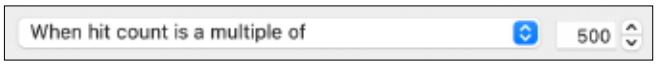
Congratulations, you're now debugging a game!

Use a hit count to skip frames

Sometimes it's useful to let your game run for a while before your breakpoint stops it. For example, you might want your game to spawn and move its enemies before your breakpoint hits. Let's tell your breakpoint to break every 500 frames. You can do that by adding a **Hit Count condition** to your breakpoint:

- On Windows, right-click on the breakpoint dot () at the left side of the line, choose **Conditions** from the pop-up menu, select *Hit Count* and *Is a multiple of* from the dropdowns, and enter 500 in the box:

- On macOS, right-click on the breakpoint dot (), choose **Edit breakpoint…** from the menu, then choose *When hit count is a multiple of* from the dropdown and enter 500 in the box:

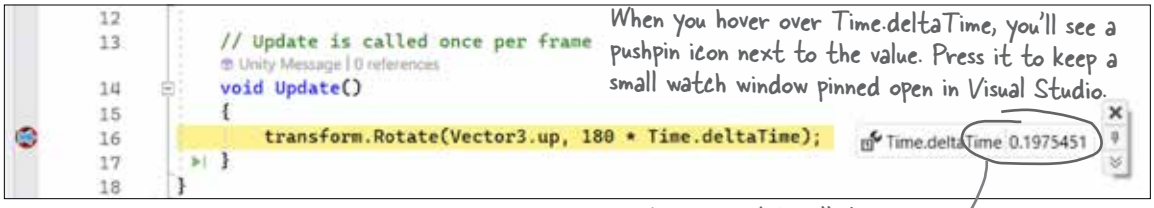


Now the breakpoint will only pause the game every 500 times the Update method is run—or every 500 frames. If your game is running at 60 FPS, then when you press Continue the game will run for a little over 8 seconds before it breaks again. **Press Continue, then switch back to Unity** and watch the ball spin until the breakpoint breaks.

Use the debugger to understand Time.deltaTime

You're going to be using Time.deltaTime in many of the Unity Labs projects. Let's take advantage of your breakpoint and use the debugger to really understand what's going on with this value.

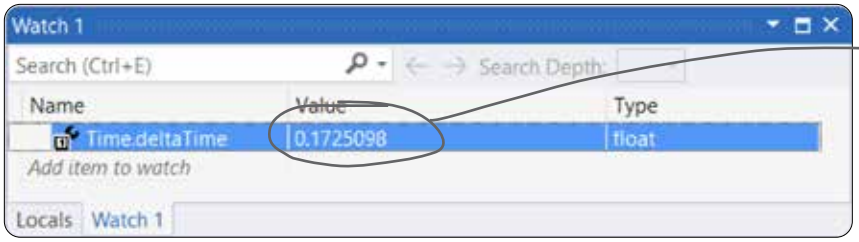
While your game is paused on the breakpoint in Visual Studio, **hover over Time.deltaTime** to see the fraction of a second that elapsed since the previous frame (you'll need to put your mouse cursor over `deltaTime`). Then **add a watch for Time.deltaTime** by selecting Time.deltaTime and choosing Add Watch from the right-mouse menu.



When you hover over Time.deltaTime, you'll see a pushpin icon next to the value. Press it to keep a small watch window pinned open in Visual Studio.

Every time the breakpoint pauses the game, your Time.deltaTime watch will show you the fraction of a second since the previous frame. Can you use this number to figure out the FPS we were getting when we took this screenshot?

Continue debugging (F5 on Windows, ⌘↵ on macOS), just like with the other apps you've debugged), to resume your game. The ball will start rotating again, and after another 500 frames the breakpoint will trigger again. You can keep running the game for 500 frames at a time. Keep your eye on the Time.deltaTime value each time it breaks, either in the pinned value or in the watch window.

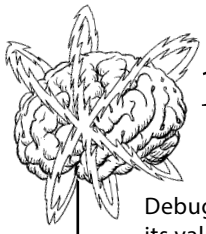


Press the Continue button to get another Time.deltaTime value, then another. You can get your approximate FPS by dividing 1 ÷ Time.deltaTime.

Stop debugging (Shift+F5 on Windows, ⇧⌘↵ on macOS) to stop your program. Then **start debugging again**.Since your game is still running, the breakpoint will continue to work when you reattach Visual Studio to Unity. Once you're done debugging, **toggle your breakpoint again** so the IDE will still keep track of it but not break when it's hit. **Stop debugging** one more time to detach from Unity.

Go back to Unity and **stop your game**—and ***save it***, because the Play button doesn't automatically save the game.

The Play button in Unity starts and stops your game. Visual Studio will stay attached to Unity even when the game is stopped.



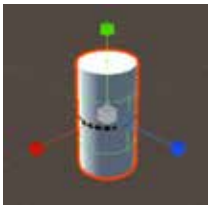
Brain Power

Debug your game again and hover over "Vector3.up" to inspect its value—you'll have to put your mouse cursor over up. It has a value of (0.0, 1.0, 0.0). What do you think that means?

Add a cylinder to show where the Y axis is

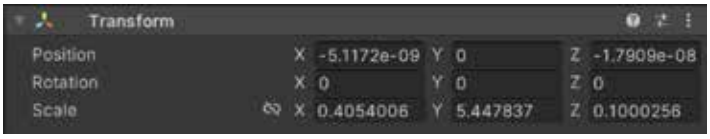
Your sphere is rotating around the Y axis at the very center of the scene. Let's add a very tall and very skinny cylinder to make it visible. ***Create a new cylinder*** by choosing *3D Object* >> *Cylinder* from the GameObject menu. Make sure it's selected in the Hierarchy window, then look at the Inspector window and check that Unity created it at position (0, 0, 0)—if not, use the context menu ( ) to reset it.

Let's make the cylinder tall and skinny. Choose the Scale tool from the toolbar: either click on it ( ) or press the R key. You should see the Scale Gizmo appear on your cylinder:



The Scale Gizmo looks a lot like the Move Gizmo, except that it has cubes instead of cones at the end of each axis. Your new cylinder is sitting on top of the sphere—you might see just a little of the sphere showing through the middle of the cylinder. When you make the cylinder narrower by changing its scale along the X and Z axes, the sphere will get uncovered.

Click and drag the green cube up to elongate your cylinder along the Y axis. Then click on the red cube and drag it toward the cylinder to make it very narrow along the X axis, and do the same with the blue cube to make it very narrow along the Z axis. Watch the Transform panel in the Inspector as you change the cylinder's scale—the Y scale will get larger, and the X and Z values will get much smaller.



You might notice the Position values change when you make the X and Z Scale values very small.

Click on the X label in the Scale row in the Transform panel and drag up and down. Make sure you click the actual X label to the left of the input box with the number. When you click the label it turns blue, and a blue box appears around the X value. As you drag your mouse up and down, the number in the box goes up and down, and the Scene view updates the scale in as you change it. Look closely as you drag—the scale can be positive and negative. Now **reset the Transform window**.

Now **select the number inside the X box and type .1**—the cylinder gets very skinny. Press Tab and type 20, then press Tab again and type .1, and press Enter.

When you edit the values in the Properties window, you can see the results update in the scene immediately.

Now your sphere has a very long cylinder going through it that shows the Y axis where Y = 0.

Add fields to your class for the rotation angle and speed

In Chapter 3 you learned how C# classes can have **fields** that store values methods can use. Let's modify your code to use fields. Add these four lines just under the class declaration, **immediately after the first curly brace {:**

```
public class BallBehaviour : MonoBehaviour
{
    public float XRotation = 0;
    public float YRotation = 1;
    public float ZRotation = 0;
    public float DegreesPerSecond = 180;
```

These are just like the fields that you added to the projects in Chapters 3 and 4. They're variables that keep track of their values—each time Update is called it reuses the same field over and over again.

The XRotation, YRotation, and ZRotation fields each contain a value between 0 and 1, which you'll combine to create a **vector** that determines the direction that the ball will rotate:

```
new Vector3(XRotation, YRotation, ZRotation)
```

The DegreesPerSecond field contains the number of degrees to rotate per second, which you'll multiply by Time.deltaTime just like before. **Modify your Update method to use the fields.** This new code creates a Vector3 variable called `axis` and passes it to the transform.Rotate method:

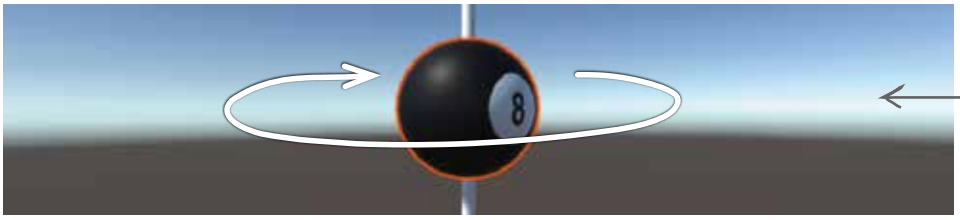
```
void Update()
{
    Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
    transform.Rotate(axis, DegreesPerSecond * Time.deltaTime);
}
```

Select the Sphere in the Hierarchy window. Your fields now show up in the Script component. When the Script component renders fields, it adds spaces between the capital letters to make them easier to read.

When you add public fields to a class in your Unity script, the Script component displays input boxes that let you modify those fields. If you modify them while the game is not running, the updated values will get saved with your scene. You can also modify them while the game is running, but they'll revert when you stop the game.

Run your game again. ***While it's running***, select the Sphere in the Hierarchy window and change the degrees per second to 360 or 90—the ball starts to spin at twice or half the speed. Stop your game—and the field will reset to 180.

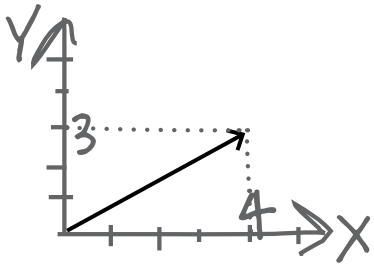
While the game is stopped, use the Unity editor to change the X Rotation to field to 1 and the Y Rotation field to 0. Start your game—the ball will rotate away from you. Click the X Rotation label and drag it up and down to change the value while the game is running. As soon as the number turns negative, the ball starts rotating toward you. Make it positive again and it starts rotating away from you.



When you use the Unity editor to set the Y Rotation field to 1 and then start your game, the ball rotates clockwise around the Y axis.

Use Debug.DrawRay to explore how 3D vectors work

A **vector** is a value with a **length** (or magnitude) and a **direction**. If you ever learned about vectors in a math class, you probably saw lots of diagrams like this one of a 2D vector:



Here's a diagram of a two-dimensional vector. You can represent it with two numbers: its value on the X axis (4) and its value on the Y axis (3), which you'd typically write as (4, 3).

That's not hard to understand…on an intellectual level. But even those of us who took a math class that covered vectors don't always have an ***intuitive*** grasp of how vectors work, especially in 3D. Here's another area where we can use C# and Unity as a tool for learning and exploration.

Use Unity to visualize vectors in 3D

You're going to add code to your game to help you really "get" how 3D vectors work. Start by having a closer look at the first line of your Update method:

```
Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
```

What does this line tell us about the vector?

- **It has a type: Vector3.** Every variable declaration starts with a type. Instead of using string, int, or bool, you're declaring it with the type Vector3. This is a type that Unity uses for 3D vectors.
- **It has a variable name: `axis`.**
- **It uses the `new` keyword to create a Vector3.** It uses the XRotation, YRotation, and ZRotation fields to create a vector with those values.

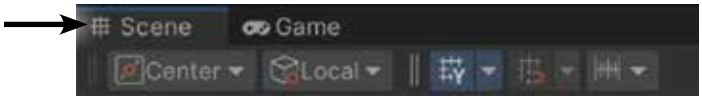
So what does that 3D vector look like? There's no need to guess—we can use one of Unity's useful debugging tools to draw the vector for us. **Add this line of code to the end of your Update method:**

```
void Update()
{
    Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
    transform.Rotate(axis, DegreesPerSecond * Time.deltaTime);
    Debug.DrawRay(Vector3.zero, axis, Color.yellow);
}
```

The Debug.DrawRay method is a special method that Unity provides to help you debug your games. It draws a **ray**—which is a vector that goes from one point to another—and takes parameters for its start point, end point, and color. There's one catch: ***the ray only appears in the Scene view***. The methods in Unity's Debug class are designed so that they don't interfere with your game. They typically only affect how your game interacts with the Unity editor.

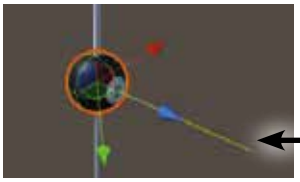
Run the game to see the ray in the Scene view

Now run your game again. You won't see anything different in the Game view because Debug.DrawRay is a tool for debugging that doesn't affect gameplay at all. Use the Scene tab to **switch to the Scene view**. You may also need to **reset the Wide layout** by choosing Wide from the Layout dropdown.



Now you're back in the familiar Scene view. Do these things to get a real sense of how 3D vectors work:

- Use the Inspector to **modify the BallBehaviour script's fields**. Set the X Rotation to 0, Y Rotation to 0, and **Z Rotation to 3**. You should now see a yellow ray coming directly out of the Z axis and the ball rotating around it (remember, the ray only shows up in the Scene view).



The vector (0, 0, 3) extends 3 units along the Z axis. Look closely at the grid in the Unity editor—the vector is exactly 3 units long. Try clicking and dragging the Z Rotation label in the Script component in the Inspector. The ray will get larger or smaller as you drag. When the Z value in the vector is negative, the ball rotates in the other direction.

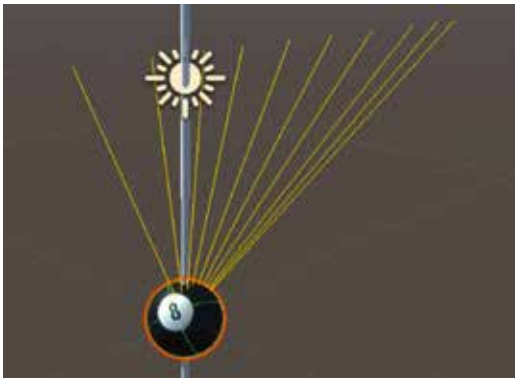
- Set the Z Rotation back to 3. Experiment with dragging the X Rotation and Y Rotation values to see what they do to the ray. Make sure to reset the Transform component each time you change them.
- Use the Hand tool and the Scene Gizmo to get a better view. Click the X cone on the Scene Gizmo to set it to the view from the right. Keep clicking the cones on the Scene Gizmo until you see the view from the front. It's easy to get lost—you can **reset the Wide layout to get back to a familiar view**.

Add a duration to the ray so it leaves a trail

You can add a fourth argument to your Debug.DrawRay method call that specifies the number of seconds the ray should stay on the screen. Add **.5f** to make each ray stay on screen for half a second:

```
Debug.DrawRay(Vector3.zero, axis, Color.yellow, .5f);
```

Now run the game again and switch to the Scene view. Now when you drag the numbers up and down, you'll see a trail of rays left behind. This looks really interesting, but more importantly, it's a great tool to visualize 3D vectors.

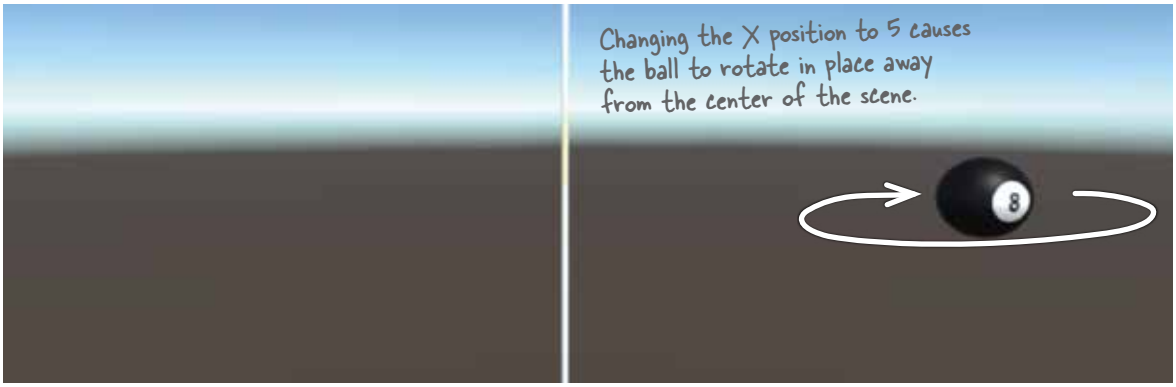


Making your ray leave a trail is a good way to help you develop an intuitive sense of how 3D vectors work.

You can use the Inspector window to modify the fields in a Script component while the game is running. The field values will reset when you stop the game. It will remember the values if you set them while the game is stopped.

Rotate your ball around a point in the scene

Your code calls the transform.Rotate method to rotate your ball around its center, which changes its X, Y, and Z rotation values. **Select Sphere in the Hierarchy window and change its X position to 5** in the Transform component. Then **use the context menu (⋮) in the BallBehaviour Script component** to reset its fields. Run the game again—now the ball will be at position (5, 0, 0) and rotating around its own Y axis.



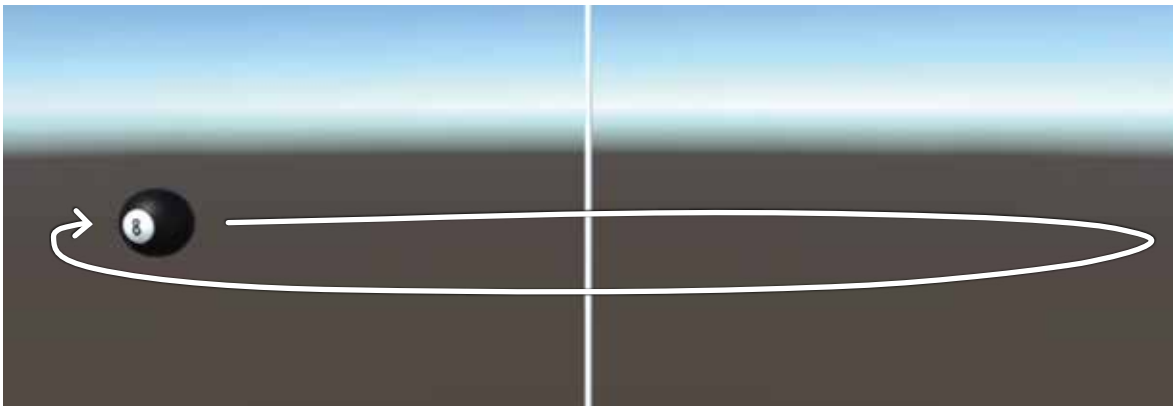
Let's modify the Update method to use a different kind of rotation. Now we'll make the ball rotate around the center point of the scene, coordinate (0, 0, 0), using the **transform.RotateAround method**, which rotates a GameObject around a point in the scene. (This is *different* from the transform.Rotate method you used earlier, which rotates a GameObject around its center.) Its first parameter is the point to rotate around. We'll use **Vector3.zero** for that parameter, which is a shortcut for writing `new Vector3(0, 0, 0)`.

Here's the new Update method:

This new Update method rotates the ball around the point (0, 0, 0) in the scene.

```
void Update()
{
    Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
    transform.RotateAround(Vector3.zero, axis, DegreesPerSecond * Time.deltaTime);
    Debug.DrawRay(Vector3.zero, axis, Color.yellow, .5f);
}
```

Now run your code. This time it rotates the ball in a big circle around the center point:

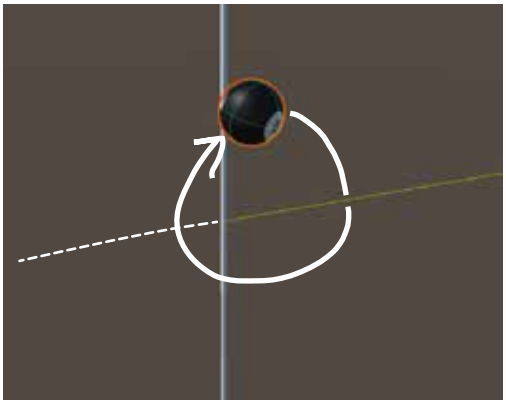


Use Unity to take a closer look at rotation and vectors

You're going to be working with 3D objects and scenes in the rest of the Unity Labs throughout the book. Even those of us who spend a lot of time playing 3D video games don't have a perfect feel for how vectors and 3D objects work, and how to move and rotate in a 3D space. Luckily, Unity is a great tool to **explore how 3D objects work**. Let's start experimenting right now.

While your code is running, try changing parameters to experiment with the rotation:

- **Switch back to the Scene view** so you can see the yellow ray that Debug.DrawRay renders in your BallBehaviour.Update method.
- Use the Hierarchy window to **select the Sphere**. You should see its components in the Inspector window.
- Change the **X Rotation, Y Rotation, and Z Rotation values** in the Script component to **10** so you see the vector rendered as a long ray. Use the Hand tool (Q) to rotate the Scene view until you can clearly see the ray.
- Use the Transform component's context menu () to **reset the Transform component**. Since the center of the sphere is now at the zero point in the scene, (0, 0, 0), it will rotate around its own center.
- Then **change the X position in** the Transform component to **2**. The ball should now be rotating around the vector. You'll see the ball cast a shadow on the Y axis cylinder as it flies by.



While the game is running, set the X, Y, and Z Rotation fields in the BallBehaviour Script component to 10, reset the sphere's Transform component, and change its X position to 2—as soon as you do, it starts rotating around the ray.

Try **repeating the last three steps** for different values of X, Y, and Z rotation, resetting the Transform component each time so you start from a fixed point. Then try clicking the rotation field labels and dragging them up and down—see if you can get a feel for how the rotation works.

Unity is a great tool to explore how 3D objects work by modifying properties on your GameObjects in real time.

Get creative!

This is your chance to **experiment on your own with C# and Unity**. You've seen the basics of how you combine C# and Unity GameObjects. Take some time and play around with the different Unity tools and methods that you've learned about in the first two Unity Labs. Here are some ideas:

- Add cubes, cylinders, or capsules to your scene. Attach new scripts to them—make sure you give each script a unique name!—and make them rotate in different ways.
- Try putting your rotating GameObjects in different positions around the scene. See if you can make interesting visual patterns out of multiple rotating GameObjects.
- Try adding a light to the scene. What happens when you use transform.rotateAround to rotate the new light around various axes?
- Here's a quick coding challenge: try using += to add a value to one of the fields in your BallBehaviour script. Make sure you multiply that value by Time.deltaTime. Try adding an `if` statement that resets the field to 0 if it gets too large.

Before you run the code, try to figure out what it will do. Does it act the way you expected it to act? Trying to predict how the code you added will act is a great technique for getting better at C#.

Take the time to experiment with the tools and techniques you just learned. This is a great way to take advantage of Unity and Visual Studio as tools for exploration and learning.

Bullet Points

- The **Scene Gizmo** always displays the camera's orientation.
- You can **attach a C# script** to any GameObject. The script's Update method will be called once per frame.
- The **transform.Rotate method** causes a GameObject to rotate a number of degrees around an axis.
- Inside your Update method, multiplying any value by **Time.deltaTime** turns it into that value per second.
- You can **attach** the Visual Studio debugger to Unity to debug your game while it's running. It will stay attached to Unity even when your game is not running.
- Adding a **Hit Count condition** to a breakpoint to makes it break after the statement has executed a certain number of times.
- A **field** is a variable that lives inside of a class outside of its methods, and it retains its value between method calls.
- Adding public fields to the class in your Unity script makes the Script component show **input boxes that let you modify those fields**. It adds spaces between capital letters in the field names to make them easier to read.
- You can create 3D vectors using **`new Vector3`**. (You learned about the `new` keyword in Chapter 3.)
- The **Debug.DrawRay method** draws a vector in the Scene view (but not the Game view). You can use vectors as a debugging tool, but also as a learning tool.
- The **transform.RotateAround method** rotates a GameObject around a point in the scene.