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

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

O'REILLY®

Fifth Edition

Also covers .NET MAUI & Unity

Head First C#

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

Andrew Stellman
& Jennifer Greene

Blazor Learner's Guide

This is a bonus companion guide to help you learn web development with C# and Blazor.

A Brain-Friendly Guide

Head First C#

What will you learn from this book?

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

Understand the relationship between classes and objects, and equip yourself with that knowledge to build successful apps.

Build satisfying and fun projects from the very first chapter.

What's so special about this book?

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

> "Thank you so much! Your books have helped me to launch my career."
>
> **—*Ryan White***
> Game Developer

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

> "Andrew and Jennifer have written a concise, authoritative, and, most of all, fun introduction to C# development."
>
> **—*Jon Galloway***
> Senior Program Manager on
> the .NET Community Team
> at Microsoft

C# / .NET

US $79.99 **CAN $99.99**

ISBN: 978-1-098-14178-3

57999

9 781098 141783

O'REILLY®

Praise for Head First C#

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

—Gerald Versluis, Senior Software Engineer at Microsoft

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

—Zakaria Soleymani, Development Team Lead

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

—Ryan White, Game Developer

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

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

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

—Jeff Counts, Senior C# Developer

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

—Lindsey Bieda, Lead Software Engineer

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

—Shalewa Odusanya, Principal

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

—Johnny Halife, Partner

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

—Rebeca Dunn-Krahn, Founding Partner, Sempahore Solutions

Praise for Head First C#

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

—Andy Parker, fledgling C# Programmer

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

—Chris Burrows, Software Engineer

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

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

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

—David Sterling, Principal Software Developer

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

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

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

—Giuseppe Turitto, Director of Engineering

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

—Bill Mietelski, Advanced Systems Analyst

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

—Krishna Pala, MCP

Praise for the Head First Approach

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

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

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

— Aaron LaBerge, SVP Technology & Product Development, ESPN

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

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

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

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

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

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

— Eric Wuehler

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

— Satish Kumar

Head First C#

Wouldn't it be dreamy if there was a C# book that's more fun than memorizing a dictionary? It's probably nothing but a fantasy...

Andrew Stellman
Jennifer Greene

O'REILLY®

Beijing • Boston • Farnham • Sebastopol • Tokyo

build web applications with C#

Head First C# presents...

Blazor Learner's Guide

Want to build great web apps…right now?

C# is great for building web applications that run in your browser. We didn't have room in *Head First C#* to teach you how to build web applications, but we wanted to give you a foundation that included **web development and page design**. (And that's why we created this **learner's guide**, a companion to *Head First C#* (5th edition) to teach the fundamentals of web application development using **Blazor**, Microsoft's free and open-source framework for building complete web applications.

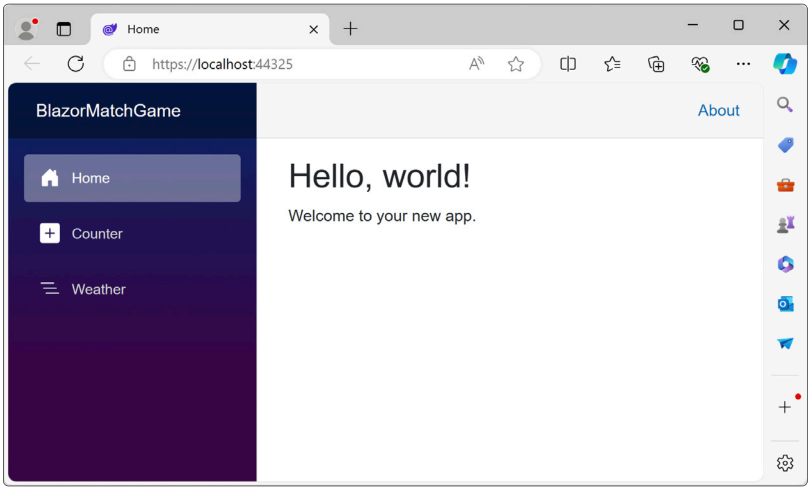
Build web applications with C# with Blazor

Welcome to the *Head First C#* **Blazor Learner's Guide**, a free downloadable companion to *Head First C#* (5th edition). The goal of this guide is to give you a foundation in **Blazor**, Microsoft's framework for building rich interactive web applications.

Blazor is great for C# developers because it allows them to use our existing skills and knowledge to build modern web applications. Here's why Blazor is so appealing:

- **Use C# Everywhere:** Developers can use C# for both client-side and server-side code, eliminating the need to switch between languages. Before Blazor, you had to use another language like JavaScript or TypeScript to build your web pages. Blazor lets you do all of your web development in C#.
- **Seamless Integration:** Blazor integrates seamlessly with the .NET ecosystem, allowing developers to use the extensive .NET classes and methods that you'll learn throughout the book.
- **Rich UI Development:** Blazor makes it straightforward to build dynamic and interactive user interfaces with features like data binding and event handling. You'll learn the basics of web design using HTML and **Bootstrap** to create responsive and visually rich pages in your web applications.

This is what you see when you create a new .NET Blazor application and run it. It will be your first building block for building web applications with C#.



Hello, world!

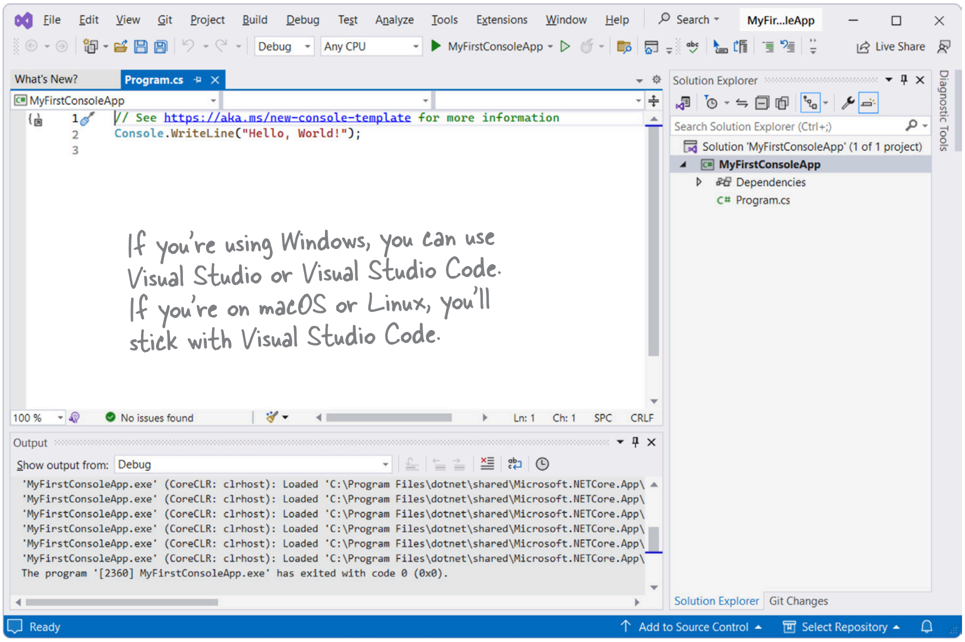
Welcome to your new app.

How to use this guide

Most projects in the book are console apps, but many chapters also include a project built using .NET MAUI, which you can use to build native and mobile applications. This learner's guide has **replacements** for all of those MAUI projects—including a *complete replacement for Chapter 1*—that use **C# to create Blazor Web Apps** that run in your browser and are equivalent to the Windows apps. You'll do it all with **Visual Studio** or **Visual Studio Code**, which you're already using in the rest of the book to **learn and explore C#**. Let's dive right in and get coding!

First step: get up and running with Visual Studio or Visual Studio Code

This book is filled with projects, and to do them you'll need to install Visual Studio or Visual Studio Code. Those are both advanced code editors and development environments built by Microsoft that you can download and use for free—and lucky for us, they make great tools for learning and exploration.



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

```
'MyFirstConsoleApp.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\
'MyFirstConsoleApp.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\
'MyFirstConsoleApp.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\
'MyFirstConsoleApp.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\
'MyFirstConsoleApp.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\
'MyFirstConsoleApp.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\
The program '[2360] MyFirstConsoleApp.exe' has exited with code 0 (0x0).
```

If you're using Windows, you can use Visual Studio or Visual Studio Code. If you're on macOS or Linux, you'll stick with Visual Studio Code.

Before you start this guide, make sure you read the first 18 pages of Head First C# (5th edition). They'll show you how to install Visual Studio or Visual Studio Code, and you'll create your first Console App project. Return to the Blazor Learner's Guide when you get to this heading:

Let's build a game!

You can download the first four chapters of the book for free from our GitHub page: https://github.com/head-first-csharp/fifth-edition

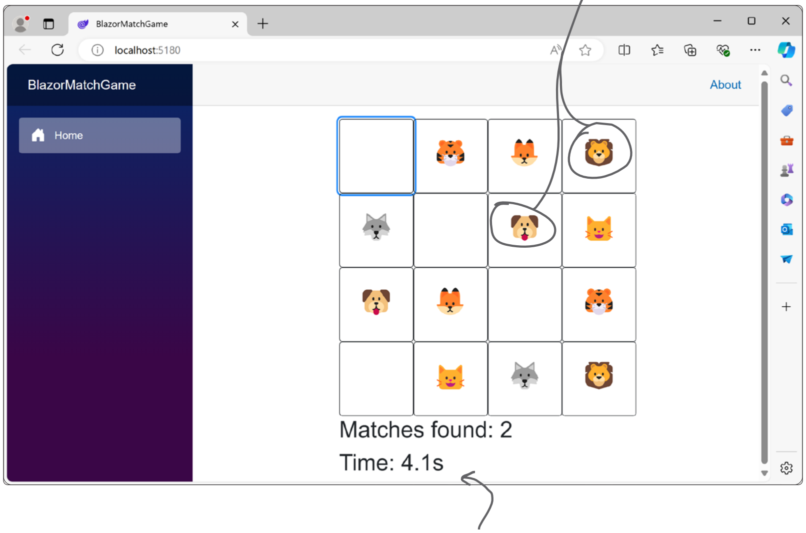
Make sure you've read the first 18 pages of the book, and have installed Visual Studio or Visual Studio Code before you start this project. You can download the first four chapters for free from our GitHub page: https://github.com/head-first-csharp/fifth-edition

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 a grid of 16 animals and needs to click on pairs to make them disappear.

Here's the animal matching game that you'll build.

The game shows eight different pairs of animals scattered randomly around the grid. The player clicks on two animals—if they match, they disappear from the page.



Matches found: 2

Time: 4.1s

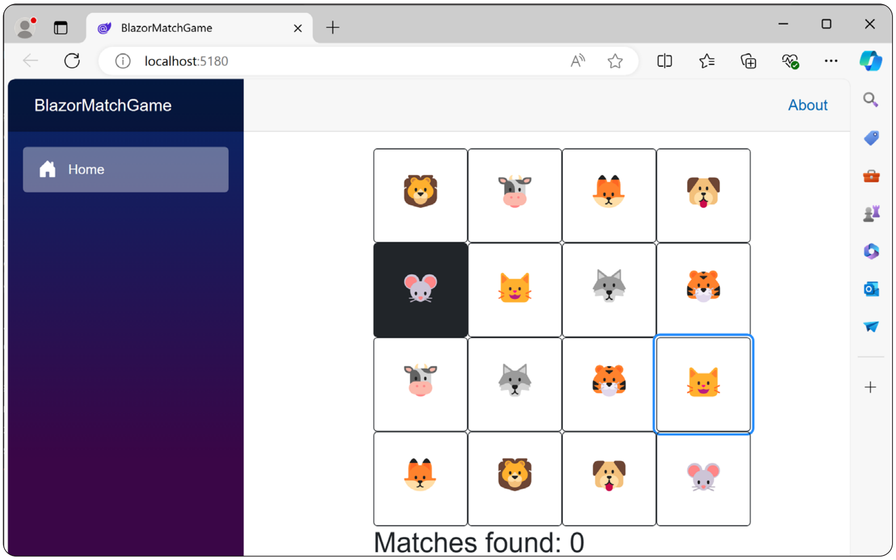
This timer keeps track of how long it takes the player to finish the game. The goal is to find all of the matches in as little time as possible.

By the time you're done with this project, you'll be a lot more familiar with the tools that you'll rely on throughout this book to learn and explore C#.

Your animal matching game is a Blazor Web App

Console apps are great if you just need to input and output text. If you want a visual app that's displayed on a browser page, you'll need to use a different technology. That's why your animal matching game will be a **Blazor Web app**. Blazor lets you create rich web applications that can run in any modern browser. Most of the chapters in this book will feature a Blazor app. The goal of this project is to introduce you to Blazor and give you tools to build rich web applications as well as console apps.

When you've found all eight pairs of animals, the game displays your final time and the text "Play again?" next to it. Click any animal button to start over again!



🦁	🐮	🦊	🐶
🐭	🐱	🐺	🐯
🐮	🐺	🐯	🐱
🦊	🦁	🐶	🐭

Matches found: 0

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? Or a farming simulator? 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.

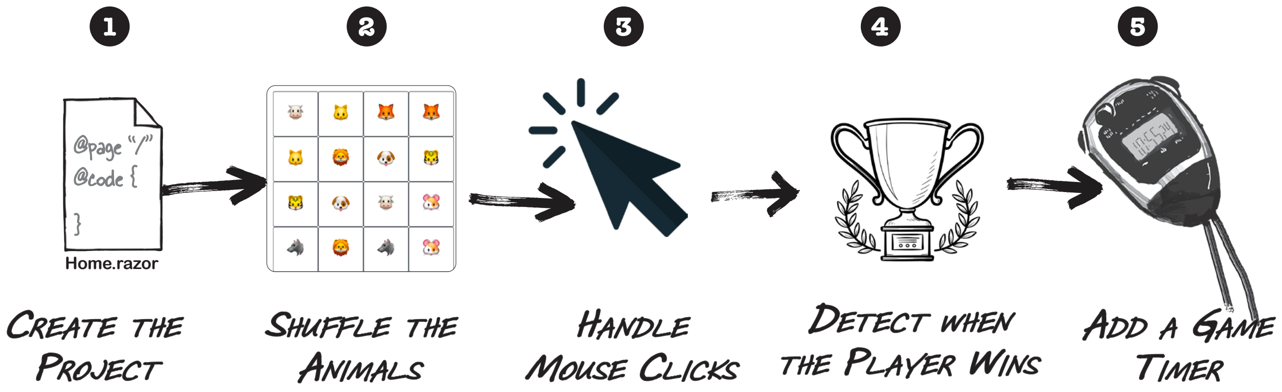
* Is there always **decision making, conflict, or problem solving**? Not in all games. Walking simulators are games where the player just explores in an environment, and there are often no puzzles or conflicts at all.

* 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 compelling 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.

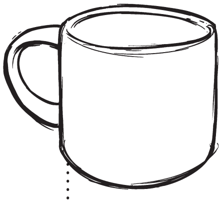
Break up large projects into smaller parts

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.



The goal of this project is to help get you used to writing C# and using the IDE. 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.

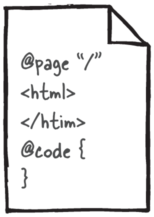
Here's how you'll build your game

You'll build your animal matching game using **Blazor**, Microsoft's technology that you can use to create highly interactive web apps in C# that can run in your browser.

The rest of this chapter will walk you through building the game. 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.

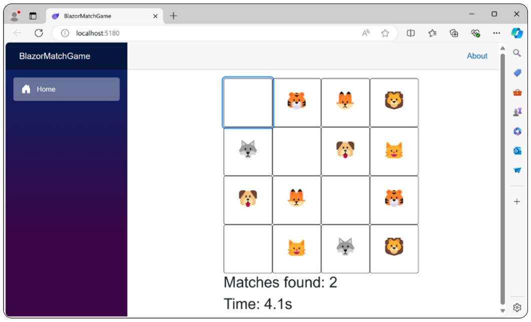
1. **First you'll create a new Blazor WebAssembly App project.**
   You just created a new console application. Now you'll create a new Blazor app.



```
@page "/"
<html>
</htim>
@code {
}
```

Home.razor

2. **Then you'll lay out the page and write C# code to shuffle the animals.**
   When your app first loads, it will run that code to display 16 buttons with eight pairs of animal emoji in a random order.



Matches found: 2
Time: 4.1s

3. **The game needs to let the user click on pairs of emoji to match them.**
   The game needs to detect when the user clicks on pairs of emoji, and keep track of those pairs. You'll write code to handle those clicks.

4. **You'll write more C# code to detect when the player has won the game.**
   The app will end the game when the player has found all of the matches. You'll write that code too.

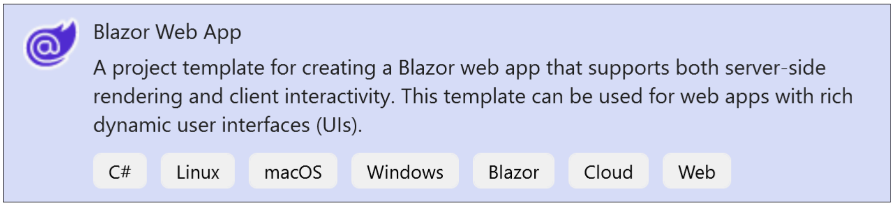
5. **Finally, you'll make the game more exciting by adding a timer.**
   Your timer will start when the player starts the game, and keep track of how long it takes the player to find all eight pairs of animals.

Create a Blazor Web App project in Visual Studio

Let's create a new Blazor Web App project. Before you start, if you still have the Console App project open, close it by choosing *File >> Close Solution* from the menu.

Next, **create a new Blazor Web App project**. If the "Get Started" window is displayed. click the *Create a new project* button. If not, choose *File >> New >> Project...* from the menu.

Click the Blazor Web App option:

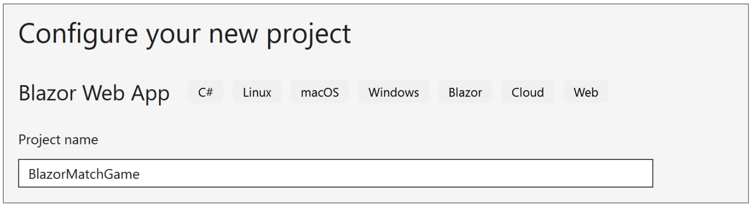


Blazor Web App

A project template for creating a Blazor web app that supports both server-side rendering and client interactivity. This template can be used for web apps with rich dynamic user interfaces (UIs).

C# Linux macOS Windows Blazor Cloud Web

Name your project BlazorMatchGame and click the Next button:



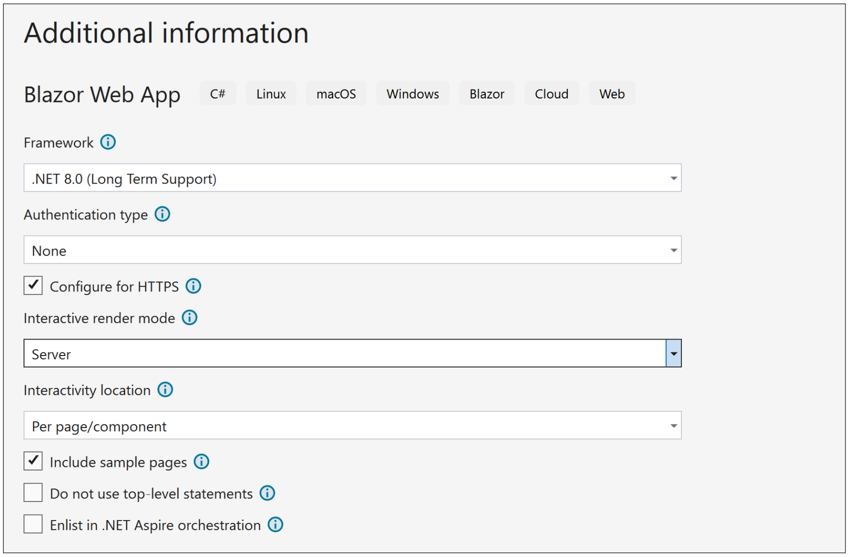
Configure your new project

Blazor Web App C# Linux macOS Windows Blazor Cloud Web

Project name

BlazorMatchGame

Accept the default *Additional information* options and click the Create button to create the project:



Additional information

Blazor Web App C# Linux macOS Windows Blazor Cloud Web

Framework ⓘ

.NET 8.0 (Long Term Support)

Authentication type ⓘ

None

☑ Configure for HTTPS ⓘ

Interactive render mode ⓘ

Server

Interactivity location ⓘ

Per page/component

☑ Include sample pages ⓘ

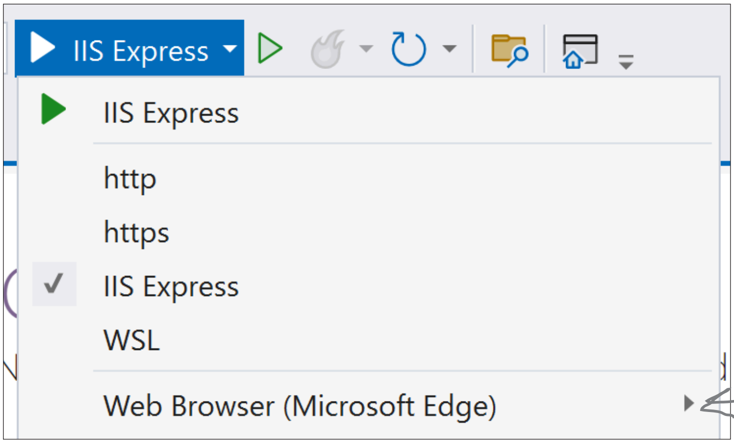
☐ Do not use top-level statements ⓘ

☐ Enlist in .NET Aspire orchestration ⓘ

Visual Studio remembers the most recent choices that you made when creating a new project, so look over them and make sure they match the selections in our screenshot.

If you're using Visual Studio Code, skip to the next section, which shows you how to create a new Blazor Web App project with VSCode.

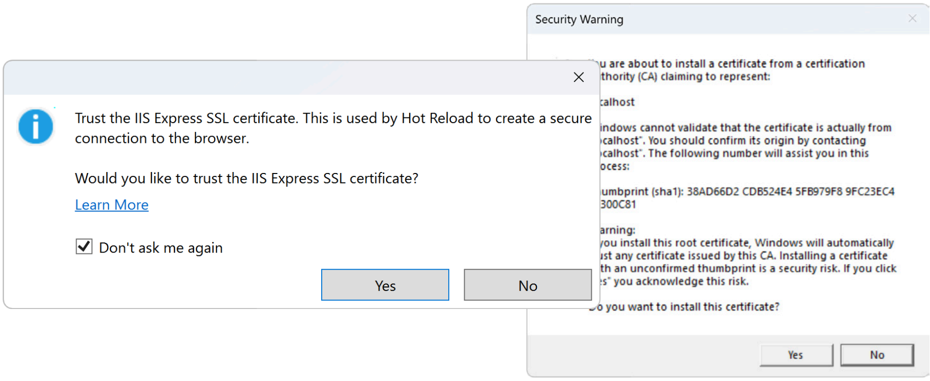
Once your project is loaded, **run your app**. Find the Run button at the top of the Visual Studio window and click the dropdown next to it to select IIS Express:



Your app will run in the Edge browser by default. If you have Chrome installed, you can use this sub-menu to run your app in Chrome instead.

Then click the ▶ IIS Express button to run your app.

You may see several windows asking you to accept certificates—make sure to accept them:



Security Warning

u are about to install a certificate from a certification thority (CA) claiming to represent:

calhost

indows cannot validate that the certificate is actually from calhost". You should confirm its origin by contacting calhost". The following number will assist you in this ocess:

umbprint (sha1): 38AD66D2 CDB524E4 5FB979F8 9FC23EC4 300C81

arning:
you install this root certificate, Windows will automatically ust any certificate issued by this CA. Installing a certificate th an unconfirmed thumbprint is a security risk. If you click s" you acknowledge this risk.

o you want to install this certificate?

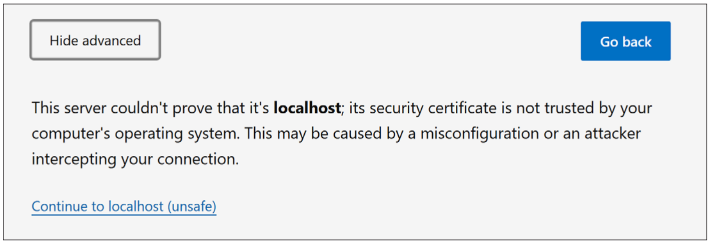
Trust the IIS Express SSL certificate. This is used by Hot Reload to create a secure connection to the browser.

Would you like to trust the IIS Express SSL certificate?

Learn More

☑ Don't ask me again

You may also be displayed a window saying that your connection isn't private. If you see it, click the Advanced button and then click the *Continue to localhost (unsafe)* link.



This server couldn't prove that it's **localhost**; its security certificate is not trusted by your computer's operating system. This may be caused by a misconfiguration or an attacker intercepting your connection.

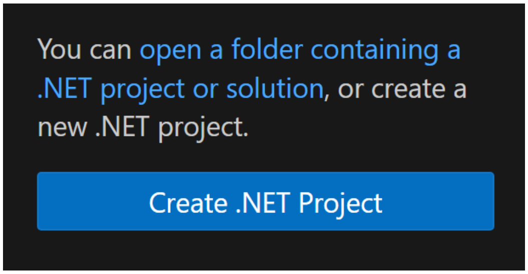
Continue to localhost (unsafe)

Skip the next two pages, which tell you how to set up your Blazor project in VSCode.

Create a Blazor Web App in Visual Studio Code

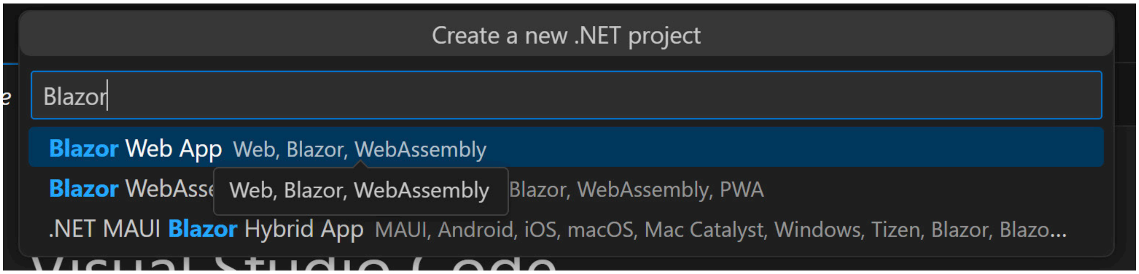
Let's create a new Blazor Web App project. Before you start, if you still have the Console App project open, close it by choosing *File >> Close Folder* from the menu.

Next, **create a new Blazor Web App project**. Start by clicking the *Create .NET Project* button.



You can open a folder containing a .NET project or solution, or create a new .NET project.

You'll be prompted to create a new .NET project. Search for *Blazor* and **select *Blazor Web App***:

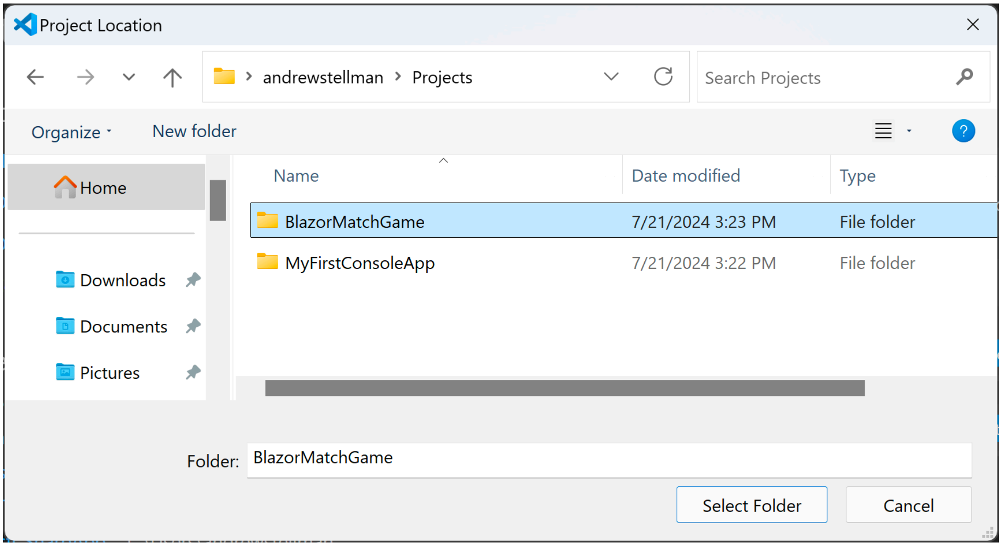


Create a new .NET project

Blazor

Blazor Web App Web, Blazor, WebAssembly
Blazor WebAss Web, Blazor, WebAssembly Blazor, WebAssembly, PWA
.NET MAUI Blazor Hybrid App MAUI, Android, iOS, macOS, Mac Catalyst, Windows, Tizen, Blazor, Blazo...

You'll be prompted to choose a project location. Create a folder called **BlazorMatchGame** and select it.



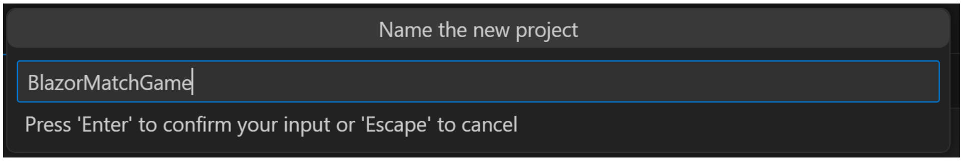
Project Location

Name	Date modified	Type
BlazorMatchGame	7/21/2024 3:23 PM	File folder
MyFirstConsoleApp	7/21/2024 3:22 PM	File folder

Folder: BlazorMatchGame

On macOS you'll see the Mac version of the dialog to select the folder.

You'll be prompted for a project name. Name your project AnimalMatchingGame, just like the folder.

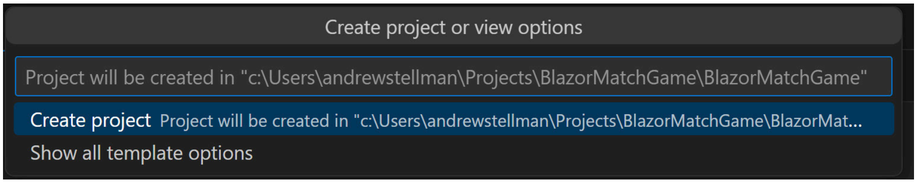


Name the new project

BlazorMatchGame

Press 'Enter' to confirm your input or 'Escape' to cancel

Press enter to accept the default location to create your project:



Create project or view options

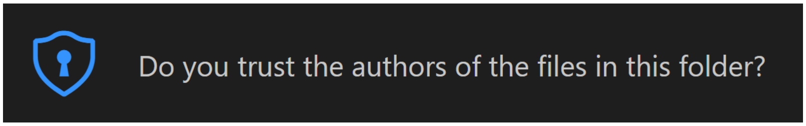
Project will be created in "c:\Users\andrewstellman\Projects\BlazorMatchGame\BlazorMatchGame"

Create project Project will be created in "c:\Users\andrewstellman\Projects\BlazorMatchGame\BlazorMat...

Show all template options

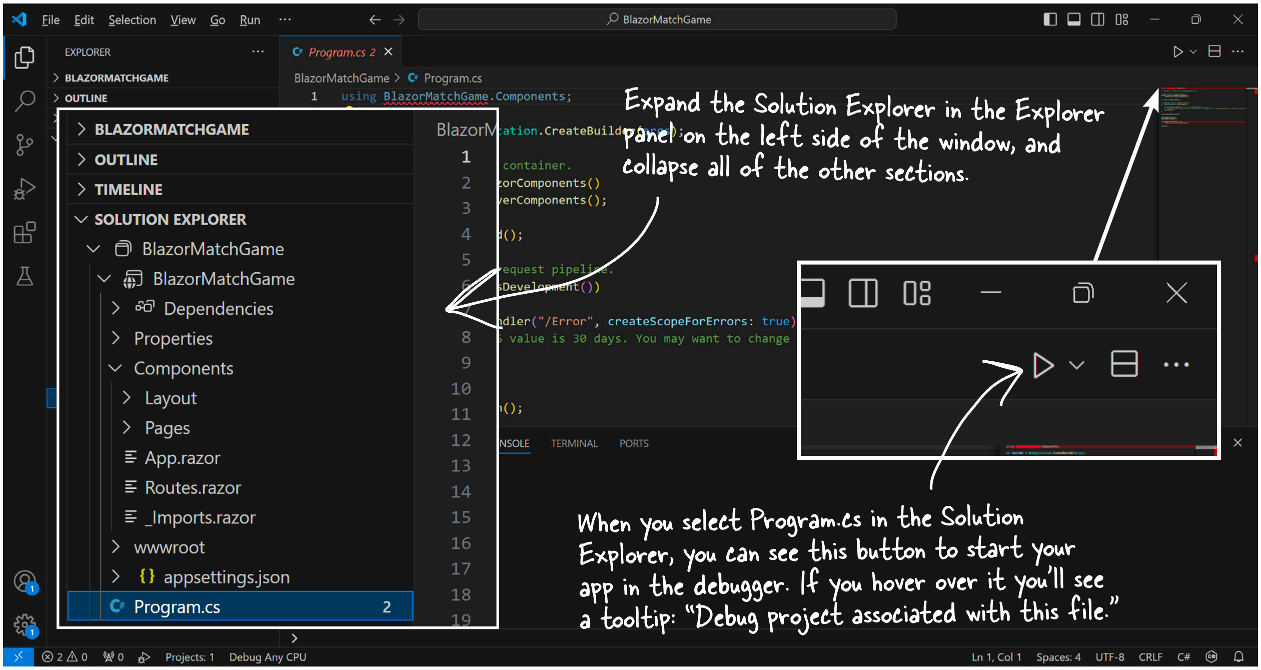
If you're on a Mac, Safari will run your web apps just fine, but you won't be able to use it to debug them. Web app debugging is only supported in Microsoft Edge or Google Chrome. Go to https://microsoft.com/edge to download Edge, or https://google.com/chrome to download Chrome.

If you're asked to trust the authors of the files in the folder, choose Yes:



Do you trust the authors of the files in this folder?

Wait for Visual Studio Code to finish creating your project.



Expand the Solution Explorer in the Explorer panel on the left side of the window, and collapse all of the other sections.

When you select Program.cs in the Solution Explorer, you can see this button to start your app in the debugger. If you hover over it you'll see a tooltip: "Debug project associated with this file."

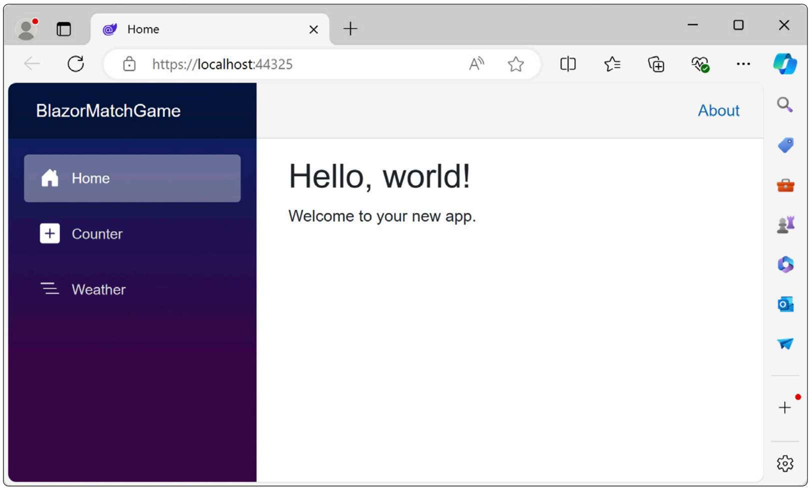
Click on Program.cs in the Solution Explorer (not any of the other sections in the Explorer panel). Find the triangle shaped button in the upper right corner and click it to start your web application open up its home page the Microsoft Edge or Google Chrome browser. You can also choose **Start Debugging (F5) from the Run menu** to start your app (choose the defaults if it prompts you), then you can **press F5** at any time to run your app.

Your Blazor web app runs in a browser

When you run a Blazor web app, there are two parts: a **server** and a **web application**. Visual Studio launches them both with one button.

Do this!

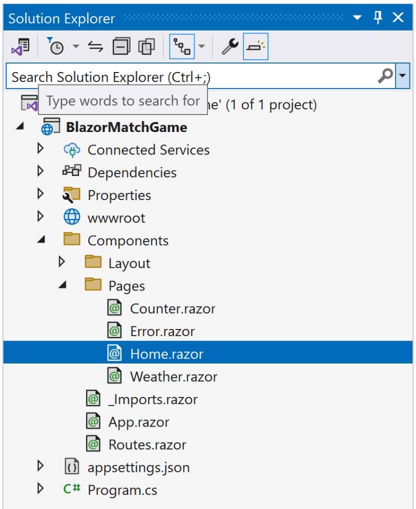
1 **Interact with your web app.**
When you run your web application, the IDE automatically opens browser window running your app:



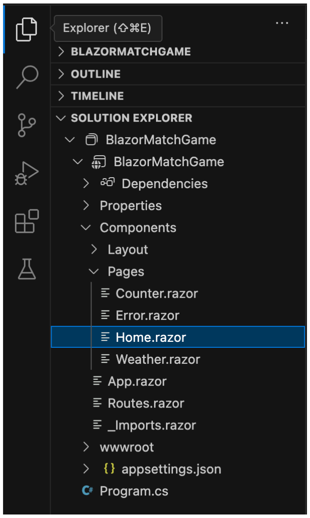
Hello, world!

Welcome to your new app.

2 **Find the file with the HTML code for the page that you're looking at.**
Go to the Solution Explorer and expand the BlazorMatchGame solution. Inside it you'll find a BlazorMatchGame project, and underneath it the Components folder that contains a Pages folder. Open the Pages folder and either double-click (Visual Studio) or click (VSCode) on *Home.razor*.



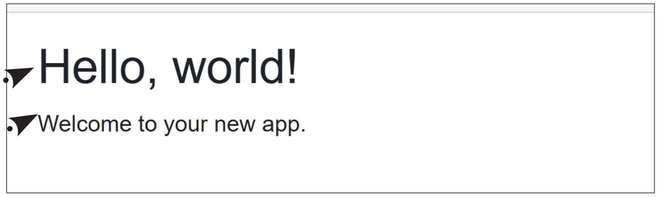
← Visual Studio and VSCode → have Solution Explorer windows that look a little different, but contain the same files and work the same way (for the most part).



3 Compare the code in Home.razor with what you see in your browser.

The web app in your browser has two parts: a **navigation menu** on the left side with links to different pages (Home, Counter, and Fetch data), and a page displayed on the right side. Compare the HTML markup in the *Home.razor* file with the app displayed in the browser.

```
@page "/"

<PageTitle>Home</PageTitle>

<h1>Hello, world!</h1>

Welcome to your new app.

```



Hello, world!

Welcome to your new app.

The <PageTitle> tag sets the title of the page that's displayed the tab in your browser.

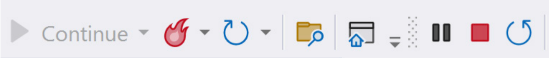
4 Change "Hello, world!" to something else.

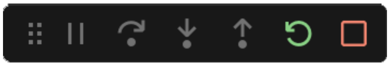
Change the third line of the *Home.razor* file so it says something else:

```
<h1>Elementary, my dear Watson.</h1>
```

Now go back to your browser and reload the page. Wait a minute, nothing changed—it still says "Hello, world!" That's because you changed your code, ***but you never updated the server***.

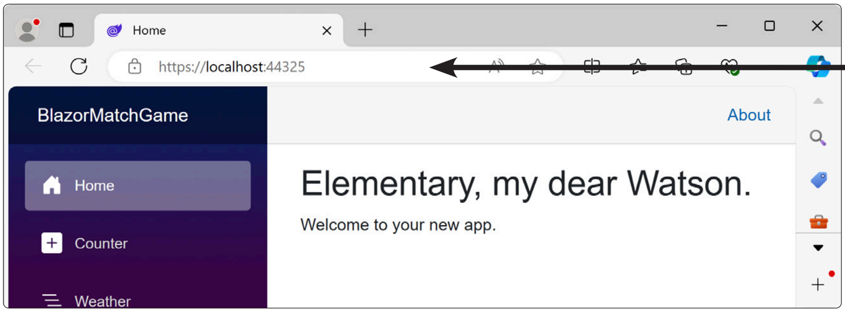
Click the Stop button to stop the application. In Visual Studio, click the square button in the toolbar or choose Stop Debugging (Shift+F5) from the menu. In VSCode, click the square button at the top of the code window, or press Shift+F5 to stop debugging.





Visual Studio may close your browser automatically for you. If it didn't, go back and reload your browser—since you stopped your app, it displays its "Site can't be reached" page. (If your browser closed when you stopped debugging, run the app it again, copy the URL, stop your app, then open a new browser window and paste it in.)

Start your app again, then reload your page in the browser. Now you'll see the updated text.

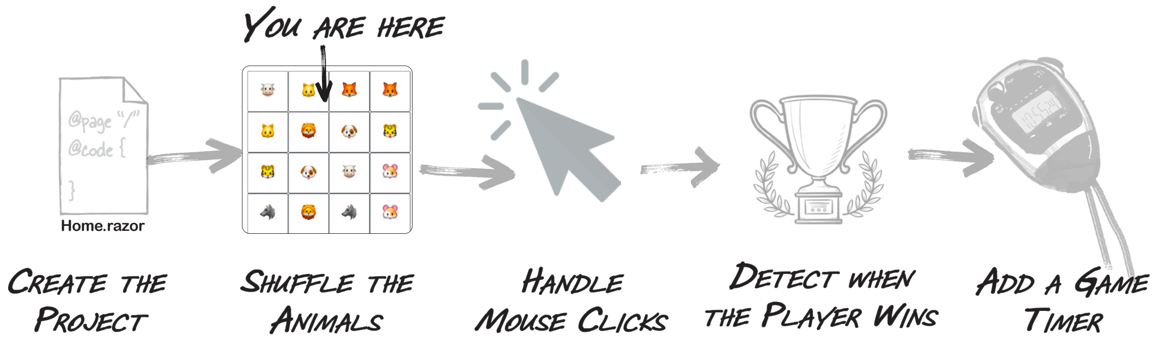


Elementary, my dear Watson.

Welcome to your new app.

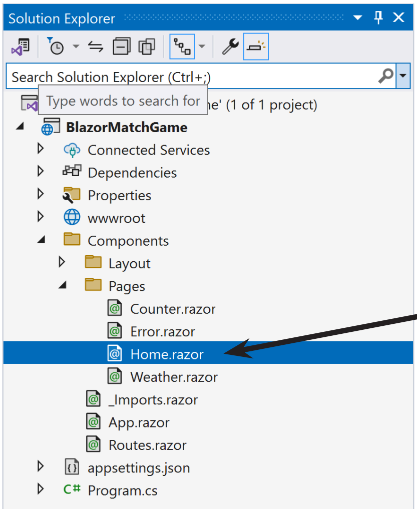
Try copying the URL from your browser, opening a new Safari window, and pasting it in. Your application will run there, too. Now you have two different browsers connecting to the same server.

Do you have extra instances of your browser open, or extra tabs? The IDE opens a new browser or tab each time you run your Blazor web app, and VSCode leaves it open. Get in the habit of closing the browser before you stop your app.



Now you're ready to start writing code for your game

You've created a new app, and Visual Studio generated a bunch of files for you. Now it's time to add C# code to start making your game work (as well as HTML markup to make it look right).



Solution Explorer

Search Solution Explorer (Ctrl+;)

Type words to search for e' (1 of 1 project)

- BlazorMatchGame
  - Connected Services
  - Dependencies
  - Properties
  - wwwroot
  - Components
    - Layout
    - Pages
      - Counter.razor
      - Error.razor
      - Home.razor
      - Weather.razor
    - _Imports.razor
    - App.razor
    - Routes.razor
  - appsettings.json
  - Program.cs

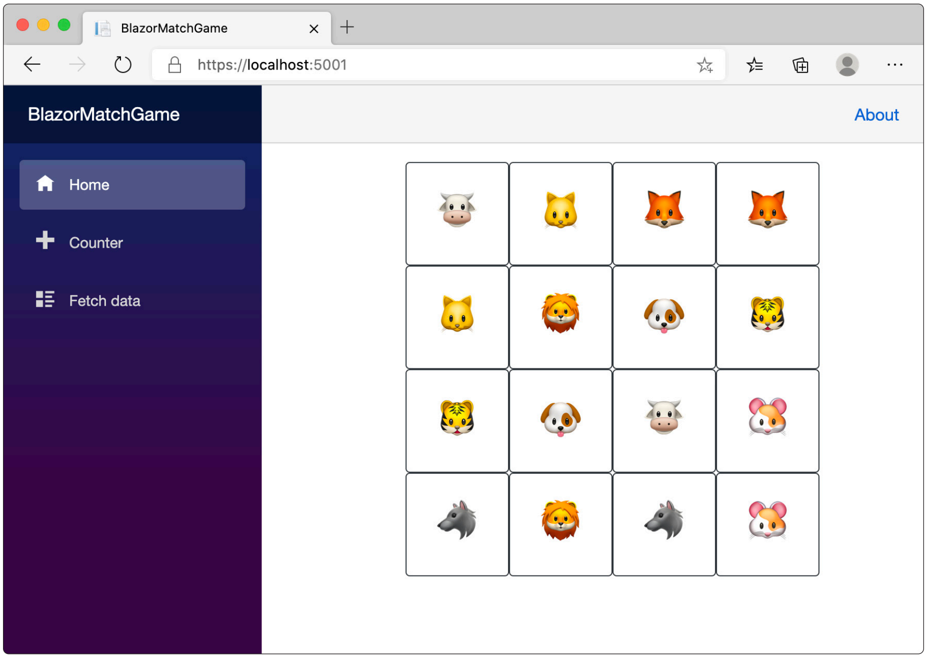
Now you'll start working on the C# code, which will be in the Home.razor file. A file that ends with .razor is a Razor markup page. Razor combines HTML for page layout with C# code, all in the same file. You'll add C# code to this file that defines the behavior of the game, including code to add the emoji to the page, handle mouse clicks, and make the countdown timer work.

When you created your console app earlier in the chapter, your C# code was in a file called Program.cs—when you see that .cs file extension, it tells you that the file contains C# code.

Watch it!

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.*



How the page layout in your animal matching game will work

Your animal matching game is laid out in a grid—or, at least, that's how it looks. It's actually made up of 16 square buttons.

You'll lay out the page by creating a container that's 400 pixels wide (a CSS "pixel" is 1/96 inch when the browser is at default scale) that contains 100-pixel-wide buttons. We'll give you all of the C# and HTML code to enter into the IDE. **Keep an eye out for this code** that you'll add to your project ***soon***—it's where the "magic" happens, by using ***Razor markup*** to mix C# code with HTML:

```
<div class="container">
   <div class="row">
      @foreach (var animal in animalEmoji)
      {
         <div class="col-3">
            <button type="button" class="btn btn-outline-dark">
               <h1>@animal</h1>
            </button>
         </div>
      }
   </div>
</div>
```

The @ symbol is used in Razor pages to switch from HTML to C# code. In this line of code, @foreach is used to create a loop that goes through a list of animal emojis to add a block of HTML to the page. For each emoji in the list, the loop generates a button.

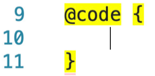
The foreach loop causes everything between the { and } to be repeated once for each emoji in a list of animal emoji, replacing `@animal` with each of the emoji in the list one by one. Since the list has 16 emoji, the result is a series of 16 buttons.

The IDE helps you write C# code

Blazor lets you create rich, interactive apps that combine HTML markup and C# code. Luckily, both Visual Studio and Visual Studio Code have useful features to help you write that C# code.

① Add C# code to your Home.razor file.

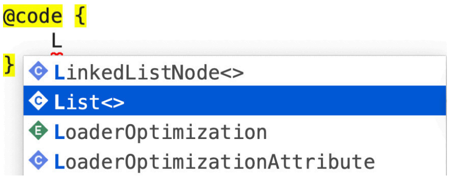
Start by **adding a `@code` block** to the end of your *Home.razor* file. (Keep the existing contents of the file there for now—you'll delete them later.) ***Go to the last line of the file*** and type `@code {`. The IDE will fill in the closing curly bracket `}` for you. Press Enter to add a line between the two brackets:



```
@code {
      |
}
```

② Use the IDE's IntelliSense window to help you write C#.

Position your cursor on the line between the `{` brackets `}` and type the letter `L`. The IDE will pop up an **IntelliSense window** with autocomplete suggestions. Choose `List<>` from the pop-up:



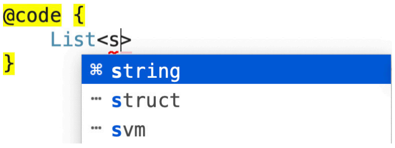
```
@code {
    L
}
```

The IntelliSense window in the IDE pops up and helps you write your C# code by suggesting useful autocomplete options. Use the arrow keys to choose an option and press Enter to select it (or use your mouse).

The IDE will fill in `List`. Add an **opening angle bracket** (greater-than sign) `<`—the IDE will automatically fill in the closing bracket `>` and leave your cursor positioned between them.

③ Start creating a List to store your animal emoji.

Type s to bring up another IntelliSense window:



```
@code {
    List<s>
}
```

Choose `string`—the IDE will add it between the brackets. Press the **right arrow and then the space bar**, then **type `animalEmoji = [`**.

As soon as you typed the opening square bracket `[`, Visual Studio added a matching one, placing your mouse cursor between the two brackets.:

```
List<string> animalEmoji = [|]
```

Press Enter, then add a semicolon 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!

④ Add a pair of animal emoji to your list.

Your C# statement isn't done yet. Make sure your cursor is placed on the blank line you added between the brackets. Now let's add **eight pairs of animal emoji**. You can find emoji by going to your favorite emoji website (for example, https://emojipedia.org/nature) and copying 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** (press the fn key, or Ctrl+⌘+Space on older Macs).

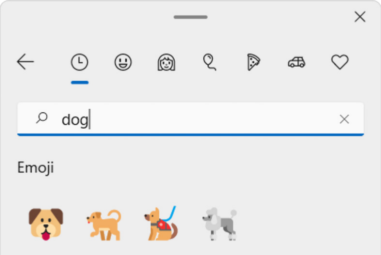
Go back to your code and add a double quote " then paste the character—we used an octopus—followed by another " and a comma, a space, another ", the same character again, and one more " and comma. You might notice Visual Studio helping you enter this list—for example, when you enter a double quote, it adds the closing quote.

Here's what your list should look like now:

```
@code {
    List<string> animalEmoji = new List<string>()
    {
        "🐶"
    };
}
```

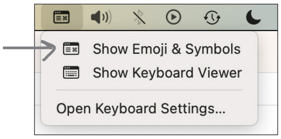
How to enter emoji

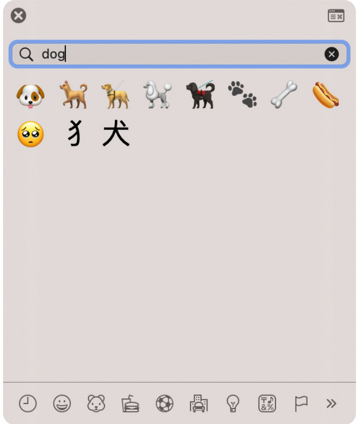
If you're using Windows, use the **emoji panel** by pressing Windows logo key ⊞ + period. Use the search box to search for a specific animal. When you find the emoji you want to enter, click on it to enter it as if you'd typed it.



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**, by pressing Ctrl + ⌘ + space. 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.

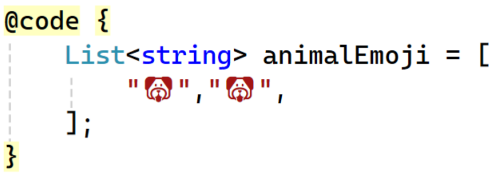




You can also bring up the macOS Character Viewer using the Input menu in the menu bar. If you don't see the Input menu, open System Settings and search for "input menu"—there's an option that you can turn on to show the input menu in the menu bar.

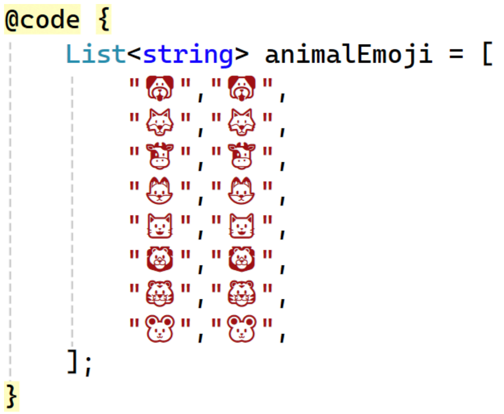
Finish creating your emoji list and display it in the app

You just added a dog emoji to your `animalEmoji` list. Now add a **second dog emoji** by adding a comma after the second quote, then a space, another quote, another dog emoji, another quote, and a comma:



```
@code {
    List<string> animalEmoji = [
        "🐶","🐶",
    ];
}
```

Now **add a second line right after it** that's exactly the same, except with a pair of wolf emoji instead of dogs. Then add six more lines with pairs of cows, foxes, cats, lions, tigers, and hamsters. You should now have eight pairs of emoji in your `animalEmoji` list:



```
@code {
    List<string> animalEmoji = [
        "🐶","🐶",
        "🐺","🐺",
        "🐮","🐮",
        "🦊","🦊",
        "🐱","🐱",
        "🦁","🦁",
        "🐯","🐯",
        "🐹","🐹",
    ];
}
```

IDE Tip: Indent lines

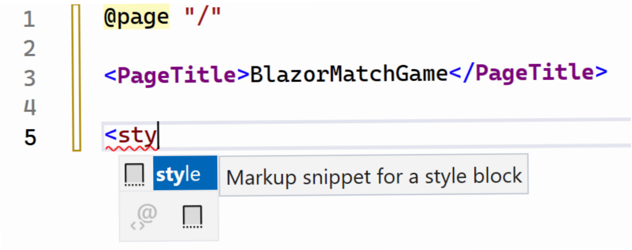
The IDE automatically indents your C# code for you as you enter it. But when you're entering the emoji or HTML tags, you might find that it doesn't quite indent them the way you want. You can easily fix that by selecting the text you want to indent and pressing ⇥ (Tab) to indent, or ⇧⇥ (Shift+Tab) to unindent.

Replace the contents of the page

Delete these lines from the top of the page:

```
~~<h1>Elementary, my dear Watson.</h1>~~
~~Welcome to your new app.~~
```

Update the <PageTitle> tag to replace Home with BlazorMatchGame. Then put your cursor on the third line of the page and **type `<st`**—the IDE will pop up an IntelliSense window:



```
@page "/"

<PageTitle>BlazorMatchGame</PageTitle>

<sty
```

The IDE will help you write HTML for your page—in this case, you're creating an HTML tag. It's OK if you don't know HTML; we'll give you all of the code that you need for your apps throughout the book.

Choose `style` from the list, then **type `>`**. The IDE will add a *closing HTML tag*: `<style></style>`

Put your cursor between `<style>` and `</style>` and press Enter, then **carefully enter all of the following code**. Make sure the code in your app matches it exactly.

```
<style>
    .container {
        width: 400px;
    }

    button {
        width: 100px;
        height: 100px;
        font-size: 50px;
    }
</style>
```

The matching game is made up of a series of buttons. This is a really simple CSS stylesheet to set the total width of the container, and the height and width of each button. Since the container is 400 pixels wide and each button is 100 pixels wide, the page will only allow four columns in a row before adding a break, making them appear in a grid.

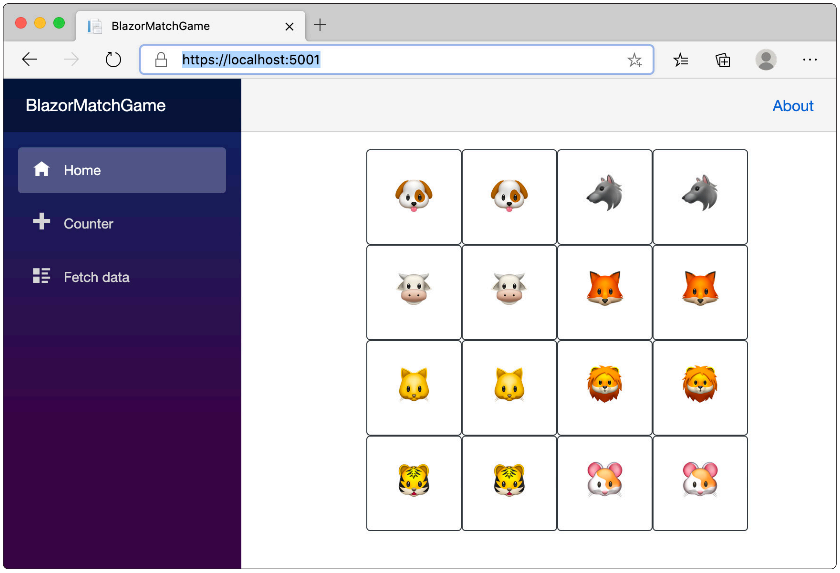
Go to the next line and use the IntelliSense to **enter an opening and closing `<div>` tag**, just like you did with `<style>` earlier. Then **carefully enter the code below**, making sure it matches exactly:

```
<div class="container">
    <div class="row">
        @foreach (var animal in animalEmoji)
        {
            <div class="col-3">
                <button type="button" class="btn btn-outline-dark">
                   <h1>@animal</h1>
                </button>
            </div>
        }
    </div>
</div>
```

If you've worked with HTML before, you'll notice the `@foreach` and `@animal` that don't look like ordinary HTML. That's Blazor—C# code embedded directly into the HTML.

Each button on the page contains a different animal. The players will press the buttons to find matches.

Changing the PageTitle tag changed the name of the page displayed in the browser tab.



Make sure your app looks like this screenshot when you run it. Once it does, you'll know you entered all of the code without any typos.

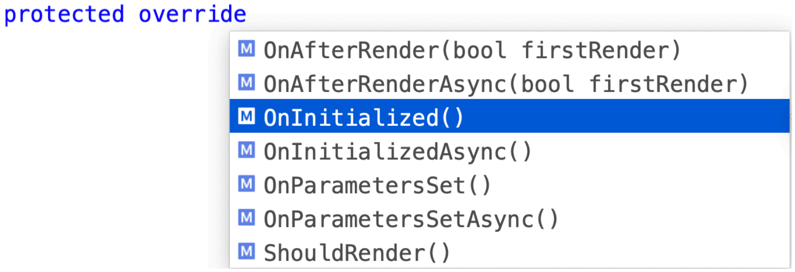
Shuffle the animals so they're in a random order

Our match game would be too easy if the pairs of animals were all next to each other. Let's add C# code to shuffle the animals so they appear in a different order each time the player reloads the page.

1. Place your cursor just after the semicolon **;** just above the closing bracket **}** near the bottom of *Home.razor* and **press Enter twice**. Then use the IntelliSense pop-ups just like you did earlier to enter the following line of code:

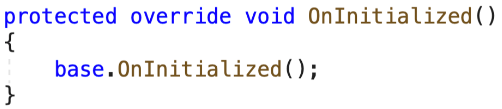
```
List<string> shuffledAnimals = new List<string>();
```

2. Next **type `protected override`** (the IntelliSense can autocomplete those keywords). As soon as you enter that and type a space, you'll get an IntelliSense pop-up—**select `OnInitialized()`** from the list:



```
protected override
```

The IDE will fill in code for a **method** called OnInitialized (we'll talk more about methods in Chapter 2):



```
protected override void OnInitialized()
{
    base.OnInitialized();
}
```

3. **Replace `base.OnInitialized()` with `SetUpGame()`** so your method looks like this:

```
protected override void OnInitialized()
{
    SetUpGame();
}
```

VSCode might require you to do a little more work, like typing the word void between override and OnInitialized(). Make sure your code matches the code on this page exactly.

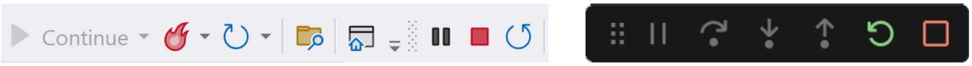
Then **add this SetUpGame method** just below your OnInitialized method—again, the IntelliSense window will help you get it right:

```
private void SetUpGame()
{
    shuffledAnimals = animalEmoji
        .OrderBy(item => Random.Shared.Next())
        .ToList();
}
```

You just added two methods to your app, but it's OK if you're still not 100% clear on what a method is. You'll learn much more about methods and how C# code is structured in the next chapter.

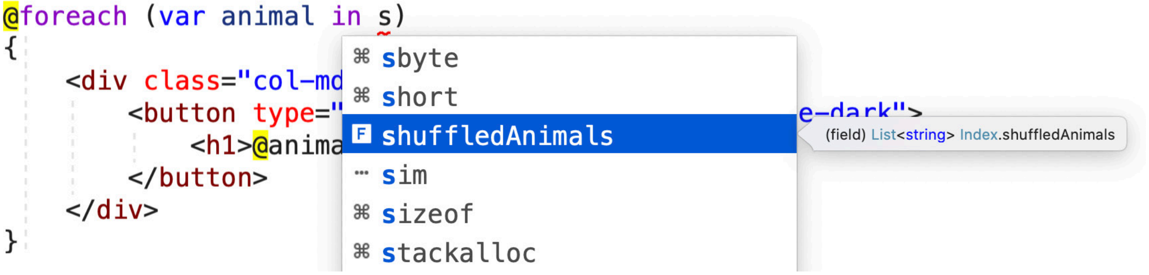
As you type in the SetUpGame method, you'll notice that the IDE pops up many IntelliSense windows to help you enter your code more quickly. The more you use Visual Studio to write C# code, the more helpful these windows will become—you'll eventually find that they significantly speed things up. For now, use them to keep from entering typos—your code needs to ***match our code exactly*** or your app won't run.

Do you see the run toolbar at the top of the IDE? That means your app is still running. Press the square stop button or choose Stop Deubgging (Shift+F5) from the Debug or Run menu.



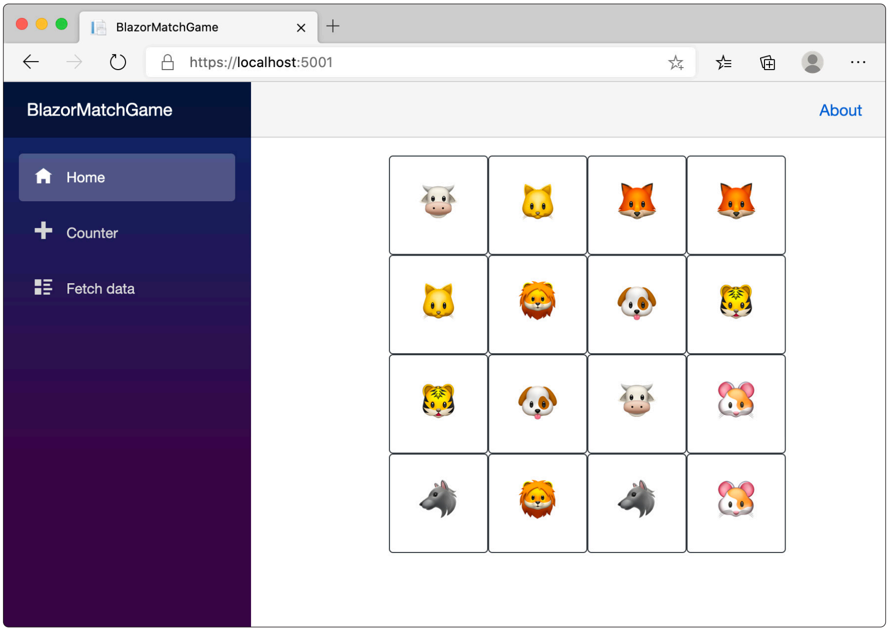
4 Scroll back up to the HTML and find this code: `@foreach (var animal in animalEmoji)`

Double-click `animalEmoji` to select it, then **type `s`**. The IDE will pop up an IntelliSense window. Choose `shuffledAnimals` from the list:



```
@foreach (var animal in s)
{
    <div class="col-md                              e-dark">
        <button type="
            <h1>@anima
        </button>
    </div>
}
```

Now **run your app again**. Your animals should be shuffled so they're in a random order. **Reload the page** in the browser—they'll be shuffled in a different order. Each time you reload, it reshuffles the animals.

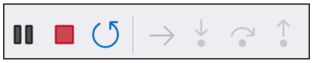


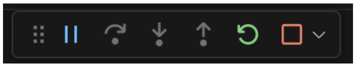
Again, make sure your app looks like this screenshot when you run it. Once it does, you'll know you entered all of the code without any typos. Don't move on until your game is reshuffling the animals every time you reload the browser page.

You're running your game in the debugger

When you click the Run button in the toolbar or choose Start Debugging (F5) from the Run or Debug menu to start your program running, you're putting the IDE into **debugging mode**.

You can tell that you're debugging an app when you see the **debug controls** appear in the toolbar (Visual Studio) or at the top of the window (VSCode).





Hover your mouse cursor over the Pause Execution (it has two lines) button to see its tooltip.

You can stop your app clicking the Stop button or choosing Stop Debugging (Shift F5) from the Debug or Run menu.

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.

Here's a pencil-and-paper exercise. It's absolutely worth your time to do all of them because they'll help get important C# concepts into your brain faster.

WHO DOES WHAT?

Congratulations—you've created a working app! Obviously, programming is more than just copying code out of a book. But even if you've never written code before, you may surprise yourself with just how much of it you already understand. Draw a line connecting each of the C# statements on the left to the description of what the statement does on the right. We'll start you out with the first one.

C# statement

```
List<string> animalEmoji = new List<string>()
{
      "🐶", "🐶",
      "🐺", "🐺",
      "🐮", "🐮",
      "🦊", "🦊",
      "🐱", "🐱",
      "🦁", "🦁",
      "🐯", "🐯",
      "🐹", "🐹",
};
```

```
List<string> shuffledAnimals = new List<string>();
```

```
protected override void OnInitialized()
{
    SetUpGame();
}
```

```
private void SetUpGame()
{

     shuffledAnimals = animalEmoji
         .OrderBy(item => Random.Shared.Next())
         .ToList();

}
```

What it does

Create a second list to store the shuffled emoji

Create copies of the animal emoji, shuffle them, and store them in the shuffledAnimals list

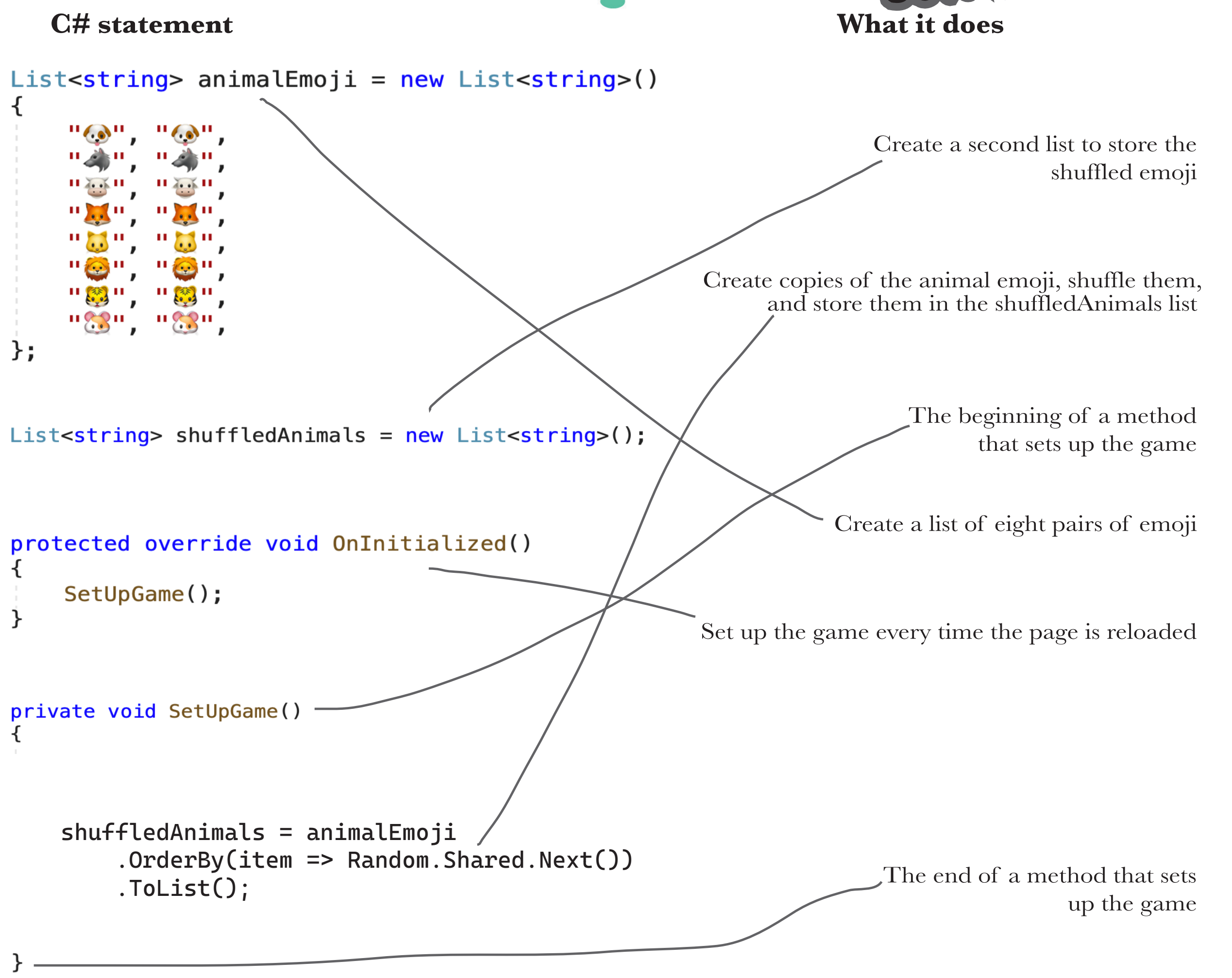
The beginning of a method that sets up the game

Create a list of eight pairs of emoji

Set up the game every time the page is reloaded

The end of a method that sets up the game

WHO DOES WHAT? solution



C# statement

What it does

```
List<string> animalEmoji = new List<string>()
{
    "🐶", "🐶",
    "🐺", "🐺",
    "🐮", "🐮",
    "🦊", "🦊",
    "🐱", "🐱",
    "🦁", "🦁",
    "🐯", "🐯",
    "🐹", "🐹",
};
```

Create a second list to store the shuffled emoji

Create copies of the animal emoji, shuffle them, and store them in the shuffledAnimals list

```
List<string> shuffledAnimals = new List<string>();
```

The beginning of a method that sets up the game

Create a list of eight pairs of emoji

```
protected override void OnInitialized()
{
    SetUpGame();
}
```

Set up the game every time the page is reloaded

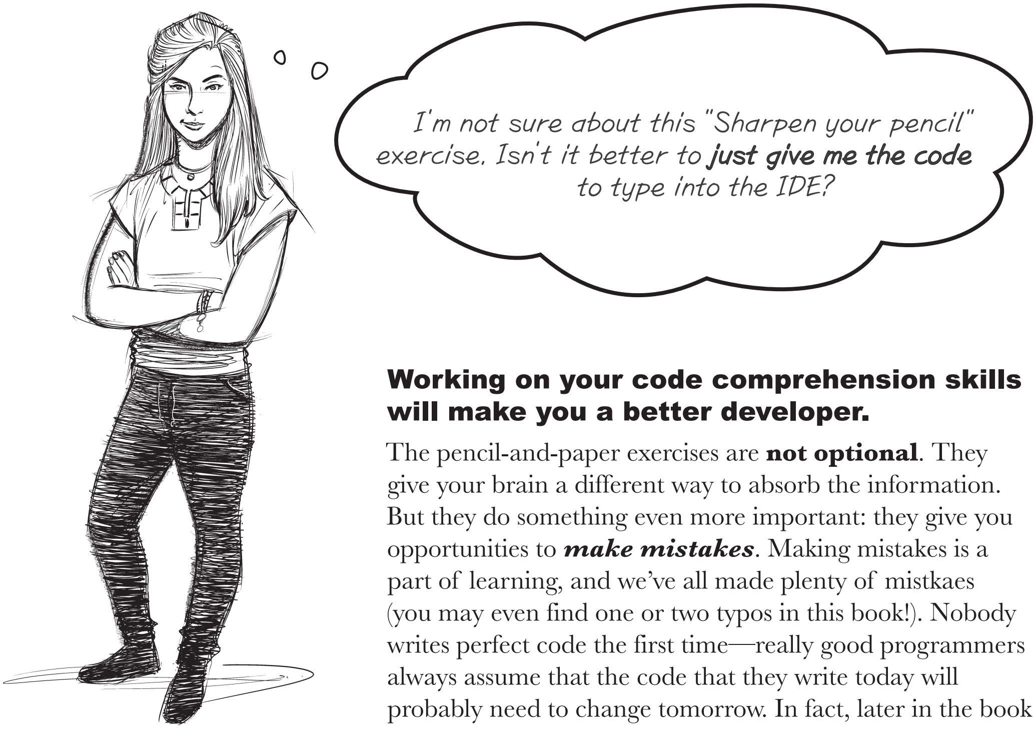
```
private void SetUpGame()
{


    shuffledAnimals = animalEmoji
        .OrderBy(item => Random.Shared.Next())
        .ToList();


}
```

The end of a method that sets up the game

MINI Sharpen your pencil

Here's a pencil-and-paper exercise that will help you really 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 all of the C# code 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 this "Sharpen your pencil" exercise. 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'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.
- .**Blazor Web Apps** let you build rich interactive web applications using C# code and HTML markup.
- C# is made up of **statements** grouped into **methods**.
- A **foreach loop** in a Razor page lets you repeat a block of HTML code for each element in a list.
- Visual Studio can **run your Blazor app** in debugging mode, opening a browser to display your app.Razor lets you add C# code directly into your HTML markup. Razor page files end with the .razor extension.
- Use an **@** to embed your C# code in a Razor page.
- User interfaces for Blazor apps are designed in **HTML**, the markup language used to design web pages.
- Visual Studio's AI-assisted **IntelliSense** and **IntelliCode** help you enter code more quickly.

My project has a lot of code already! Wouldn't it be dreamy if there was an easy way for me to save everything I've done someplace where I can **save my code**, **share it**, and **always find it** any time I want?

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 could see a history of all the changes you made, right?

Git can help you do all of those things!

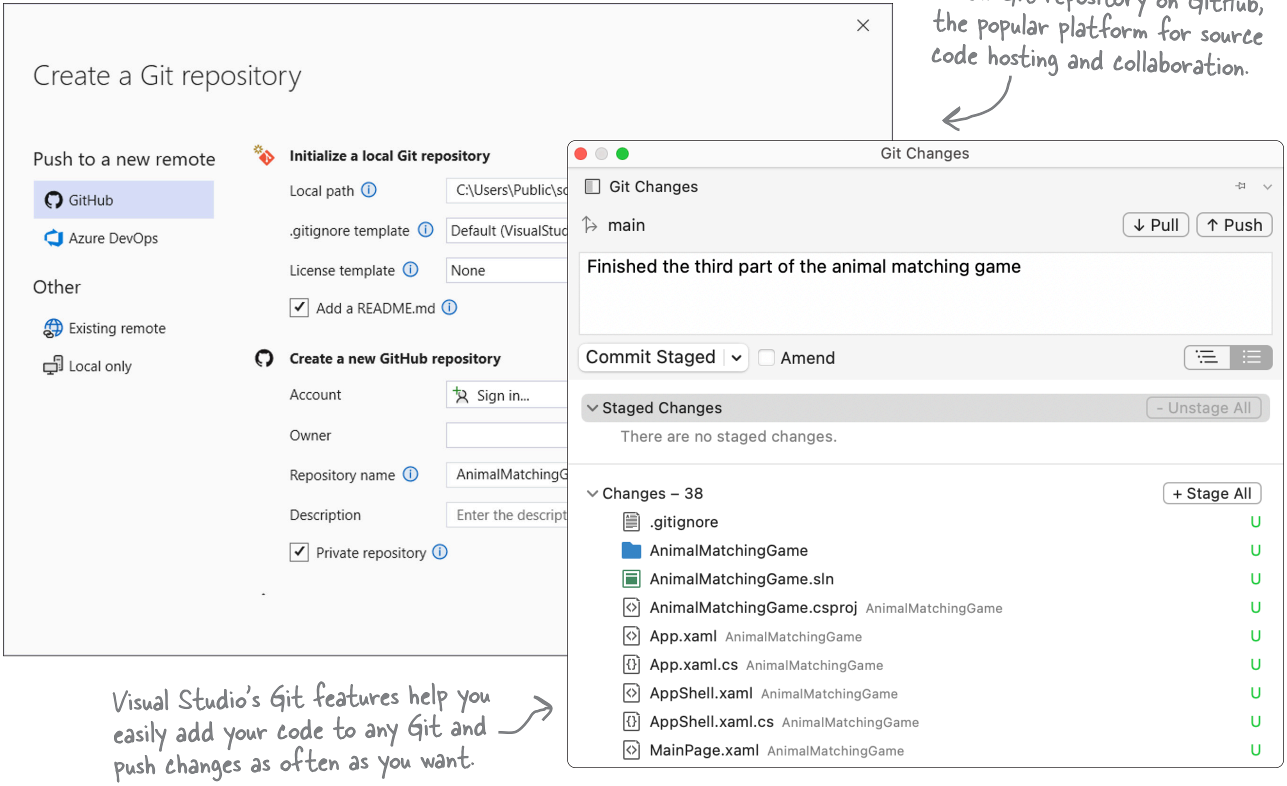
Here are just a few things Git can do for you

- It can save your files somewhere that you can access them from 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 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 support** 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.



Create a Git repository

Push to a new remote
- GitHub
- Azure DevOps

Other
- Existing remote
- Local only

Initialize a local Git repository

Local path ⓘ: C:\Users\Public\sc

.gitignore template ⓘ: Default (VisualStud

License template ⓘ: None

☑ Add a README.md ⓘ

Create a new GitHub repository

Account: Sign in...

Owner:

Repository name ⓘ: AnimalMatchingG

Description: Enter the descript

☑ Private repository ⓘ

Git Changes

main — ↓ Pull — ↑ Push

Finished the third part of the animal matching game

Commit Staged — □ Amend

Staged Changes — - Unstage All

There are no staged changes.

Changes – 38 — + Stage All

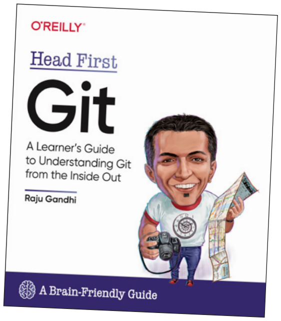
File	Project	Status
.gitignore		U
AnimalMatchingGame		U
AnimalMatchingGame.sln		U
AnimalMatchingGame.csproj	AnimalMatchingGame	U
App.xaml	AnimalMatchingGame	U
App.xaml.cs	AnimalMatchingGame	U
AppShell.xaml	AnimalMatchingGame	U
AppShell.xaml.cs	AnimalMatchingGame	U
MainPage.xaml	AnimalMatchingGame	U

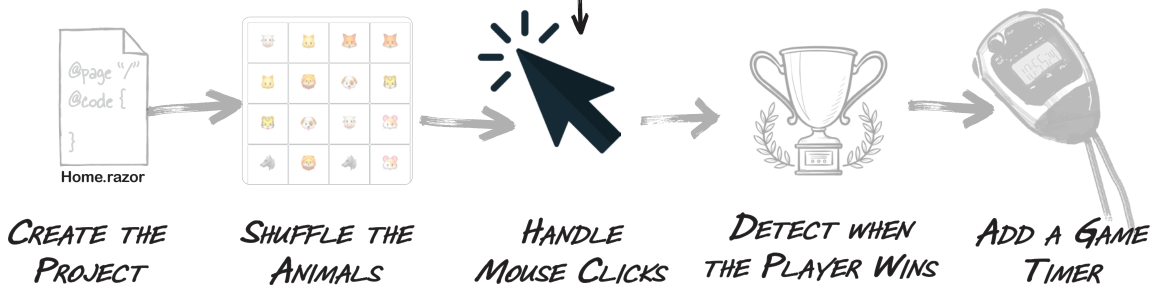
Visual Studio's Git features help you easily add your code to any Git and push changes as often as you want.

We recommend that you *create a GitHub account* and use it to 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.](https://github.com/head-first-csharp/fifth-edition)

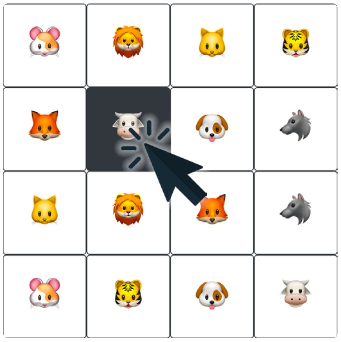
We'll give you everything you need to use Visual Studio to save and share your projects. But there is a lot more that you can do with Git, especially if you're working with large teams! If you're fascinated by what you see and want to do a deep dive into Git, check out *Head First Git* by Raju Gandhi.





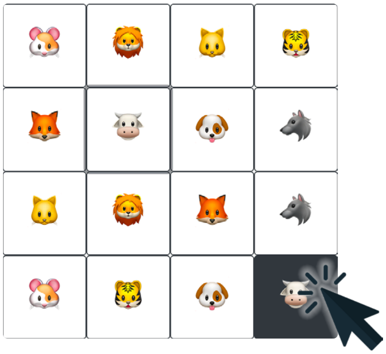
Add C# code to handle mouse clicks

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:



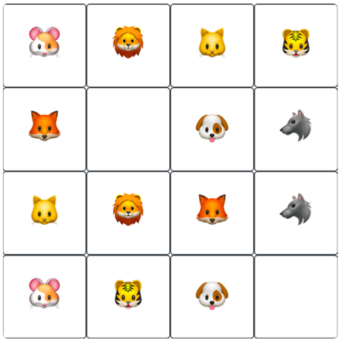
The player clicks the first button.

The player clicks buttons in pairs. When they click the first button, the game keeps track of that particular button's animal.



The player clicks the second button.

The game looks at the animal on the second button and compares it against the 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.

Before you go on, add the following line of code to the top of your Home.razor file:

`@rendermode InteractiveServer` ← Add this to your file!

In a Blazor app, adding **`@rendermode InteractiveServer`** to your Razor file instructs it to render the component interactively on the server, enabling real-time updates and interactions. This makes sure that actions like mouse clicks and timer ticks work. They’re processed on the server and then sent back to your browser for seamless updates.

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 answers. ***It’s OK 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.

```
string lastAnimalFound = string.Empty;

private void ButtonClick(string animal)
{
    if (lastAnimalFound == string.Empty)
    {
        lastAnimalFound = animal;
    }
    else if (lastAnimalFound == animal)
    {
        lastAnimalFound = string.Empty;

        shuffledAnimals = shuffledAnimals
            .Select(a => a.Replace(animal, string.Empty))
            .ToList();
    }
    else
    {
        lastAnimalFound = string.Empty;
    }
}
```

1. What do these lines of code do?

____

____

____

____

3. What does this block of code do?

____

____

____

____

____

____

____

____

____

4. What do the last lines of the method starting with `else` and going to the end do?

____

____

____

____

Sharpen your pencil Solution

We’ve asked you a few questions about what the code does. Try writing down the answers. ***It’s OK 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.

```
string lastAnimalFound = string.Empty;

private void ButtonClick(string animal)
{
    if (lastAnimalFound == string.Empty)
    {
        // First selection of the pair. Remember it.
        lastAnimalFound = animal;
    }
    else if (lastAnimalFound == animal)
    {
        // Match found! Reset for next pair.
        lastAnimalFound = string.Empty;

        // Replace found animals with empty string to hide them.
        shuffledAnimals = shuffledAnimals
            .Select(a => a.Replace(animal, string.Empty))
            .ToList();
    }
    else
    {
        // User selected a pair that doesn't match.
        // Reset selection.
        lastAnimalFound = string.Empty;
    }
}
```

1. What do these lines of code do?

When the player clicks on the first animal in a pair, these lines of code keep track of which animal the player clicked.

3. What does this block of code do?

This block of code is run when the player successfully clicks on a matching animal. If the animals match, resets for the next pair. Then it goes through clears the matching animals in the list.

4. What do the last lines of the method starting with `else` and going to the end do?

If the player clicks on a second animal that doesn’t match the first, it resets to wait for a first click.

Add the event handler and hook it up to the buttons

Go ahead and **add all of the above code** to your Razor file.

Then modify your buttons to **call the ButtonClick method** when clicked:

```
@foreach (var animal in animalEmoji)
{
    <div class="col-3">
        <button @onclick="@(() => ButtonClick(animal))"
                type="button" class="btn btn-outline-dark">
            <h1>@shuffledAnimals</h1>
        </button>
    </div>
}
```

The lines starting with // are comments. They don’t do anything—they’re only there to make the code easier to understand. We added them to help you read the code more easily.

When we ask you to update one thing in a block of code, we might make the rest of the code a lighter shade and make the part of the code you change boldface.

Your Event Handler Up Close

Let's take a closer look at how that event handler works. We've matched up the code from the event handler against our earlier explanation of how the game detects mouse clicks. Look at the code below and compare it with the code that you just typed into the IDE. See if you can follow along—it's OK if you don't get 100% of it, just try to follow the general idea of how the code that you just added fits together. This is a useful exercise for ramping up your C# comprehension skills.

The player clicks the first button.

This code checks to see if this is the first button clicked. If it is, it uses `lastAnimalFound` to keep track of the button's animal.

```
if (lastAnimalFound == string.Empty)
{
    lastAnimalFound = animal;
}
```

Are you clicking on the buttons, but your app isn't responding? Make sure you added
@rendermode InteractiveServer
to the top of your Home.razor file.

The player clicks the second button.

The statements between the opening **{** and closing **}** brackets only execute if the player clicked on a button whose animal matches the last one clicked.

```
else if (lastAnimalFound == animal)
{



}
```

The game checks for a match.

This C# code is only run if the second animal matches the first one. It goes through the shuffled list of animal emoji and replaces the ones that match the pair that the player found with blanks.

```
shuffledAnimals = shuffledAnimals
  .Select(a => a.Replace(animal, string.Empty))
  .ToList();
```

You'll find this statement in the code twice: in the section that's run if the second animal the player clicked matches the first, and in the section that's run if the second animal doesn't match. It blanks out the last animal found to reset the game so the next button click is the first of the pair. →

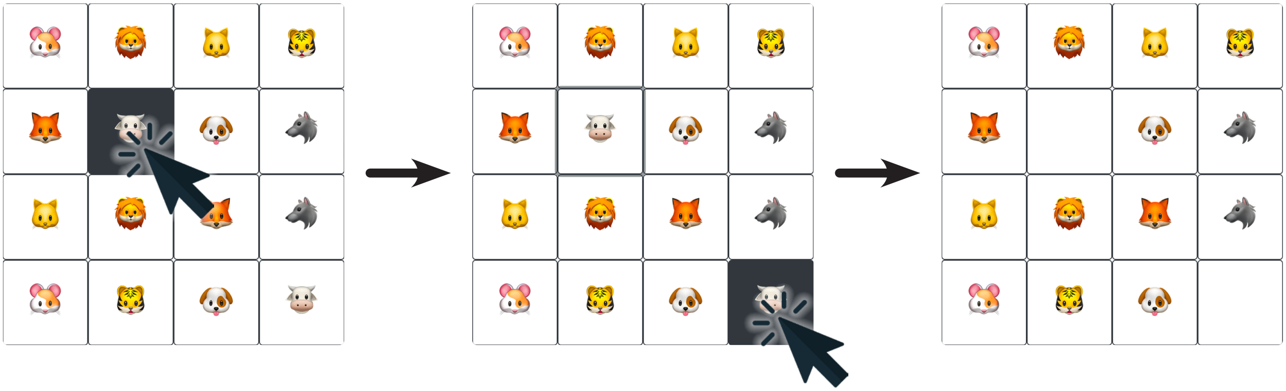
```
lastAnimalFound = string.Empty;
```

Uh-oh—there's a bug in this code! Can you spot it? We'll track it down and fix it in the next section.

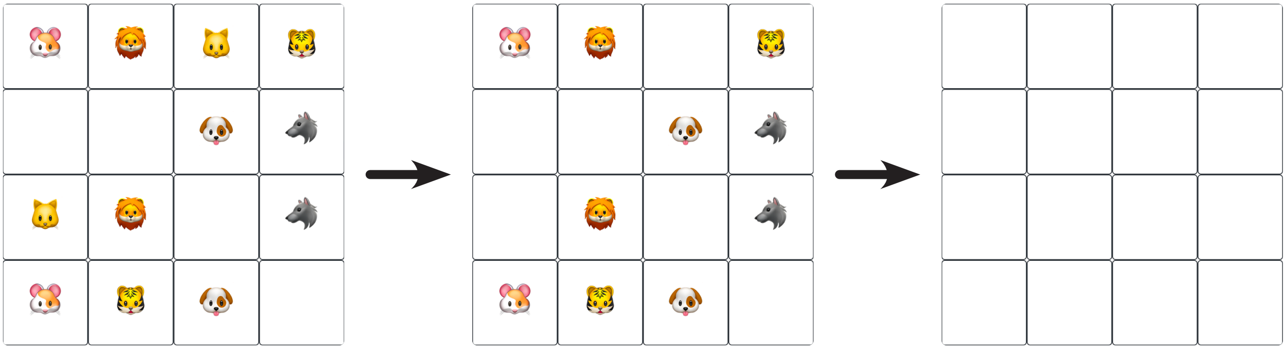
Are your buttons not clicking? Make sure you added the `@rendermode InteractiveServer` line to the top of your Home.razor file.

Test your event handler

Run your app again. When it comes up, test your event handler by clicking on a button, then clicking on the button with the matching emoji. They should both disappear.



Click on another, then another, then another. You should be able to keep clicking on pairs until all of the buttons are blank. Congratulations, you found all the pairs!



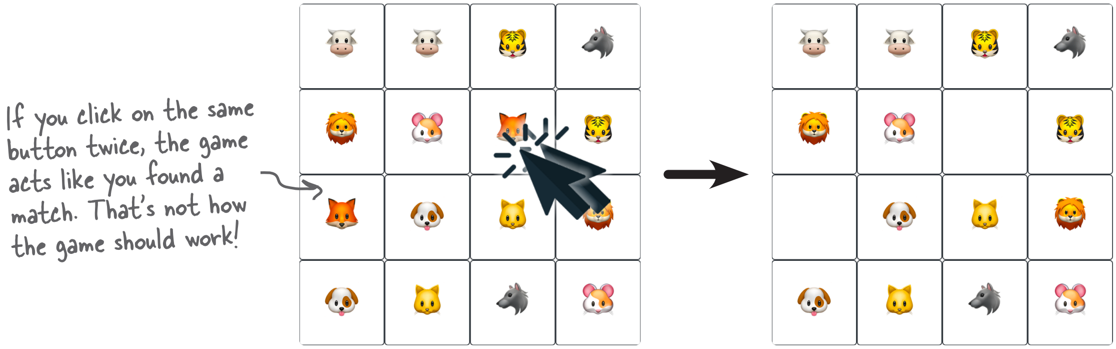
If your game doesn't work the way it should or you don't see the bug on this page, go back and check the code you entered against the code in the book. It's really easy to overlook a typo. Finding those issues is a good use of your time, because spotting errors in your code is a really good developer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

Uh-oh—there's a bug in the 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.

But what happens if you click on the same button twice?

Reload the page in your browser to reset the game. But this time instead of finding a pair, **click twice on the same button**. Hold on—***there's a bug in the game!*** It should have ignored the click, but instead, it acted like you got a match.



If you click on the same button twice, the game acts like you found a match. That's not how the game should work!

When you double-clicked on the same button, your game **removed both of the animals in the pair**. Wait, what!? 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 detective show, you know that 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**.

Use the debugger to troubleshoot the problem

You might have 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—everything in your program happens for a reason—but not every bug is easy to track down.

Understanding a bug is the first step in fixing it. 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!)

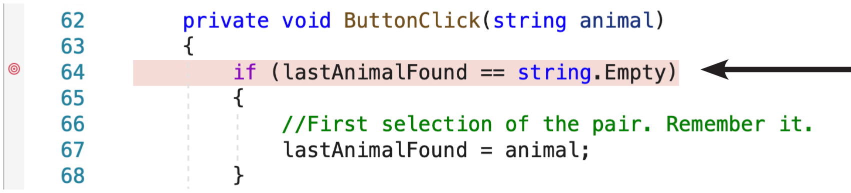
1. **Think about what's going wrong.**

   The first thing to notice is that your bug is **reproducible**: any time you click on the same button twice, it always acts like you clicked a matching pair.

   The second thing to notice is that you have a **pretty good idea** where the bug is. The problem only happened *after* you added the code to handle the Click event, so that's a great place to start.

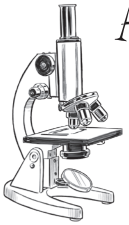
2. **Add a breakpoint to the Click event handler code that you just wrote.**

   Click on the first line of the ButtonClick method and **choose Run >> Toggle Breakpoint** (⌘\) from the menu. The line will change color and you'll see a dot in the left margin:



```
        private void ButtonClick(string animal)
        {
            if (lastAnimalFound == string.Empty)
            {
                //First selection of the pair. Remember it.
                lastAnimalFound = animal;
            }
```

When a breakpoint is set on a line, the IDE changes its background color and displays a dot in the left margin.

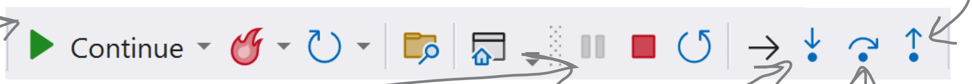


Anatomy of the Debugger

When your app is paused in the debugger—that's called "breaking" the app—the Debug controls show up in the toolbar. You'll get plenty of practice using them throughout the book, so you don't need to memorize what they do. For now, just read the descriptions we've written, hover your mouse over them to see the tooltips, and check the Run or Debug menu to see their corresponding shortcut keys (like F10 for Step Over).

The Continue Execution button starts your app running again.

The Step Out button finishes executing the current method and breaks on the line after the one that called it.



You can use the Pause Execution button to pause your app when it's running.

The Step Into button also executes the next statement, but if that statement is a method it only executes the first statement inside the method.

The Step Over button executes the next statement. If it's a method, it runs the whole thing.

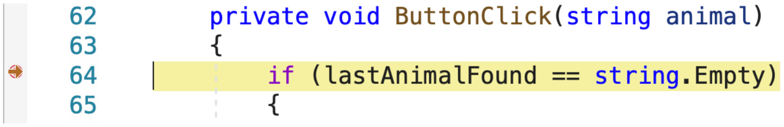
VSCode looks a little different but uses very similar icons to do the same things.

Keep debugging your event handler

Now that your breakpoint is set, use it to get a handle on what's going on with your code.

3 Click on an animal to trigger the breakpoint.

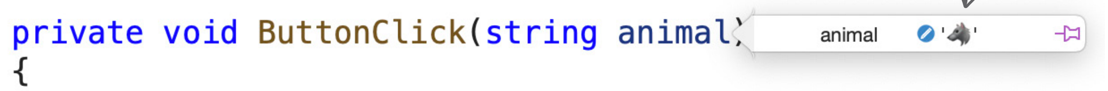
If your app is already running, stop it and close all browser windows. Then **run your app** again and **click any animal button**. Visual Studio should pop into the foreground. The line where you toggled the breakpoint should now be highlighted in a different color:



```
        private void ButtonClick(string animal)
        {
            if (lastAnimalFound == string.Empty)
            {
```

Move your mouse to the first line of the method, which starts `private void`, and **hover your cursor over `animal`**. A small window will pop up that shows you the animal that you clicked on:

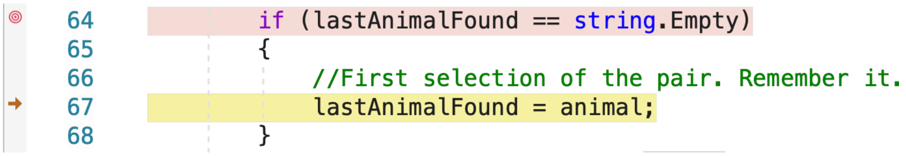
Hover over "animal" to see the emoji that you clicked.



```
private void ButtonClick(string animal)
{
```

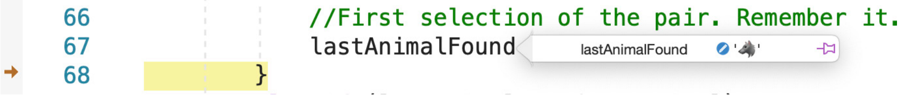
animal '🐺'

Press the **Step Over** button or choose Run >> Step Over (⇧⌘O) from the menu. The highlight will move down to the **{** line. Step over again to move the highlight to the next statement:



```
            if (lastAnimalFound == string.Empty)
            {
                //First selection of the pair. Remember it.
                lastAnimalFound = animal;
            }
```

Step over one more time to execute that statement, then hover over `lastAnimalFound`:



```
                //First selection of the pair. Remember it.
                lastAnimalFound
            }
```

lastAnimalFound '🐺'

The statement that you stepped over set the value of `lastAnimalFound` so it matches `animal`.

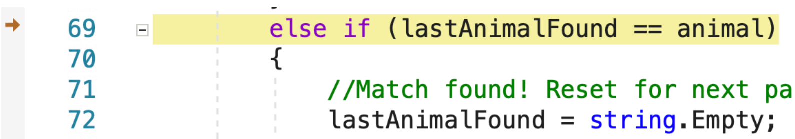
That's how the code keeps track of the first animal that the player clicked.

4 Continue execution.

Press the **Continue Execution** button or choose Run >> Continue Debugging (⌘↩) from the menu. Switch back to the browser—your game will keep going until it hits the breakpoint again.

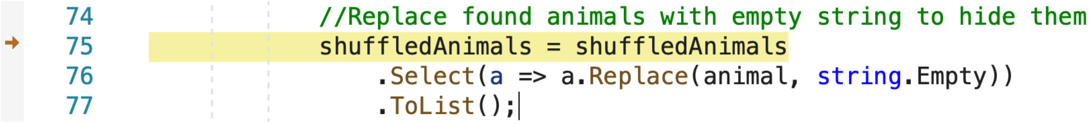
5 Click the matching animal in the pair.

Find the button with the matching emoji and **click it**. The IDE will trigger the breakpoint and pause the app again. Press **Step Over**—it will skip the first block and jump to the second:



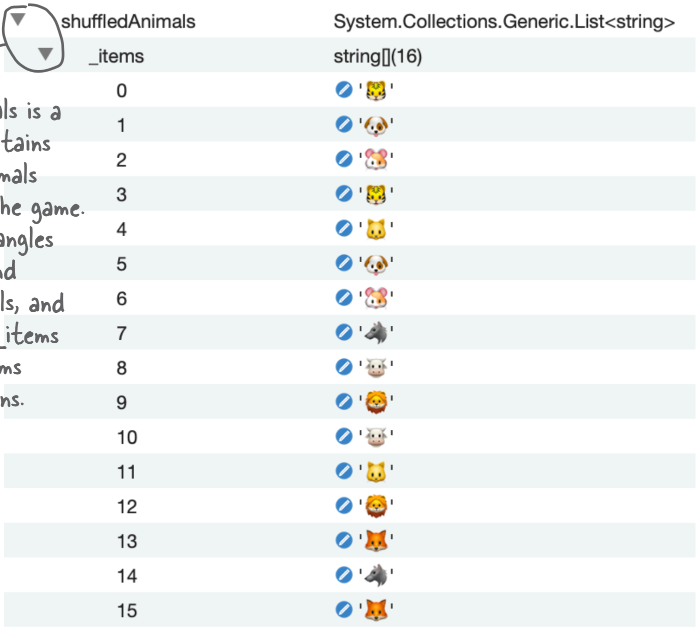
```
        else if (lastAnimalFound == animal)
        {
            //Match found! Reset for next pair.
            lastAnimalFound = string.Empty;
```

Hover over `lastAnimalFound` and `animal`—they should both have the same emoji. That's how the event handler knows that you found a match. **Step over three more times**:



```
            //Replace found animals with empty string to hide them
            shuffledAnimals = shuffledAnimals
                .Select(a => a.Replace(animal, string.Empty))
                .ToList();
```

Now **hover over `shuffledAnimals`**. You'll see several items in the window that pops up. Click the triangle next to `shuffledAnimals` to expand it, then **expand `_items`** to see all the animals:

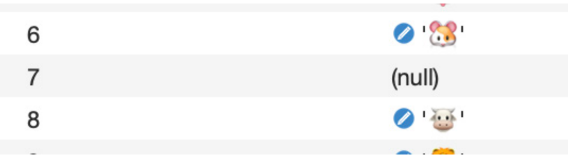


▼ shuffledAnimals	System.Collections.Generic.List<string>
▼ _items	string[](16)
0	'🐯'
1	'🐶'
2	'🐹'
3	'🐯'
4	'🐱'
5	'🐶'
6	'🐹'
7	'🐺'
8	'🐮'
9	'🦁'
10	'🐮'
11	'🐱'
12	'🦁'
13	'🦊'
14	'🐺'
15	'🦊'

shuffledAnimals is a List that contains all of the animals currently in the game. Use these triangles to first expand shuffledAnimals, and then expand _items to see the items that it contains.

Once you've expanded shuffledAnimals and _items, you can use the debugger to inspect the contents of the List. You'll learn more about what a List is and how it works in Chapter 8.

Continue Execution to resume your game, then **click another matched pair** of animals to trigger your breakpoint again and return to the debugger. Then **hover over `shuffledAnimals` again** and look at its items. There are now two (*null*) values where the matched emoji used to be:



6	'🐹'
7	(null)
8	'🐮'

We've sifted through a lot of evidence and gathered some important clues. What do you think is causing the problem?

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

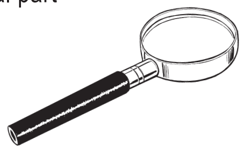
You've probably heard the word "bug" before.

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 throughout 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 a 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:

1. Every time you click the button, the click event handler runs.
2. The event handler uses `animal` to figure out which animal you clicked first.
3. The event handler uses `lastAnimalFound` to figure out which animal you clicked second.
4. If `animal` equals `lastAnimalFound`, it decides it has a match and removes the matching animals from the list.

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 decide that it found a match when you double-click on an animal?

You'll get a lot of practice sleuthing out bugs. It's a really useful developer skill.

Sleuth it Out

"Elementary, my dear Watson."

What happens if you click the same animal button twice? Let's find out! **Repeat the same steps you just did**, except this time **click the same animal twice**. Watch what happens when you get to step 5.

Hover over `animal` and `lastAnimalFound`, just like you did before. They're the same! That's because the event handler ***doesn't have a way to distinguish between different buttons with the same animal***.

...and fix the bug!

Now that we know what's causing the bug, we know how to fix it: give the event handler a way to distinguish between the two buttons with the same emoji.

First, **make these changes** to the ButtonCick event handler (make sure you don't miss any changes):

```
string lastAnimalFound = string.Empty;
string lastDescription = string.Empty;

private void ButtonClick(string animal, string animalDescription)
{
    if (lastAnimalFound == string.Empty)
    {
        // First selection of the pair. Remember it.
        lastAnimalFound = animal;
        lastDescription = animalDescription;
    }
    else if ((lastAnimalFound == animal) && (animalDescription != lastDescription))
```

Add this line of code just above the beginning of the ButtonClick event handler method.

Make these changes inside the method.

Then **replace the `foreach` loop** with a different kind of loop, a `for` loop—this for loop counts the animals:

```
<div class="row">
   @for (var animalNumber = 0; animalNumber < shuffledAnimals.Count; animalNumber++)
   {
     var animal = shuffledAnimals[animalNumber];
     var uniqueDescription = $"Button #{animalNumber}";

     <div class="col-3">
       <button @onclick="@(() => ButtonClick(animal, uniqueDescription))"
               type="button" class="btn btn-outline-dark">@animal</button>
```

Replace the line that starts with @for with this line

Then add these two lines after the {.

Finally, modify this line.

Now debug through the app again, just like you did before. This time when you click the same animal twice it will skip down to the end of the event handler. ***The bug is fixed!***

there are no Dumb Questions

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!

Q: You mentioned that I'm running a server and a web application. What did you mean by that?

A: When you run your app, the IDE starts up the browser that you selected. The address bar in the browser has a URL like https://localhost:5001/—if you **copy that URL** and paste it into the URL bar of **another browser**, that browser will run your game, too. That's because the browser is running a **web application**, or a web page that runs entirely inside your browser. Like any web page, it needs to be hosted on a web server.

Q: What web server is my browser connecting to?

A: Your browser is connecting to a server that's running *inside Visual Studio*. Click the Application Output button at the bottom of the IDE to open a window that shows you the output of whatever application is running—in this case, it's an application that includes the server that's hosting your web application. Scroll or search through that window to find the line that shows it listening for incoming browser connections:

```
Now listening on: https://localhost:5001
```

Q: When I press ⌘⇥ (Command-Tab) to switch between macOS apps, there are a bunch of instances of Edge or Chrome still open. What's happening?

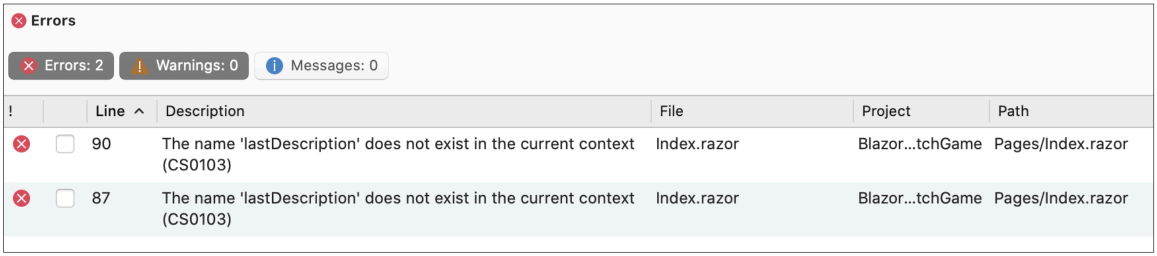
A: Every time you stop and restart your app in Visual Studio, it launches a new instance of the browser because it needs to establish a separate connection for debugging. You can connect other instances of a browser, but you can only debug with the browser that the IDE launched. You can test this yourself: start, stop, and restart your app in the IDE, then set a breakpoint. Only one of the browsers will actually pause when the breakpoint is hit.

Q: Blazor web apps seem a lot more complicated than console apps. Do they really work the same way?

A: Yes. When you get down to it, all C# code works the same way: one statement executes, then the next one, and then the next one. One reason web apps seem more complex is because some methods are only called when certain things happen, like when the page is loaded or the user clicks on a button. Once a method gets called, it works exactly like in a console app—and you can prove that to yourself by setting a breakpoint inside of it.

IDE Tip: The Errors Window

Unless you have a superhuman ability to enter code perfectly without a single typo, you've seen the Errors window at the bottom of the IDE. It pops up when you try to run your project but it has errors. Here's what it looked like when we tried to fix the bug, but accidentally included this typo: `string lsatDescription = string.Empty;`

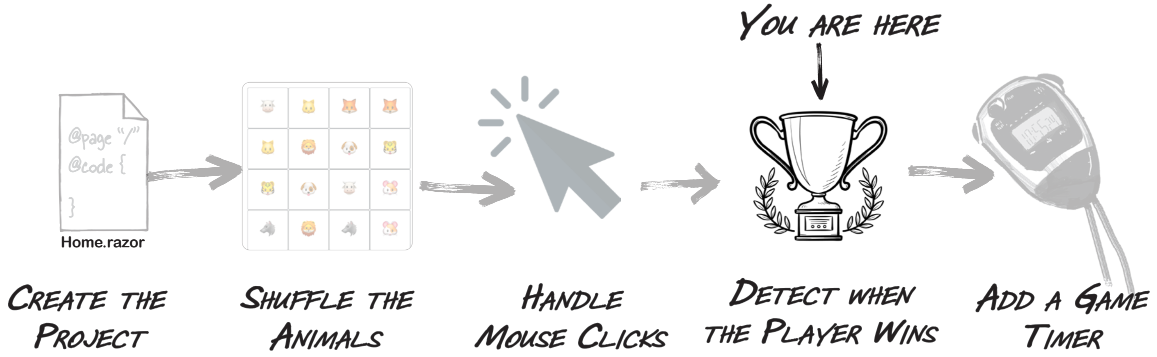


!		Line ^	Description	File	Project	Path
✕	□	90	The name 'lastDescription' does not exist in the current context (CS0103)	Index.razor	Blazor...tchGame	Pages/Index.razor
✕	□	87	The name 'lastDescription' does not exist in the current context (CS0103)	Index.razor	Blazor...tchGame	Pages/Index.razor

You can always check for errors by **building** your code, either by running it or choosing Build All (⌘B) from the Build menu. If the Errors window doesn't pop up, that means your code **builds**, which is what the IDE does to turn your code into a **binary**, or an executable file that macOS can run.

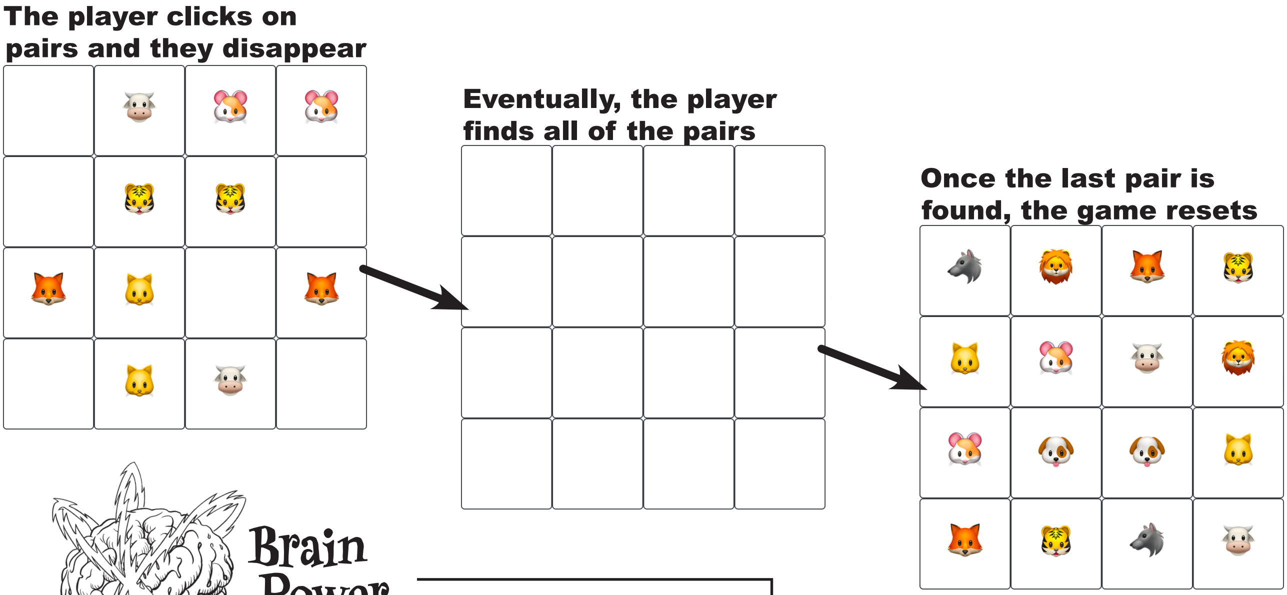
Let's add an error to your code. Go to the first line in your SetUpGame method, then add this on its own line: `Xyz`

Build your code. The IDE will open the Errors window with ✕ Errors: 1 at the top and one error. If you click elsewhere, the Errors window will disappear—but don't worry, you can always reopen it by clicking ✕ Errors at the bottom of the IDE.



Add code to reset the game when the player wins

The game is coming along—your player starts out with a grid full of animals to match, and they can click on pairs of animals that disappear when they're matched. But what happens when all of the matches are found? We need a way to reset the game so the player gets another chance.


The player clicks on pairs and they disappear
Eventually, the player finds all of the pairs
Once the last pair is found, the game resets

Brain Power

Take a minute and look through the C# code and HTML markup. What parts of it do you think you'll need to change to make it reset the game once the player has clicked all of the matched pairs?

When you see a Brain Power element, take a minute and really think about the question that it's asking.

Exercise

Here are four blocks of code to add to your app. Once each block is in the right place, the game will reset as soon as the player gets all of the matches.

```
int matchesFound = 0;
```

```
matchesFound = 0;
```

```
<div class="row">
  <h2>Matches found: @matchesFound</h2>
</div>
```

```
matchesFound++;
if (matchesFound == 8)
{
    SetUpGame();
}
```

Your job is to figure out where each of the four blocks goes. We've copied parts of the code for your game below and added four boxes, one for each block of code above. Can you figure out which block of code goes in each box?

```
<div class="container">
    <div class="row">
        @for (var animalNumber = 0; animalNumber < shuffledAnimals.Count; animalNumber++)
        {
            var animal = shuffledAnimals[animalNumber];
            var uniqueDescription = $"Button #{animalNumber}";

            <div class="col-3">
                <button @onclick="@(() => ButtonClick(animal, uniqueDescription))"
                        type="button" class="btn btn-outline-dark">
                    <h1>@animal</h1>
                </button>
            </div>
        }
    </div>

</div>
```

```
List<string> shuffledAnimals = new List<string>();

```

```
private void SetUpGame()
{
    shuffledAnimals = animalEmoji
        .OrderBy(item => Random.Shared.Next())
        .ToList();

}
```

This isn't a pencil-and-paper exercise—you should do this exercise by modifying your code in the IDE. When you see an Exercise with the running shoe icon in the corner, that's your cue to go back to the IDE and start writing C# code.

```
    else if ((lastAnimalFound == animal) && (animalDescription != lastDescription))
    {
        // Match found! Reset for next pair.
        lastAnimalFound = string.Empty;

        // Replace found animals with empty string to hide them
        shuffledAnimals = shuffledAnimals
            .Select(a => a.Replace(animal, string.Empty))
            .ToList();

    }
```

When you're doing a code exercise, it's not cheating to peek at the solution! We don't learn effectively if we're frustrated—it's easy to get stuck on one little thing, and the solution can help you get past it.

Exercise Solution

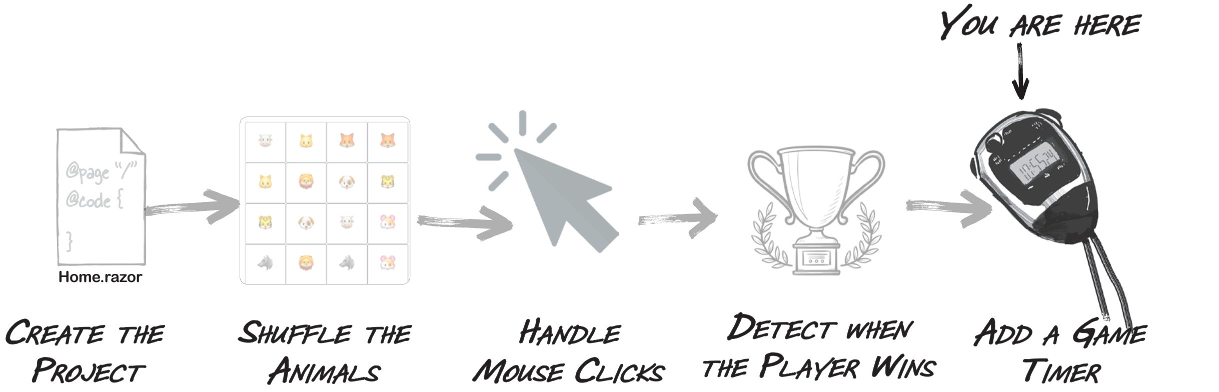
Here's what the code looks like with each block of code in the correct place. If you haven't already, **add all four blocks of code to your game** to make it reset when the player finds all the matches.

```
<div class="container">
    <div class="row">
        @for (var animalNumber = 0; animalNumber < shuffledAnimals.Count; animalNumber++)
        {
            var animal = shuffledAnimals[animalNumber];
            var uniqueDescription = $"Button #{animalNumber}";

            <div class="col-3">
                <button @onclick="@(() => ButtonClick(animal, uniqueDescription))"
                        type="button" class="btn btn-outline-dark">
                    <h1>@animal</h1>
                </button>
            </div>
        }
    </div>
    <div class="row">
      <h2>Matches found: @matchesFound</h2>
    </div>
</div>
```

```
List<string> shuffledAnimals = new List<string>();
int matchesFound = 0;
```

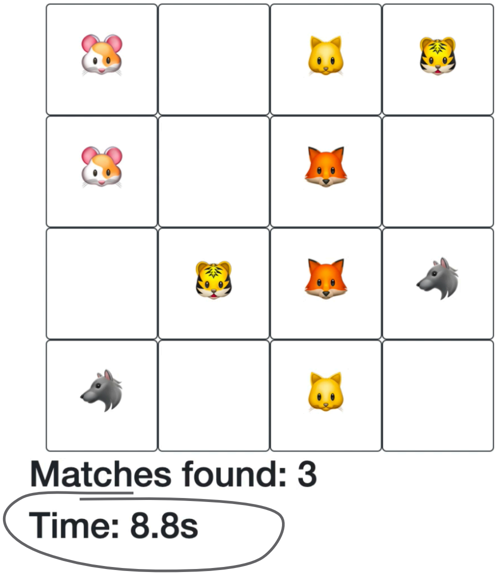
```
private void SetUpGame()
{

    shuffledAnimals = animalEmoji
        .OrderBy(item => Random.Shared.Next())
        .ToList();
    matchesFound = 0;
}
```

```
    else if ((lastAnimalFound == animal) && (animalDescription != lastDescription))
    {
        // Match found! Reset for next pair.
        lastAnimalFound = string.Empty;

        // Replace found animals with empty string to hide them
        shuffledAnimals = shuffledAnimals
            .Select(a => a.Replace(animal, string.Empty))
            .ToList();

        matchesFound++;
        if (matchesFound == 8)
        {
            SetUpGame();
        }
    }
```



Finish the game by adding a timer

Your animal matching 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.



Matches found: 3

Time: 8.8s

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

Add this!

1. Start by finding this line at the very top of the *Home.razor* file: `@page "/"`

   Add this line just below it—you need it in order to use a Timer in your C# code:

```
@using System.Timers
```

2. You'll need to update the HTML markup to display the time. Add this just below the first block that you added in the exercise:

```
   </div>
    <div class="row">
        <h2>Matches found: @matchesFound</h2>
    </div>
    <div class="row">
        <h2>Time: @timeDisplay</h2>
    </div>
</div>
```

3. Your page will need a timer. It will also need to keep track of the elapsed time:

```
List<string> shuffledAnimals = new List<string>();
int matchesFound = 0;
Timer timer;
int tenthsOfSecondsElapsed = 0;
string timeDisplay;
```

4. You need to tell the timer how frequently to "tick" and what method to call. You'll do this in the OnInitialized method, which is called once after the page is loaded:

```
protected override void OnInitialized()
{
    timer = new Timer(100);
    timer.Elapsed += Timer_Elapsed;

    SetUpGame();
}
```

5. Reset the timer when you set up the game:

```
private void SetUpGame()
{
    shuffledAnimals = animalEmoji
        .OrderBy(item => Random.Shared.Next())
        .ToList();

    matchesFound = 0;
    tenthsOfSecondsElapsed = 0;
}
```

6 You need to stop and start your timer. Add this line of code near the top of the ButtonClick method to start the timer when the player clicks the first button:

```
if (lastAnimalFound == string.Empty)
{
    // First selection of the pair. Remember it.
    lastAnimalFound = animal;
    lastDescription = animalDescription;

    timer.Start();
}
```

And finally, add these two lines further down in the ButtonClick method to stop the timer and display a "Play Again?" message after the player finds the last match:

```
matchesFound++;
if (matchesFound == 8)
{
    timer.Stop();
    timeDisplay += " - Play Again?";

    SetUpGame();
}
```

7 Finally, your timer needs to know what to do each time it ticks. Just like buttons have Click event handlers, timers have Tick event handlers: methods that get executed every time the timer ticks.

Add this code at the very bottom of the page, just above the closing bracket **}**:

```
private void Timer_Elapsed(object? sender, ElapsedEventArgs e)
{
    InvokeAsync(() =>
    {
        tenthsOfSecondsElapsed++;
        timeDisplay = (tenthsOfSecondsElapsed / 10F)
            .ToString("0.0s");
        StateHasChanged();
    });
}
```

The timer starts when the player clicks the first animal and stops when the last match is found. This doesn't fundamentally change the way the game works, but makes it more exciting.

Having trouble getting back to the Solution Explorer in Visual Studio Code? Just choose Explorer (Ctrl+Shift+E) from the View menu.

Clean up the navigation menu

Your game is working! But did you notice that there are other pages in your app? Try clicking on "Counter" or "Fetch data" in the navigation menu on the left side. When you created the Blazor WebAssembly App project, Visual Studio added these additional sample pages. You can safely remove them.

Expand the **Layout folder** underneath Components and **double-click *NavMenu.razor***. Find this line:

```
<a class="navbar-brand" href="">BlazorMatchGame</a>
```

and **replace it with this**:

```
<a class="navbar-brand" href="">Animal Matching Game</a>
```

Then **delete these lines**:

Make sure you only delete the second and third <div> blocks for Counter and Weather.

~~<div class="nav-item px-3">~~
~~   <NavLink class="nav-link" href="counter">~~
~~   <span class="bi bi-plus-square-fill-nav-menu" aria-hidden="true"></span> Counter~~
~~   </NavLink>~~
~~</div>~~

~~<div class="nav-item px-3">~~
~~   <NavLink class="nav-link" href="weather">~~
~~   <span class="bi bi-list-nested-nav-menu" aria-hidden="true"></span> Weather~~
~~   </NavLink>~~
~~</div>~~

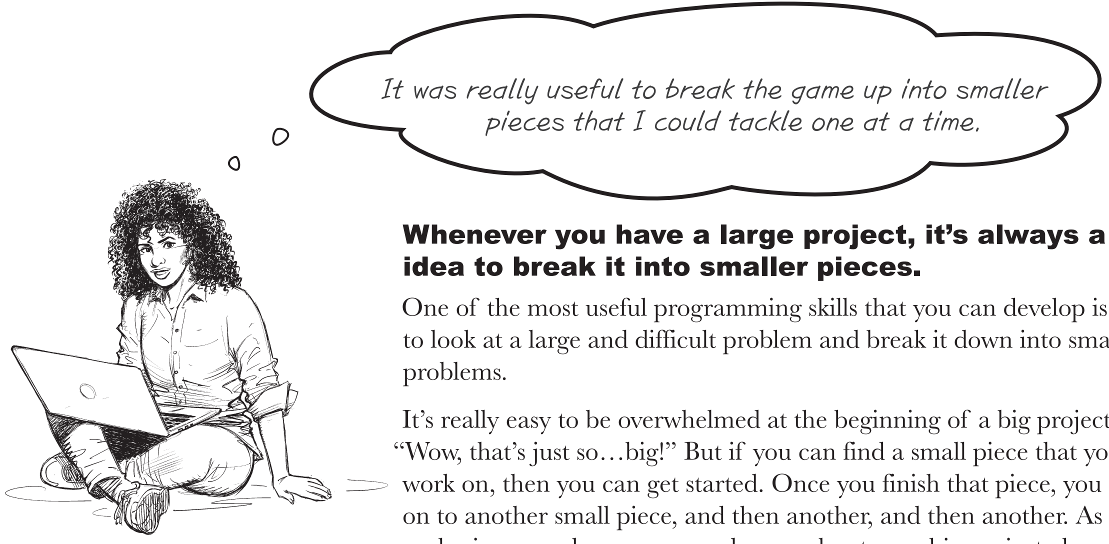
Hold down Control or ⌘ (Command) and **click to multiselect *Counter.razor* and *Weather.razor*** in the Pages folder. Right-click on one of them and **choose Delete** from the menu to delete the two files.

Finally, go back to your Home.razor file and change the page title:

```
<PageTitle>Animal Matching Game</PageTitle>
```

Now your app doesn't have any extraneous files that it doesn't need, and the navigation bar on the side doesn't include links to pages that no longer exist.

And now your game is done!


It was really useful to break the game up into smaller pieces that I could tackle one at a time.

Whenever you have a large project, it's always a good idea to break it into smaller pieces.

One of the most useful programming skills that you can develop is the ability to look at a large and difficult problem and break it down into smaller, easier problems.

It's really easy to be overwhelmed at the beginning of a big project and think, "Wow, that's just so…big!" But if you can find a small piece that you can work on, then you can get started. Once you finish that piece, you can move on to another small piece, and then another, and then another. As you build each piece, you learn more and more about your big project along the way.

Even better ifs...

Your game is pretty good! But 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

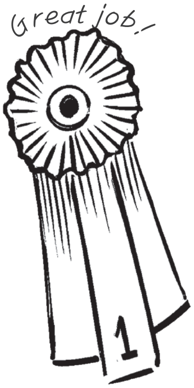
Can you think of your own "even better if" improvements for the game? This is a great exercise—take a few minutes and write down at least three improvements to the animal matching game.

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.

Bullet Points

- An **event handler** is a method that your application calls when a specific event like a mouse click, page reload, or timer tick happens.
- The IDE's **Errors window** shows any errors that prevent your code from building.
- **Timers** execute Tick event handler methods over and over again on a specified interval.
- **`foreach`** is a kind of loop that iterates through a collection of items.
- **`for`** is a kind of loop that can be used for counting.
- When your program has a **bug**, gather evidence and try to figure out what's causing it.
- Bugs are easier to fix when they're **reproducible**.
- The IDE's **debugger** lets you pause your app on a specific statement to help track down problems.
- Setting a **breakpoint** makes the debugger pause on the statement where the breakpoint is set.
- 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.

Just a quick reminder: we'll refer to Visual Studio or Visual Studio Code as "the IDE" a lot in this book.

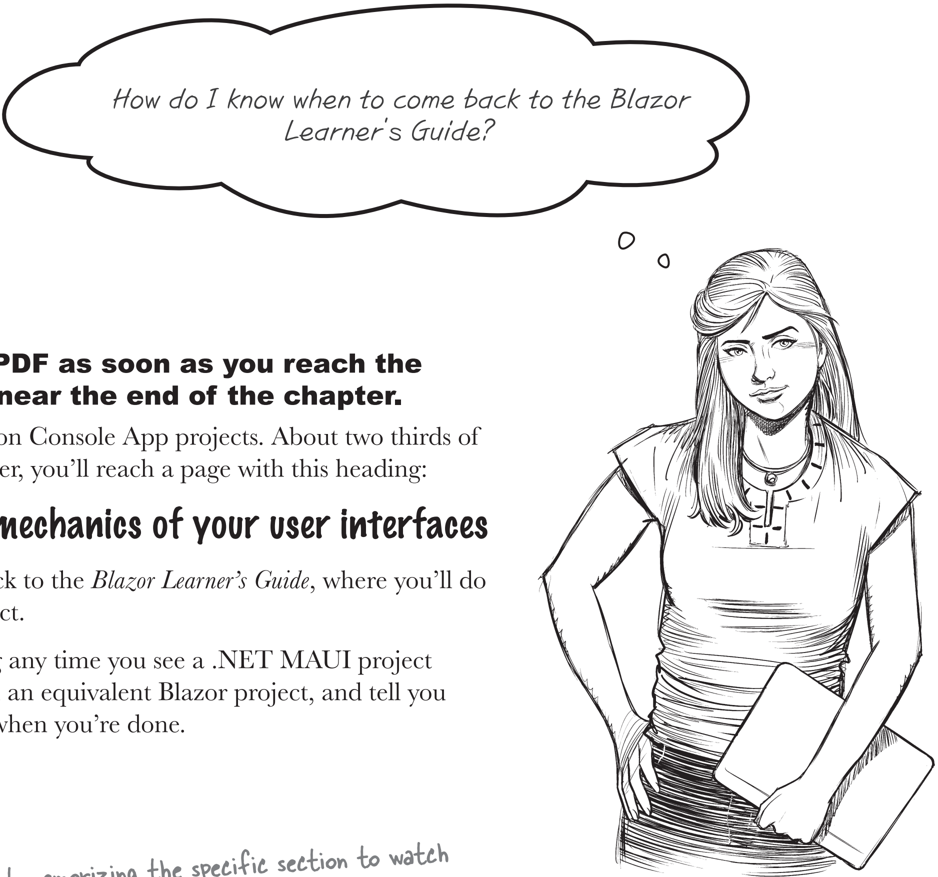


This is a great time to push your code to Git! Then you'll always be able to go back to your project if you want to reuse some of the code in it.

It looks like we're done with the project - and that's the end of Chapter 1. It's time to go back to the book, right?

That's right. You can pick up your learning at the start of Chapter 2.

Chapter 1 was all about getting used to writing code in Visual Studio or VSCode, and starting to get some important C# ideas into your brain. Now it's time to head back to the main book. Go to the **start of Chapter 2** and jump right back in.


How do I know when to come back to the Blazor Learner's Guide?

Come back to this PDF as soon as you reach the .NET MAUI project near the end of the chapter.

In Chapter 2, you'll work on Console App projects. About two thirds of the way through the chapter, you'll reach a page with this heading:

Controls drive the mechanics of your user interfaces

That's when you come back to the *Blazor Learner's Guide*, where you'll do an equivalent Blazor project.

You can do the same thing any time you see a .NET MAUI project in the book. We'll give you an equivalent Blazor project, and tell you where to pick up reading when you're done.

Don't worry about memorizing the specific section to watch out for. All you need to know is that

You can figure out what chapter you're in by looking here.

Controls drive the mechanics of your user interfaces

In the last chapter, you built a game using Button **controls**. But 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 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 control to let them do that—***and that choice affects how your user experiences the app***.

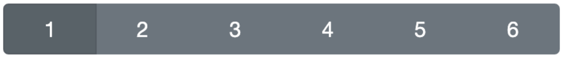
Meet some of the controls you'll use in this book

There's a Blazor project for most of the chapters in the book. 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 **user interface** (or **UI**)—or the way the window is laid out so the user can interact with it—of each app.

Here are some controls you'll see in Blazor applications.

★ A **text box** lets a user enter any text they want. But we need a way to make sure they're only entering numbers and not just any text.

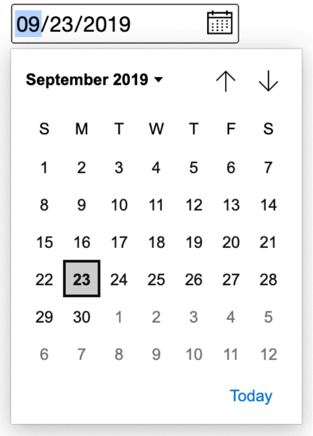
★ **Sliders** are used exclusively to choose a number. Phone numbers are just numbers, too, so *technically* you could use a slider to choose a phone number. Do you think that's a good choice?



★ **Radio buttons** let you restrict the user's choice. They often look like circles with dots in them, but you can style them to look like regular buttons too.

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

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.

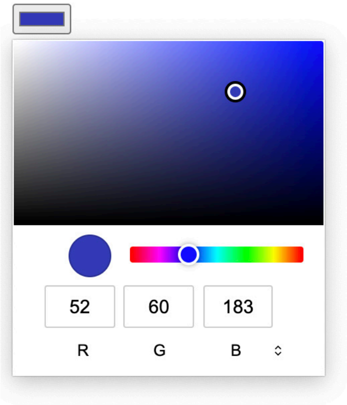


09/23/2019

September 2019

S	M	T	W	T	F	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	1	2	3	4	5
6	7	8	9	10	11	12

★ **Pickers** are controls that are specially built to pick a specific type of value from a list. For example, **date pickers** let you specify a date by picking its year, month, and day, and **color pickers** let you choose a color using a spectrum slider or by its numeric value.



R	G	B
52	60	183

Create a new Blazor Web App project

Earlier in this ***Blazor Learner's Guide***, you created a Blazor Web App project for your animal matching game. You'll do the same thing for this project, too.

> ***Here's a concise set of steps to follow to create a Blazor Web App project, change the title text for the main page, and remove the extra files that Visual Studio creates. We won't repeat this for every additional project in this guide—you should be able to follow these same instructions for all of the future Blazor Web App projects.***

1 **Create a new Blazor Web App project.**

Create a new project just like you did at the beginning of Chapter 1, except this time give it a different name. In this case, use the name **ExperimentWithControlsBlazor**.

2 **Delete the extra lines from Home.razor and set the PageTitle tag to match your app.**

When you create a Blazor Web App project, the Home.razor file contains lines that display "Hello, World!" and "Welcome to your new app." Delete those lines.

All of the projects in this book will use interactive controls, so add a `@rendermode InteractiveServer` line.

In Chapter 1 we learned that the <PageTitle> tag in the Razor file sets the page title that the browser displays in the tab. Change it to match the name of the app. Here's what your page will look like:

```
@rendermode InteractiveServer
@page "/"

<PageTitle>Experiment With Controls</PageTitle>
```

Modify Home.razor to delete the @rendermode line, set the page title so it matches your app, and delete the extra lines at the end.

3 **Remove the extra navigation menu options and their corresponding files.**

This is just like what you did at the end of the animal matching game project. Expand the Layout folder inside Coponents and **double-click on *NavMenu.razor***, then **delete these lines**:

```
~~<div class="nav-item px-3">~~
~~    <NavLink class="nav-link" href="counter">~~
~~        <span class="bi bi-plus-square-fill-nav-menu" aria-hidden="true"></span> Counter~~
~~    </NavLink>~~
~~</div>~~

~~<div class="nav-item px-3">~~
~~    <NavLink class="nav-link" href="weather">~~
~~        <span class="bi bi-list-nested-nav-menu" aria-hidden="true"></span> Weather~~
~~    </NavLink>~~
~~</div>~~
```

Finally, delete the ***Counter.razor*** and ***Weather.razor*** files in the Pages folder.

The layout of a Blazor Web App project may change if you're using a different version of .NET, so make sure you create your projects using .NET 8.0.

Create a page with a slider control

Many of your programs will need the user to input numbers, and one of the most basic controls to input a number is a **slider**, also known as a **range input**. Let's create a new Razor page that uses a slider to update a value.

1 Replace the Home.razor page.

Open *Home.razor* and **replace** all of its contents with **this HTML markup:**

Edit the Razor page, just like you did with the animal match game in Chapter 1.

```
@rendermode InteractiveServer
@page "/"

<PageTitle>Experiment With Controls</PageTitle>

<div class="container">
    <div class="row">
        <h1>Experiment with controls</h1>
    </div>
    <div class="row mt-2">
        <div class="col-sm-6">
            Pick a number:
        </div>
        <div class="col-sm-6">
            <input type="range"/>
        </div>
    </div>
    <div class="row mt-5">
        <h2>
            Here's the value:
        </h2>
    </div>
</div>
```

The <PageTitle> tag sets the title that gets displayed in the browser tab.

The `class="row"` attribute in this tag tells the page to render everything between the opening `<div class="row">` tag and the closing `</div>` tag in a single row on the page.

Adding mt-2 to the class causes the page to add a two-space top margin above the row.

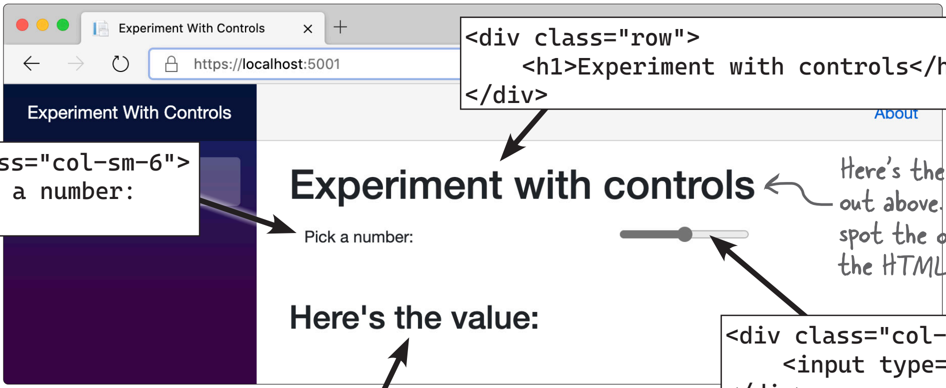
This is an input tag. It has a type attribute that determines what kind of input control appears on the page. When you set the `type` to `range`, it displays a slider:

```
<input type="range"/>
```

HTML controls sometimes look different depending on what browser you use. A slider in Edge looks like this:

2 Run your app.

Run your app just like you did with the app in Chapter 1. Compare the HTML markup with the page displayed in the browser—match up the individual `<div>` blocks with what's displayed on the page.



```
<div class="row">
    <h1>Experiment with controls</h1>
</div>
```

```
<div class="col-sm-6">
    Pick a number:
</div>
```

Here's the row we pointed out above. See if you can spot the other rows in the HTML markup.

```
<div class="col-sm-6">
    <input type="range"/>
</div>
```

```
<div class="row mt-5">
    <h2>Here's the value:</h2>
</div>
```

3 Add C# code to your page.

Go back to *Home.razor* and **add this C# code** to the bottom of the file:

```
@code
{
    private string DisplayValue = "";

    private void UpdateValue(ChangeEventArgs e)
    {
        DisplayValue = e.Value.ToString();
    }
}
```

The UpdateValue method is a Change event handler. It takes one parameter, which your method can use to do something with the data that changed.

The change event handler updates DisplayValue any time it's called with a value. It's okay if you see a squiggly underline and get a warning on this line of code—we'll learn how to fix that later in the book

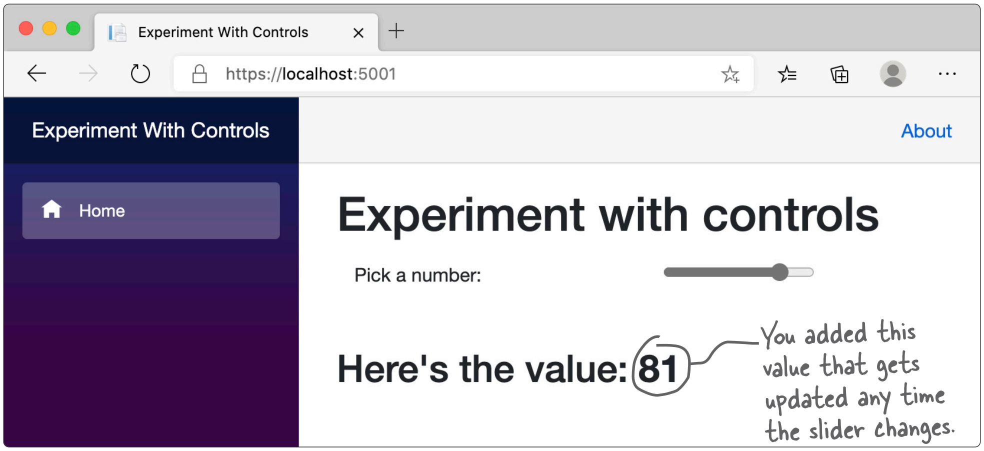
4 Hook up your range control to the Change event handler you just added.

Add an **@onchange** attribute to your range control:

```
@rendermode InteractiveServer
@page "/"

<PageTitle>Experiment With Controls</PageTitle>

<div class="container">
    <div class="row">
        <h1>Experiment with controls</h1>
    </div>
    <div class="row mt-2">
        <div class="col-sm-6">
            Pick a number:
        </div>
        <div class="col-sm-6">
            <input type="range" @onchange="UpdateValue" />
        </div>
    </div>
    <div class="row mt-5">
        <h2>
            Here's the value: <strong>@DisplayValue</strong>
        </h2>
    </div>
</div>
```

When you use @onchange to hook up a control to a Change event handler, your page calls the event handler any time the control's value changes.

Any time DisplayValue changes, the value displayed on the page will change too.



You added this value that gets updated any time the slider changes.

Add a text input to your app

The goal of this project is to experiment with different kinds of controls, so let's add a **text input control** so users can type text into the app and have it display at the bottom of the page.

1 Add a text input control to your page's HTML markup.

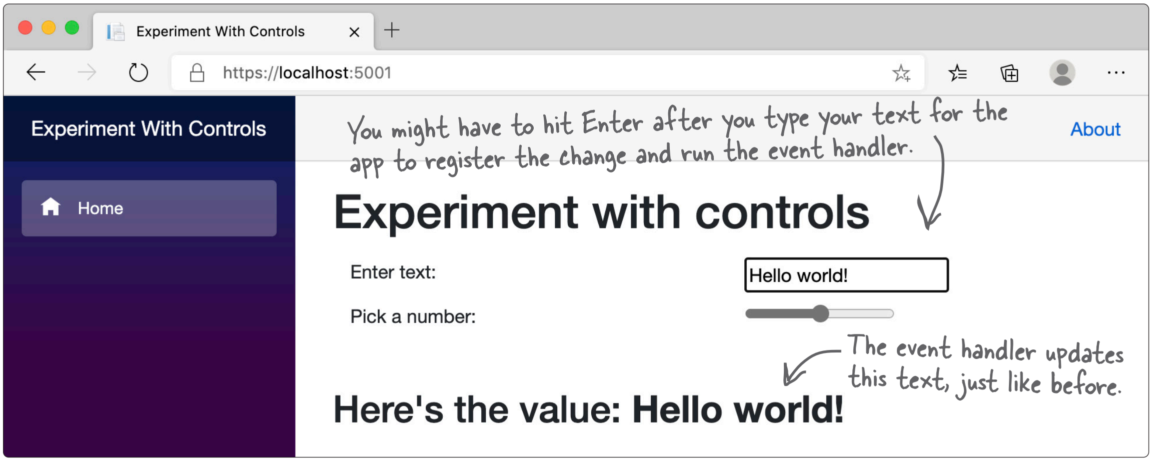
Add an `<input ... />` tag that's almost identical to the one you added for the slider. The only difference is that you'll set the `type` attribute to `"text"` instead of `"range"`. Here's the HTML markup:

```
<div class="container">
    <div class="row">
        <h1>Experiment with controls</h1>
    </div>
    <div class="row mt-2">
        <div class="col-sm-6">
            Enter text:
        </div>
        <div class="col-sm-6">
            <input type="text" placeholder="Enter text"
                   @onchange="UpdateValue" />
        </div>
    </div>
    <div class="row mt-2">
        <div class="col-sm-6">
            Pick a number:
        </div>
```

You're adding another row to your page with a two-space top margin.

Here's the markup for the text input control. Its type is `"text"` and it uses the same `@onchange` tag as the slider. There's an additional tag to set the placeholder text, so the control looks like this until the user enters text:

Run your app again—now it has a text input control. Any text you enter will show up at the bottom of the page. Try changing the text, then moving the slider, then changing the text again. The value at the bottom will change each time you modify a control.



You might have to hit Enter after you type your text for the app to register the change and run the event handler.

Experiment with controls

Enter text: Hello world!

Pick a number:

The event handler updates this text, just like before.

Here's the value: Hello world!

2 Add an event handler method that only accepts numeric values.

What if you only want to accept numeric input from your users? **Add this method** to the code between the brackets at the bottom of the Razor page:

```
private void UpdateNumericValue(ChangeEventArgs e)
{
    if (int.TryParse(e.Value.ToString(), out int result))
    {
        DisplayValue = e.Value.ToString();
    }
}
```

You'll learn all about int.TryParse later in the book—for now, just enter the code exactly as it appears here.

You may see warnings on the code in this method. You can ignore them for now, we'll learn more about them in Chapter 6.

Try putting a breakpoint in this method and using the debugger to explore how it works.

3 Change the text input to use the new event handler method.

Modify your text control's **@onchange** attribute to call the new event handler:

```
<input type="text" placeholder="Enter text"
       @onchange="UpdateNumericValue" />
```

Now try entering text into the text input—it won't update the value at the bottom of the page unless the text that you enter is an integer value.

Exercise

You used **Button controls** in your animal matching game in Chapter 1. Here's some HTML markup to add a strip of buttons to your page—it's very similar to the code that you used earlier. Your job is to **finish this code** so it adds six buttons, and **add an event handler to the C# code.**

```
<div class="row mt-2">
    <div class="col-sm-6">Pick a number:</div>
    <div class="col-sm-6"><input type="range" @onchange="UpdateValue" /></div>
</div>
<div class="row mt-2">
    <div class="col-sm-6">Click a button:</div>
    <div class="col-sm-6 btn-group" role="group">
        
        {
            string valueToDisplay = $"Button #{buttonNumber}";
            <button type="button" class="btn btn-secondary"
                    @onclick="() => ButtonClick(valueToDisplay)">
                @buttonNumber
            </button>
        }
    </div>
</div>
<div class="row mt-5">
    <h2>
        Here's the value: <strong>@DisplayValue</strong>
    </h2>
</div>
```

Replace this box with a line of C# code that will cause the page to display six buttons.

When the buttons are clicked, they call an event handler method called ButtonClick. Add that method to the code at the bottom of the page—it contains just one statement..

Exercise Solution

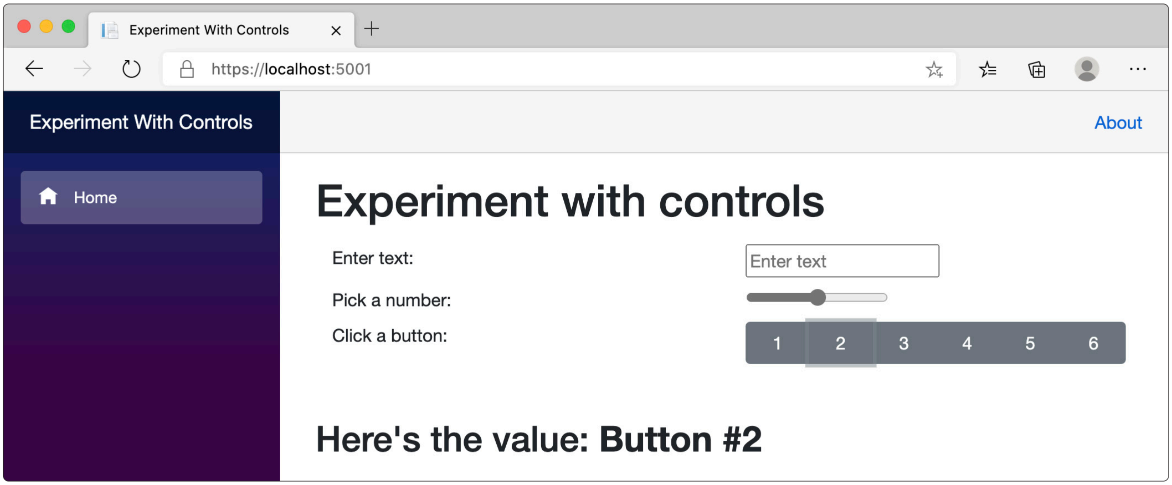
Here's the line of code that makes the Razor markup add six buttons to the page. It's a **for** loop, and it works just like the other **for** loops you learned about in Chapter 2:

```
<div class="row mt-2">
    <div class="col-sm-6">Pick a number:</div>
    <div class="col-sm-6"><input type="range" @onchange="UpdateValue" /></div>
</div>
<div class="row mt-2">
    <div class="col-sm-6">Click a button:</div>
    <div class="col-sm-6 btn-group" role="group">
        @for (var buttonNumber = 1; buttonNumber <= 6; buttonNumber++)
        {
            string valueToDisplay = $"Button #{buttonNumber}";
            <button type="button" class="btn btn-secondary"
                    @onclick="() => ButtonClick(valueToDisplay)">
                @buttonNumber
            </button>
        }
    </div>
</div>
<div class="row mt-5">
    <h2>
        Here's the value: <strong>@DisplayValue</strong>
    </h2>
</div>
```

The for loop that creates the buttons works exactly like the one in the animal matching game—the code is almost identical. The buttons are styled as a group (that's what btn-group does) and shaded differently in some browswers (that's what btn-secondary does).

Here's the event handler method to **add to the @code section** at the bottom of the page. It sets DisplayValue to the value passed to it by the button when it's clicked:

```
private void ButtonClick(string displayValue)
{
    DisplayValue = displayValue;
}
```



Here's the value: Button #2

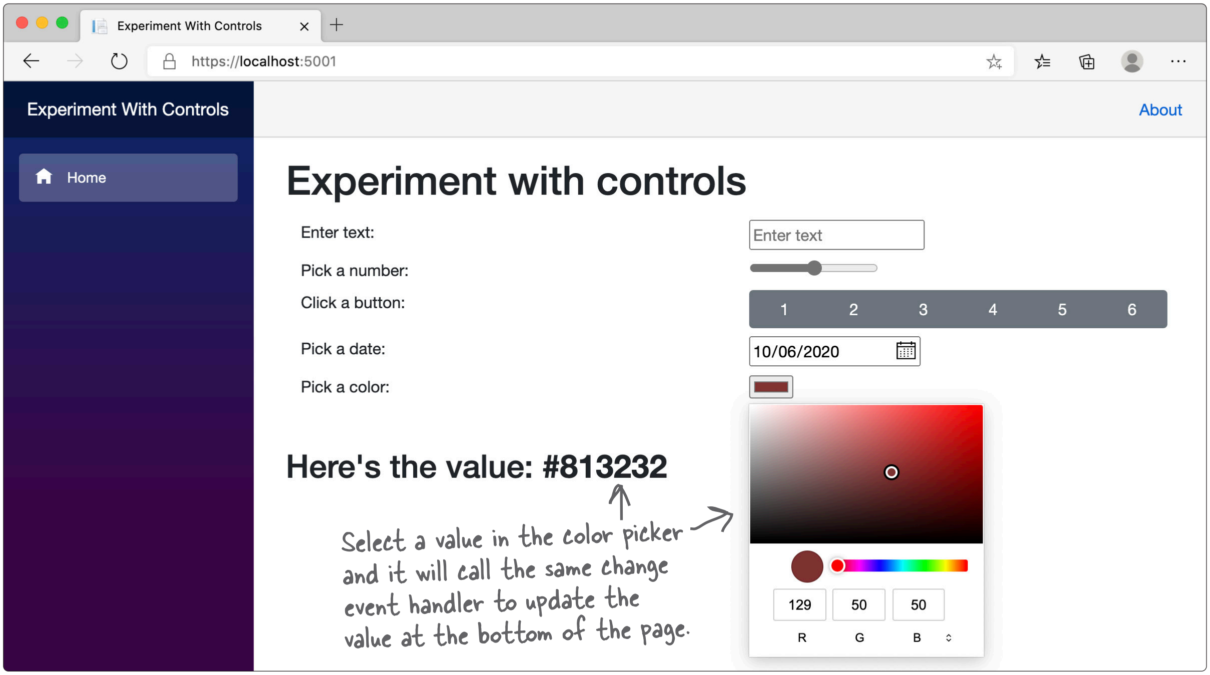
Add color and date pickers to your app

Pickers are just different types of inputs. A **date picker** has the input type `"date"` and a **color picker** has the input type `"color"`—other than that, the HTML markup for those input types is identical.

Modify your app to **add a date picker and a color picker**. Here's the HTML markup—add it just above the `<div>` tag that contains the display value:

```
    <div class="row mt-2">
        <div class="col-sm-6">Pick a date:</div>
        <div class="col-sm-6">
            <input type="date" @onchange="UpdateValue" />
        </div>
    </div>
    <div class="row mt-2">
        <div class="col-sm-6">Pick a color:</div>
        <div class="col-sm-6">
            <input type="color" @onchange="UpdateValue" />
        </div>
    </div>
    <div class="row mt-5">
        <h2>Here's the value: @DisplayValue</h2>
    </div>
</div>
```

The date and color pickers use the same Change event handler method, so you don't need to modify the code at all to display the color or date that the user picks.



Experiment with controls

Enter text: Enter text

Pick a number:

Click a button: 1 2 3 4 5 6

Pick a date: 10/06/2020

Pick a color:

R 129 G 50 B 50

Here's the value: #813232

Select a value in the color picker and it will call the same change event handler to update the value at the bottom of the page.

Add a dropdown control to display a list of choices

A **dropdown control** (also called a **select control**) displays a list of items in a dropdown so the user can pick one of them. Let's add one to your app that lets the user pick a bird from a list of birds.

1. **Add the HTML for a dropdown (or select) control.**

Let's add three rows to the bottom of your page: a row with a dropdown that contains a list of birds, a row with a button the user can click to add a bird, and a row that displays the list of birds that were added.

Go ahead and **add this HTML code** just above the closing </div> tag:

```
<div class="row mt-5">
     <h2>
         Here's the value: <strong>@DisplayValue</strong>
     </h2>
 </div>
 <div class="row mt-5">
     <label>Pick a bird</label>
     <select @bind="selectedBird">
         @foreach (var bird in birds)
         {
             <option value="@bird">@bird</option>
         }
     </select>
 </div>
 <div class="row mt-2">
     <button type="button" class="btn btn-primary"
             @onclick="AddBird">Add a bird</button>
 </div>
 <div class="row mt-2" style="background-color: darkblue; color: white;">
     @foreach (var bird in addedBirds)
     {
         <div>@bird</div>
     }
 </div>
</div>
```

This is the existing HTML to display the value from the radio buttons, date picker, or color picker.

This row contains a dropdown that displays everything between its opening and closing tags in a dropdown. You'll use a foreach loop to include everything in a variable called birds.

This row contains a foreach loop that displays the birds that were added. We used a style property to give it white text on a dark blue background.

Here's the closing <div> tag.

2. **Add three fields to store the birds in the dropdown list, the birds that were added, and the bird that's currently selected.**

Add these variables just under the `@code` opening bracket:

```
@code
{
    string[] birds = [ "Duck", "Pigeon", "Penguin", "Ostrich", "Owl" ];
    string[] addedBirds = [];
    string selectedBird;
```

birds and addedBirds are string arrays. You'll learn more about arrays in Chapter 3. For now, we'll give you all of the code you need to work with them.

We gave you a lot of code, but it's a lot like code on the page you've already seen. Let's break down this part of it:

```
<div class="row mt-2" style="background-color: darkblue; color: white;">
    @foreach (var bird in addedBirds)
    {
        <div>@bird</div>
    }
</div>
```

This <div> block adds a new row to the page. The style property changes its color.

This @foreach adds a block of HTML to the page for each bird in addedBirds.

This is the block of HTML that the @foreach adds to the page. It displays a bird on a new line.

3 Add a method that reads the selected bird and adds it to the page.

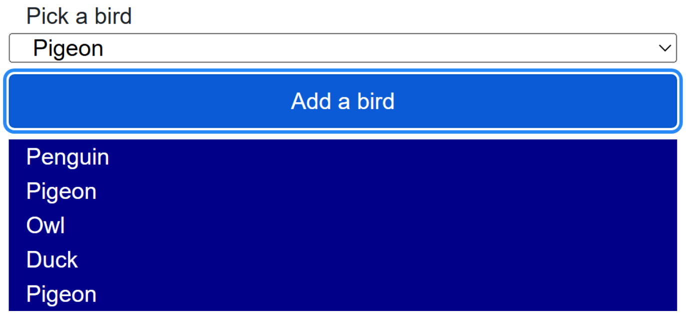
Take a closer look at the :

```
@code
{
    string[] birds = [ "Duck", "Pigeon", "Penguin", "Ostrich", "Owl" ];
    string[] addedBirds = [];
    string selectedBird;

    private void AddBird()
    {
        string[] newAddedBirds = new string[addedBirds.Length + 1];
        for (int i = 0; i < addedBirds.Length; i++)
        {
            newAddedBirds[i] = addedBirds[i];
        }
        newAddedBirds[newAddedBirds.Length - 1] = selectedBird;
        addedBirds = newAddedBirds;
    }
```

You'll learn a lot more about arrays—and specifically how this code works—when we get to Chapter 6. For now, just type in the code exactly like it appears here.

4 Run your app and use your new dropdown control.

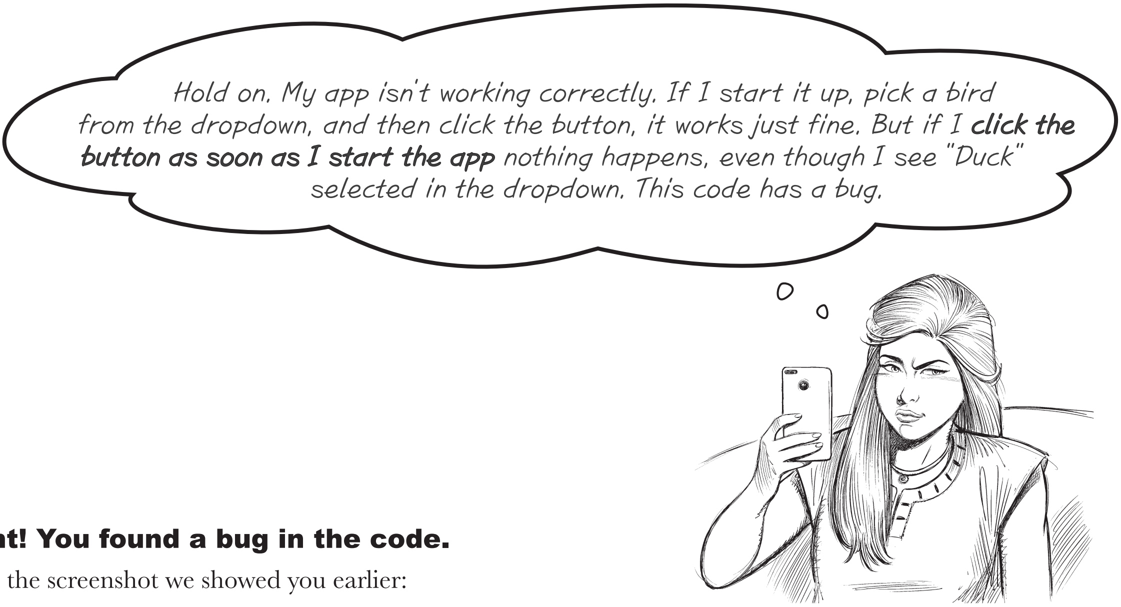
Scroll to the bottom of the page, choose a bird from the dropdown, and click the Add a bird button. The bird will get added to the list that contains the birds. Select a few more birds and add them.



Penguin
Pigeon
Owl
Duck
Pigeon

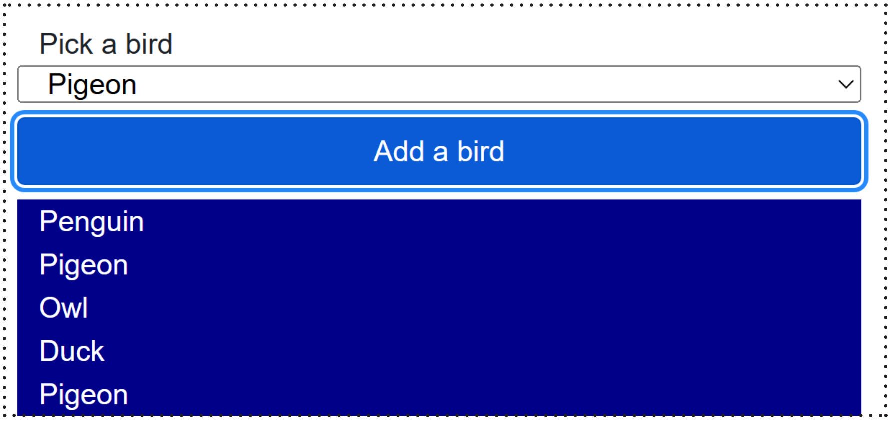
Choose a bird from the dropdown, then click the "Add a bird" button.

The bird that you selected in the dropdown gets added to the list of birds on the page.

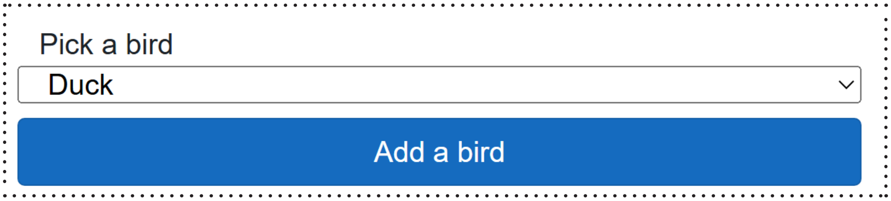

Hold on. My app isn't working correctly. If I start it up, pick a bird from the dropdown, and then click the button, it works just fine. But if I **click the button as soon as I start the app** nothing happens, even though I see "Duck" selected in the dropdown. This code has a bug.

You're right! You found a bug in the code.

Take a look at the screenshot we showed you earlier:



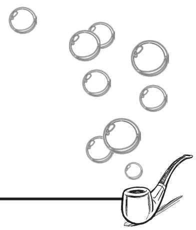
We were really careful to show that birds were already added. But when you first start your app, you don't see any added birds.



Oops! It looks like the button doesn't work until you choose something from the dropdown, even though it looks like Duck is already selected.

Run your app and try clicking the button ***before you select a bird***. Nothing happens! If you want to add a duck, you have to click on the dropdown and choose Duck, *even though Duck is already selected*.

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 Duck That Didn't Quack

Understanding a bug is the first step in fixing it.

In Chapter 1, 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.

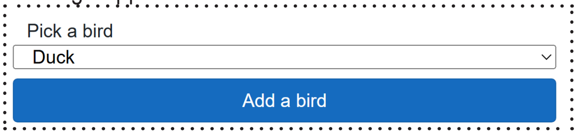
Luckily, the debugger in Visual Studio and Visual Studio Code is a great tool for that. (That's why it's called a debugger: it's a tool that helps you get rid of bugs!)

Reproduce the bug

It seems 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, you can't just rely on what seems 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:

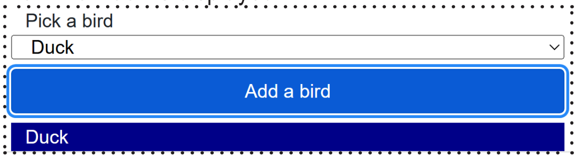
1. Start your app again.
2. Click the "Add a bird" button.
3. Nothing happens.



Pick a bird

Duck

4. Choose "Duck" from the dropdown.
5. Click the "Add a bird" button again.
6. Now the Duck is displayed.



Pick a bird

Duck

Duck

> "There is nothing more deceptive than an obvious fact."
> – Sherlock Holmes

Now restart your app, then try a few different things. Does it always happen, every time you run the app? What happens if you choose another bird first? What if you click the button several times before selecting a bird?

You can make the bug happen over and over again, at will. That means the problem is **reproducible**: you can follow a set of steps 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:

```
<div class="row mt-2">
    <button type="button" class="btn btn-primary"
            @onclick="AddBird">Add a bird</button>
</div>
```

All of the code to display the selected bird is in that **AddBird method**. Now we have a suspect!

Remember, If your app doesn't pause on the breakpoint, make sure you're starting the app with debugging. Run the app by pressing F5 or choosing Start Debugging from the Debug (Visual Studio) or Run (VSCode) menu.

IDE Tip: Using the debugger

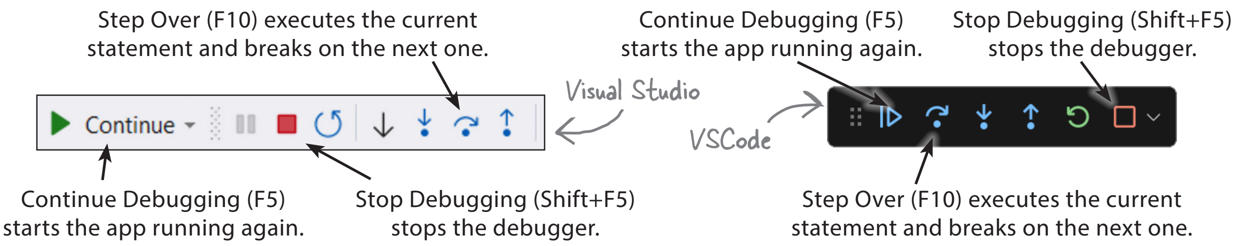
You're going to be using the debugger a lot in this book! We've walked you through it a few times, but as you get further in the book and write more and more code, you should feel comfortable using the debugger on your own.

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 handy **list of useful 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), 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) from 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 IDE's toolbar, along with their keyboard shortcuts:



Step Over (F10) executes the current statement and breaks on the next one.

Continue Debugging (F5) starts the app running again.

Stop Debugging (Shift+F5) stops the debugger.

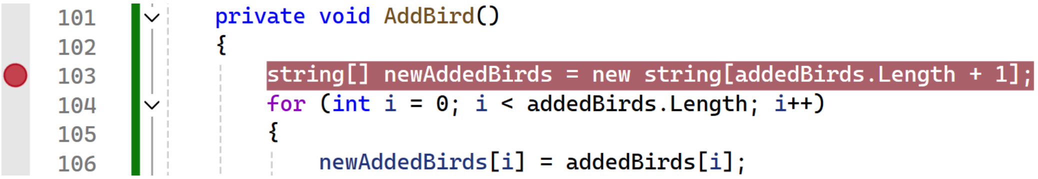
Continue Debugging (F5) starts the app running again.

Stop Debugging (Shift+F5) stops the debugger.

Step Over (F10) executes the current statement and breaks on the next one.

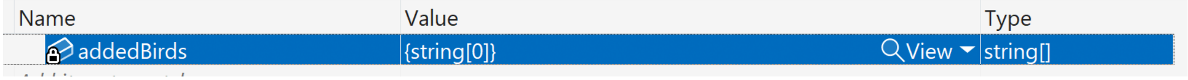
Sleuth it Out

Now that we have a suspect, let's catch it in the act. **Add a breakpoint** to the first line in the AddBird method:



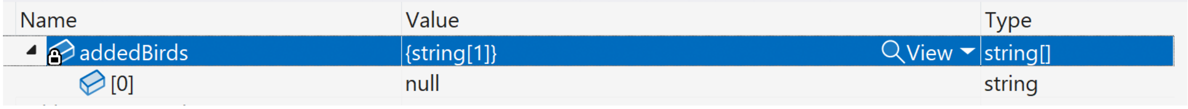
```
private void AddBird()
{
    string[] newAddedBirds = new string[addedBirds.Length + 1];
    for (int i = 0; i < addedBirds.Length; i++)
    {
        newAddedBirds[i] = addedBirds[i];
```

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 addedBirds**, just like you did earlier in the chapter. The value should be **`{string[0]}`**:



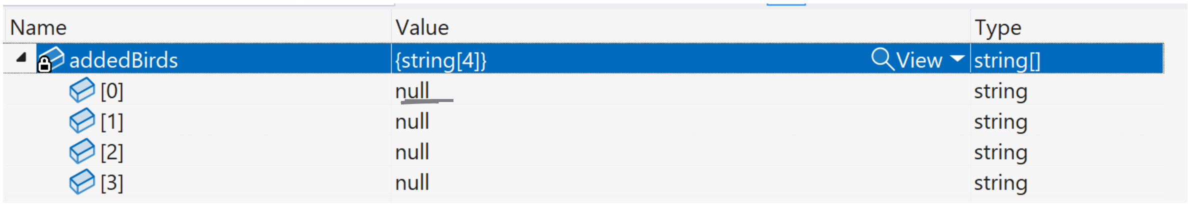
Name	Value	Type
addedBirds	{string[0]} 🔍View ▾	string[]

Then **choose Continue (F5) from the Run or Debug menu** (or click the triangle Continue button) to start up the app again. Now click the button again. The breakpoint stops, but now addedBirds has the value **`{string[1]}`**, and there's a triangle next to the watch. Click on the triangle to expand addedBirds:



Name	Value	Type
◢ addedBirds	{string[1]} 🔍View ▾	string[]
[0]	null	string

Repeat that step (press F5 then click the button) three times. Now you'll see **`{string[4]}`** with these values:



Name	Value	Type
◢ addedBirds	{string[4]} 🔍View ▾	string[]
[0]	null	string
[1]	null	string
[2]	null	string
[3]	null	string

We haven't talked about arrays or told you what null means, but even with this limited information we've got a lot of clues. We know that addedBirds has the birds to display, and somehow null keeps them from being displayed. We just need to figure out where that null is coming from. Let's start with the HTML that displays the dropdown:

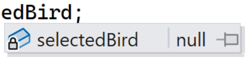
```
<select @bind="selectedBird">
```

That tells the app to store the selected bird in the selectedBird variable that you added. Now look at this line of code from the AddBird method:

```
newAddedBirds[newAddedBirds.Length - 1] = selectedBird;
```

Even though we haven't talked about arrays yet, you can see that something is being set to the value in selectedBird. Hover over it to see its value:

```
selectedBird;
```



selectedBird | null

There's that null value again.

When the app starts, selectedBird contains null, even though Duck is selected. We have our culprit! We've sleuthed out the bug, and we know enough to fix it.

Before we show you the solution, can you think of how you would fix this bug? Is there a way to set selectedBird as soon as the page is initialized?

Use the OnInitialized method to set selectedBird

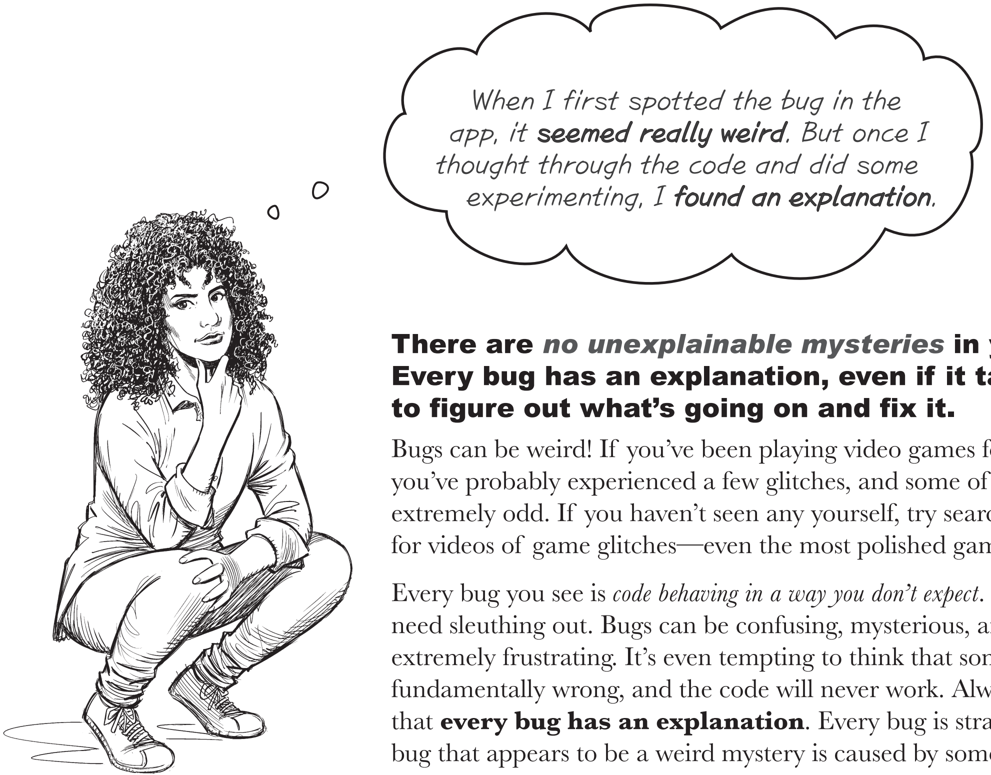
In Chapter 1 we learned about the the OnInitialized method, which gets run as soon as the page is initialized, and we used it to set up the game. Now you can use it to set up your app so selectedBird starts out with a bird.

Add this OnInitialized method that sets the selectedBird variable:

```
protected override void OnInitialized()
{
    selectedBird = birds[0];
}
```

This statement sets selectedBird to the first value in the birds array. You'll learn more about arrays in Chapter 3.

Run your app again. **The bug is fixed!**


When I first spotted the bug in the app, it **seemed really weird**. But once I thought through the code and did some experimenting, I **found an explanation**.

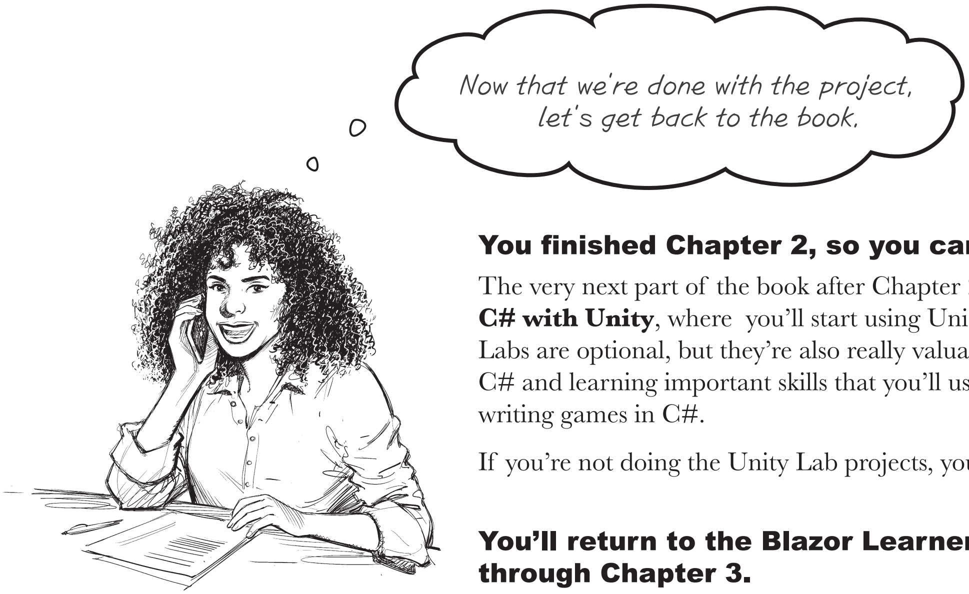
There are *no unexplainable mysteries* in your code. Every bug has an explanation, even if it takes work to figure out what's going on and fix it.

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 seen 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.
- You can build up a page with **rows** using `<div class="row">` tags.
- Add a **slider** to your page with an input tag like this: `<input type="range" @onchange="UpdateValue" />`
- The **@onchange property** causes the control to run an event handler method every time the value is changed.
- Use an <input> tag with different type properties to add other kinds of controls. Get user text input with a text box control: `<input type="text" placeholder="Enter text" @onchange="UpdateValue" />`
- A **date picker** has the input type "date" and a **color picker** has the input type "color".
- You can **pass values** to a button's click event handler: **`@onclick="() => ButtonClick(valueToDisplay)"`**
- A **dropdown (or select) control** creates a dropdown: `<select @bind="selectedBird">` Everything between the opening and closing tags is displayed in the dropdown. the @bind-value property causes the control to update a variable every time it's changed.
- 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.


Now that we're done with the project, let's get back to the book.

You finished Chapter 2, so you can go to the next chapter.

The very next part of the book after Chapter 2 is **Unity Lab #1: Explore C# with Unity**, where you'll start using Unity to create 3D games. The Unity Labs are optional, but they're also really valuable for getting practice using C# and learning important skills that you'll use even if you aren't planning on writing games in C#.

If you're not doing the Unity Lab projects, you can go straight to Chapter 3.

You'll return to the Blazor Learner's Guide partway through Chapter 3.

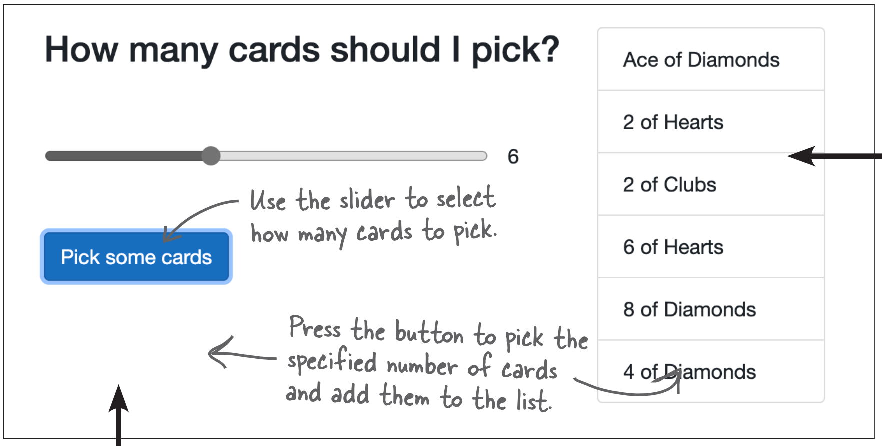
Watch for this heading—it comes after you create a console app that picks random cards and displays them:

As soon as you get to this heading in the book, come back to the Blazor Learner's Guide so you can build a Blazor version of that project. →

Build a MAUI version of your random card app

Build a Blazor version of your random card app

In the next project, you'll build a Blazor app called PickACardBlazor. It will use a slider to let you choose the number of random cards to pick and display those cards in a list. Here's what it will look like:



How many cards should I pick?

Ace of Diamonds
2 of Hearts
2 of Clubs
6 of Hearts
8 of Diamonds
4 of Diamonds

Use the slider to select how many cards to pick.

Press the button to pick the specified number of cards and add them to the list.

You'll use a loop to turn an array of cards into a series of HTML tags, just like you did with the buttons in the previous Blazor projects.

This button's event handler will call a method in your class that returns a list of cards, then it will add each card to an array, just like you did with the bird dropdown in Chapter 2.

Do this!

Create a new Blazor WebAssembly App project called PickRandomCardsBlazor.

You'll follow exactly the same steps you used to create your animal matching game in Chapter 1:

- Open Visual Studio and create a new project.
- Select **Blazor WebAssembly App**, just like you did with your previous Blazor apps.
- Name your new app **PickRandomCardsBlazor**. Visual Studio will create the project.

Brain Power

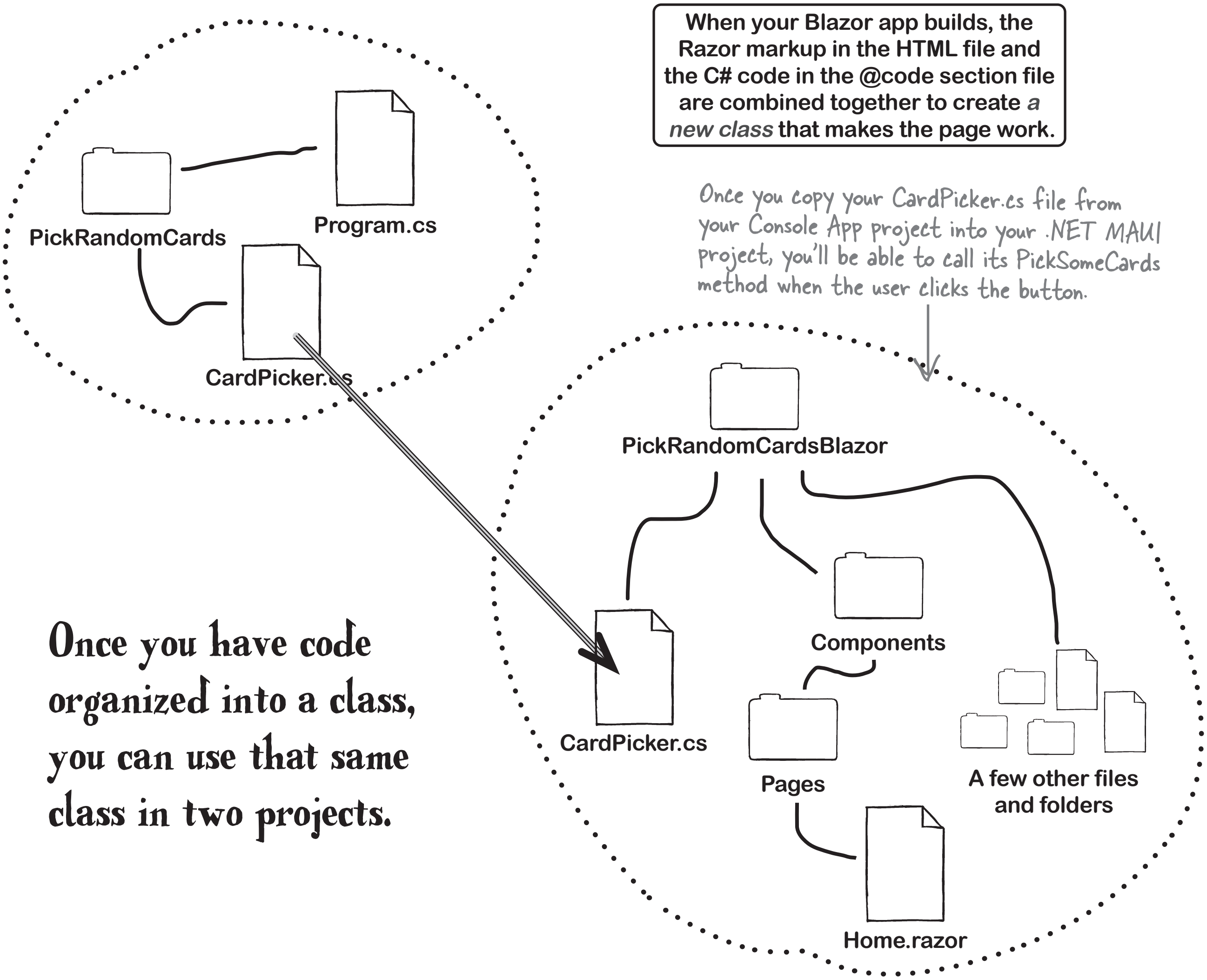
Go back to the Blazor app you built in Chapter 2 and look at how the AddBird method worked. Now look at how the "Pick some cards" button in this app will work. What do you think will go in the event handler method for that button?

Reuse your CardPicker class in your new Blazor app

If you've written a class for one program, you'll often want to use the same behavior in another. That's why one of the big advantages of using classes is that they make it easier to **reuse** your code. You'll give your card picker app a shiny new user interface, but keep the same behavior by reusing your CardPicker class.

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.

When your Blazor app builds, the Razor markup in the HTML file and the C# code in the @code section file are combined together to create *a new class* that makes the page work.



Once you copy your CardPicker.cs file from your Console App project into your .NET MAUI project, you'll be able to call its PickSomeCards method when the user clicks the button.

Once you have code organized into a class, you can use that same class in two projects.

Reuse your 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 **reuse that class** by ***copying the file*** with the C# code into your new Blazor project.

Do this!

1. **Choose Add Existing Item in Visual Studio or manually copy the file in VSCode.**
This feature in the IDE will copy an existing file into your project. You created a file called *CardPicker.cs* in your PickRandomCards console app. Now you'll tell the IDE to **add that class file** to your Blazor project, which will cause it to copy the file into your MAUI app's project folder.

    - In Visual Studio, right-click on the project in the Solution Explorer window and choose Add >> Existing Item (Shift+Alt+A), or choose Add Existing Item from the Project menu.
    - In VSCode, you'll need to manually copy the file into the folder. Right-click on the project in the Solution Explorer and choose "Reveal in File Explorer" (or "Reveal in Finder" if you're using a Mac). Use your operating system to copy the file into your project folder that VSCode opened. Once the file is copied, it will automatically appear in the Solution Explorer.

2. **Find your *CardPicker.cs* file and add it to your project.**
The IDE 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.

Make sure CardPicker.cs now shows up in your Solution Explorer. Open it and make sure that you see the code for the CardPicker class from earlier in the chapter.

3. **Try to use your CardPicker class in the @code section of your home page.**
Open *Home.razor*. Make sure you've added the @rendermode line, updated the <PageTitle> tag, and deleted everything else (follow the instructions we gave you back in Chapter 2).

    Next, **add a @code section**. You'll need an array to hold the cards, so add a pickedCards variable and use the CardPicker.PickSomeCards method to initialize it:

    ```
        string[] pickedCards = CardPicker.PickSomeCards(5);
    ```

    Here's what your *Home.razor* file will look like. Did you run into a problem trying to call PickSomeCards?

```
@rendermode InteractiveServer
@page "/"

<PageTitle>Pick Random Cards</PageTitle>

@code {
    string[] pickedCards = CardPicker.
}
```

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

You used either a **file-scoped namespace** or **block-scoped namespace** to put your CardPicker class in the PickRandomCards namespace. But that's not the namespace of the code in your Blazor app.

The component in your .razor file is a blend of C# code and HTML with Razor markup. When your Blazor web app gets compiled, **each Razor file is transformed into a class**. This class handles both rendering (HTML) and logic (C#), and it lives in the `PickRandomCardsBlazor.Components.Pages` namespace.

The reason your Blazor code 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.

Open the Program.cs file at the root of your Blazor app and look at the first line:

```
using PickRandomCardsBlazor.Components;
```

That's a using directive. Using directives in C# code start with the keyword `using` and end with a semicolon. They look a little different in a Razor file—then start with `@using` and don't end in a semicolon.

Add this!

Add this line to the top of your Home.razor file. If you chose a different name for your console app, replace PickRandomCards with the namespace in your *CardPicker.cs* file.

```
@using PickRandomCards
```

This *using directive* will let you add code to your *Home.razor* file that uses classes in the PickRandomCards namespace—so now you 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 the IDE will pop up its IntelliSense window, just like you'd expect it to.

Finish adding the code section to your Home.razor file

Make sure you have the `@rendermode`, `@page`, and `@using` lines at the top, the PageTitle tag is updated, and your @code section has a pickedCards variable.

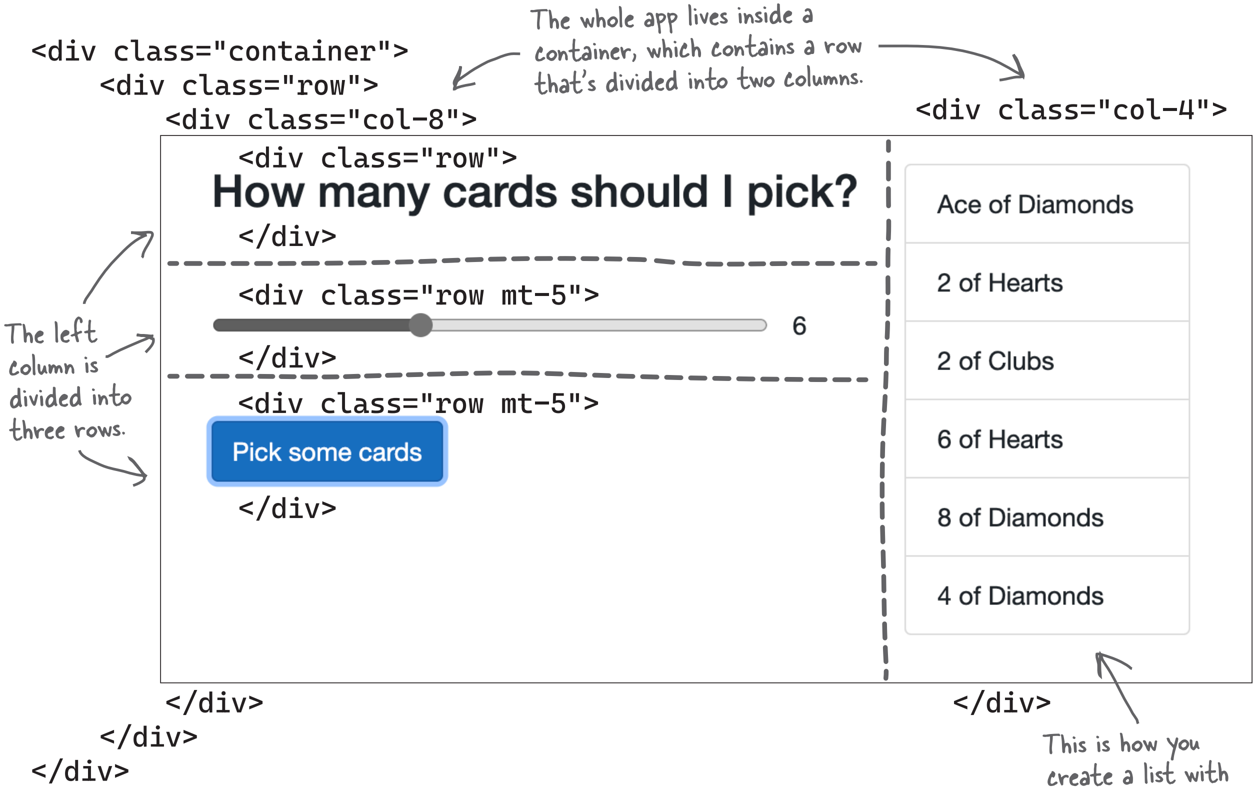
Here's what your *Home.razor* file will look like.

```
@rendermode InteractiveServer
@page "/"
@using PickRandomCards

<PageTitle>Pick Random Cards</PageTitle>

@code {
t}
```

When your project builds, the HTML, Razor markup, and C# code in the Home.razor file is built into a class called Home in the a separate namespace. Since it's a class, that means pickedCards is more than just a variable. It's actually a field in that class.

The page is laid out with rows and columns

The Blazor apps in Chapters 1 and 2 used HTML markup to create rows and columns, and this new app does the same thing. Here's a picture that shows you how your app will be laid out:



```
<div class="container">
    <div class="row">
        <div class="col-8">
            <div class="row">
            How many cards should I pick?
            </div>
            <div class="row mt-5">
            6
            </div>
            <div class="row mt-5">
            Pick some cards
            </div>
        </div>
                                            <div class="col-4">
                                            Ace of Diamonds
                                            2 of Hearts
                                            2 of Clubs
                                            6 of Hearts
                                            8 of Diamonds
                                            4 of Diamonds
                                            </div>
    </div>
</div>
```

The whole app lives inside a container, which contains a row that's divided into two columns.

The left column is divided into three rows.

This is how you create a list with HTML markup.

Here's the code that generates the list of cards in the right column. It uses a **foreach** loop (like the one you used in your animal matching game) to create a list from an array called **pickedCards**:

```
<div class="col-4">
   <ul class="list-group">
      @foreach (var card in pickedCards)
      {
         <li class="list-group-item">@card</li>
      }
   </ul>
</div>
```

```
<div class="col-4">
  <ul>
    <li>Ace of Diamonds</li>
    <li>2 of Hearts</li>
    <li>2 of Clubs</li>
    <li>6 of Hearts</li>
    <li>8 of Diamonds</li>
    <li>4 of Diamonds</li>
  </ul>
</div>
```

The list starts with **<ul class="list-group">** and ends with **</ul>** (which stands for "unnumbered list"). Each list item begins with **<li class="list-group-item">** and ends with **</li>**.

The slider uses data binding to update a variable

The code at the bottom of the page will start with a variable called `numberOfCards`:

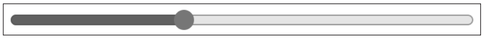
```
@code {
    int numberOfCards = 5;
```

You *could* use an event handler to update `numberOfCards`, but Blazor has a better way: **data binding**, which lets you set up your input controls to automatically update your C# code, and can automatically insert values from your C# code back into the page.

Here's the HTML markup for the header, the range input, and the text next to it that shows its value:

```
    <div class="row">
        <h3>How many cards should I pick?</h3>
    </div>
    <div class="row mt-5">
        <input type="range" class="col-10 form-control-range"
               min="1" max="15" @bind="numberOfCards" />
        <div class="col-2">@numberOfCards</div>
    </div>
```

How many cards should I pick?



6

Take a closer look at the attributes for the `input` tag. The `min` and `max` attributes restrict the input to values from 1 to 15. The **`@bind`** attribute sets up the data binding, so any time the slider changes Blazor automatically updates `numberOfCards`.

The `input` tag is followed by `<div class="col-2">@numberOfCards</div>`—that markup adds text (with `ml-2` adding space to the left margin). This also uses data binding, but to go in the other direction: every time the `numberOfCards` field is updated, Blazor automatically updates the text inside that `div` tag.

Exercise

We've given all you almost all of the parts you need to add the HTML markup and code to your *Index.razor* file. Can you figure out how to put them together to make your web app work?

Step 1: Finish the HTML markup

The first four lines of *Index.razor* are identical to the first four lines in the ExperimentWithControlsBlazor app from Chapter 2. You can find the next two lines of HTML at the top of the screenshot where we explain how the rows and columns work. The only markup we haven't given you yet is for the button—here it is:

```
<button type="button" class="btn btn-primary"
        @onclick="UpdateCards">Pick some cards</button>
```

When you enter this into the IDE, it may add a line break after the opening tag and before the closing tag.

Step 2: Finish the code

We gave you the beginning of the `@code` section at the bottom of the page, with an int field called `numberOfCards`.

- You already have a string array field called `pickedCards`.
- Add the UpdateCards event handler method called by the button. It calls CardPicker.PickSomeCards and assigns the result to the `pickedCards` field.

Exercise Solution

Here's the entire code for the *Index.razor* file. You can also follow exactly the same steps from the ExperimentWithControlsBlazor project to remove the extra files and update the navigation menu.

```
@rendermode InteractiveServer
@page "/"
@using PickRandomCards

<PageTitle>Pick Random Cards</PageTitle>

<div class="container">
    <div class="row">
        <div class="col-8">
            <div class="row">
                <h3>How many cards should I pick?</h3>
            </div>
            <div class="row mt-5">
                <input type="range" class="col-10 form-control-range"
                       min="1" max="15" @bind="numberOfCards" />
                <div class="col-2">@numberOfCards</div>
            </div>
            <div class="row mt-5">
                <button type="button" class="btn btn-primary"
                        @onclick="UpdateCards">
                    Pick some cards
                </button>
            </div>
        </div>
        <div class="col-4">
            <ul class="list-group">
                @foreach (var card in pickedCards)
                {
                    <li class="list-group-item">@card</li>
                }
            </ul>
        </div>
    </div>
</div>

@code {
    int numberOfCards = 5;

    string[] pickedCards = CardPicker.PickSomeCards(5);

    void UpdateCards()
    {
        pickedCards = CardPicker.PickSomeCards(numberOfCards);
    }
}
```

The range input and text after it are columns in their own little row.

When you click the button, its Click event handler method UpdateCards sets the pickedCards array to a new set of random cards. As soon as it changes, Blazor's data binding kicks in and it automatically runs the foreach loop again.

numberOfCards and pickedCards are both fields in the Home class that's built from the HTML and C# code in the file.

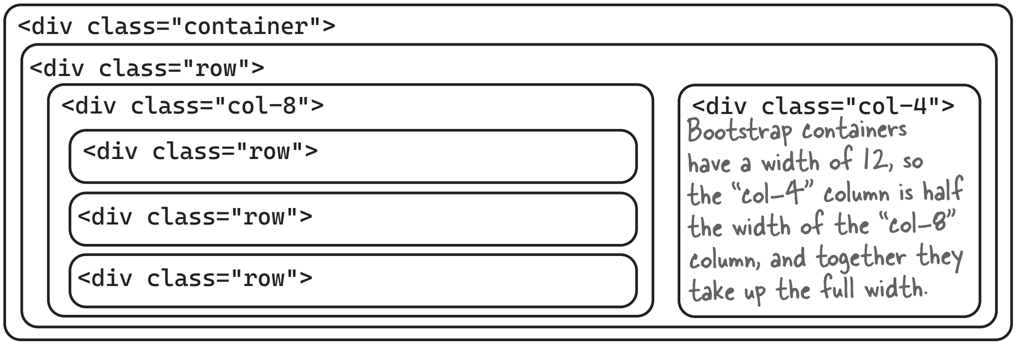
The button's Click event handler method calls the PickSomeCards method in the CardPicker class that you wrote earlier in the chapter.

Behind the Scenes

Your Blazor web apps use Bootstrap for page layout.

Your app looks pretty good! Part of the reason for that is because it uses **Bootstrap**, a free and open source framework for creating web pages that are responsive—they adjust automatically when the screen size changes—and work well on mobile devices.

The row and column layout that drives your app's layout comes straight out of Bootstrap. Your app uses the `class` attribute (which has nothing to do with C# classes) to take advantage of Bootstrap's layout features.



Bootstrap containers have a width of 12, so the "col-4" column is half the width of the "col-8" column, and together they take up the full width.

You can experiment with this—try changing `col-8` and `col-4` so they're both `col-6` to make them equal sizes. What happens when you choose numbers that don't add up to 12?

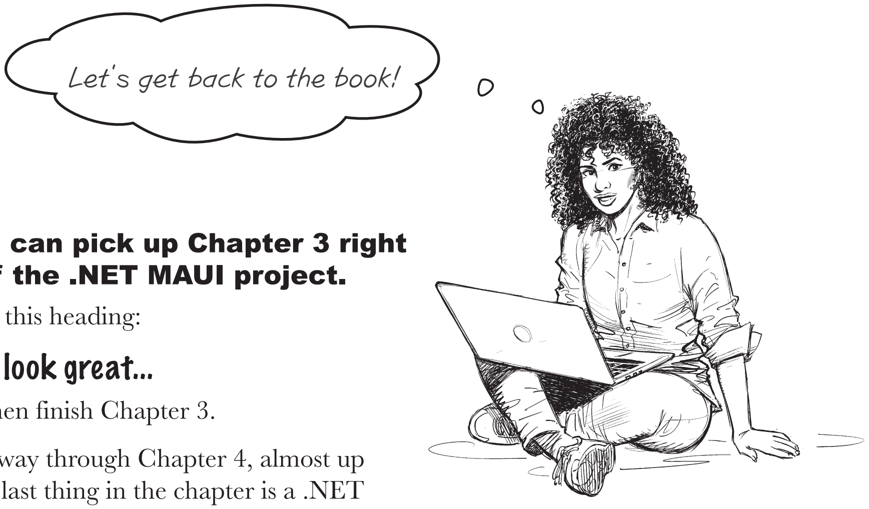
Bootstrap also helps style your controls. Try removing the `class` attribute from the `button`, `input`, `ul`, or `li` tags and running the app again. It still works the same way, but it looks different—the controls lost some of their styling. Try removing all of the `class` attributes—the rows and columns disappear, but the app still functions.

You can learn more about Bootstrap at https://getbootstrap.com.

Take a few minutes and read about Bootstrap. Go to the Bootstrap website, open the documentation, and read the introduction in the "Quick start" guide. You may not understand everything yet, but you'll recognize some of the most important concepts—and you'll know where to learn more if you want to do more advanced Bootstrap design.

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.
- The HTML and Razor markup code combines with the C# code in the Razor component file to **create a new class**.
- You can create an array of values using a **collection expression** by putting the values between a pair of square brackets `[ ]` and separating them with commas.
- The **global namespace** is contains the top-level statements and any class not explicitly put into a namespace using a namespace declaration.
- Use a **using directive** like `using CardPicker;` to use classes from other namespaces in your C# code. Razor pages have slightly different using directives that look like this: `@using CardPicker`
- You can design Blazor apps using **Bootstrap**, a framework that helps you design responsive web pages.
- Bootstrap uses a **grid system with a twelve-column layout**, which lets you lay out your pages horizontally by dividing content into equal-width columns1.
- Use the **class property in your <div> tags** to Add columns to your page. `<div class="col-4">` adds a column that takes up a third of the width of the page (or 4 of the 12 columns in the grid).
- You can **mix rows and columns** in your layout by nesting row <div> tags inside col <div> tags or vice versa to create more complex layouts.



Great idea! You can pick up Chapter 3 right after the end of the .NET MAUI project.

Look for a page with this heading:

Ana's prototypes look great...

Start on that page, then finish Chapter 3.

You can read all the way through Chapter 4, almost up to the end. The very last thing in the chapter is a .NET MAUI project that starts with this heading:

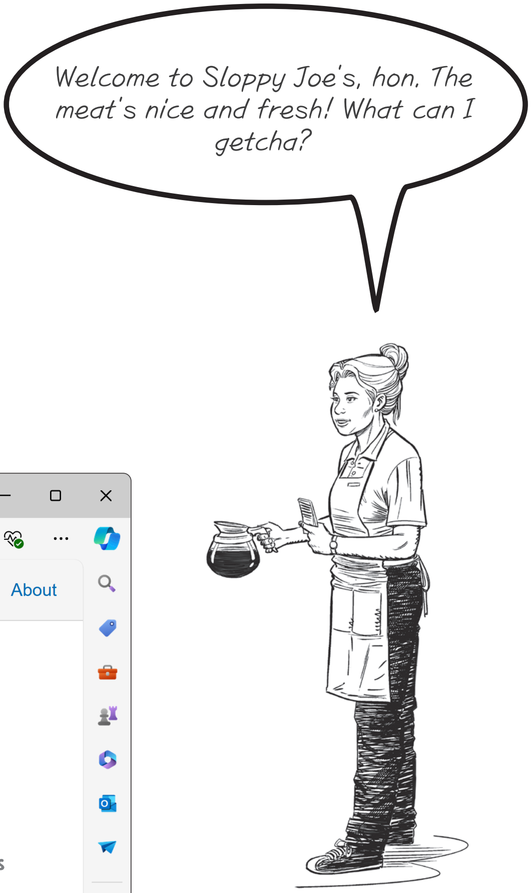
Welcome to Sloppy Joe's Budget House o' Discount Sandwiches!

As soon as you get to that page, come back to the Blazor Learner's Guide for a Blazor version of that project.

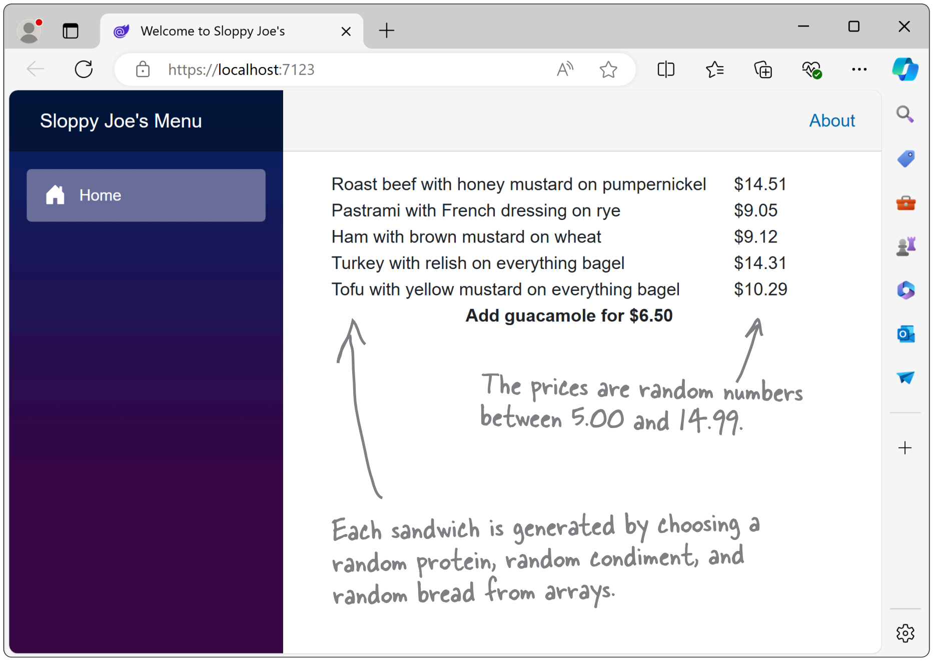
Welcome to Sloppy Joe's Budget House o' Discount Sandwiches!

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 Blazor 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 given a random price, and there's a special random price at the bottom to add guacamole on the side.


Welcome to Sloppy Joe's, hon. The meat's nice and fresh! What can I getcha?

Sloppy Joe needs a new menu every day. Your app will generate random sandwiches and prices for him.



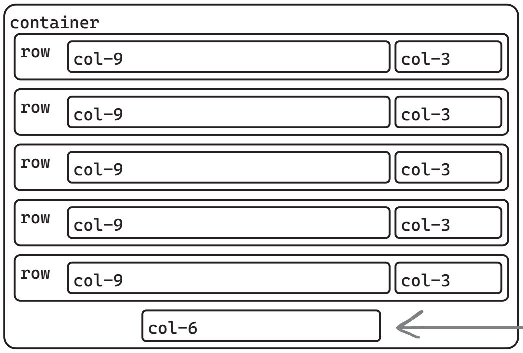
```
Roast beef with honey mustard on pumpernickel   $14.51
Pastrami with French dressing on rye            $9.05
Ham with brown mustard on wheat                 $9.12
Turkey with relish on everything bagel          $14.31
Tofu with yellow mustard on everything bagel    $10.29
              Add guacamole for $6.50
```

The prices are random numbers between 5.00 and 14.99.

Each sandwich is generated by choosing a random protein, random condiment, and random bread from arrays.

Sharpen your pencil

The menu page is made up of a series of Bootstrap rows, one for each menu item. Each row has two columns, a col-9 with the menu item description and a col-3 with the price. There's one last row on the bottom with a centered col-6 for the guacamole. Can you fill in the blank lines of HTML?



These rows are generated using a @foreach loop. Each row has two column <div>s in the 12-column Bootstrap grid layout, one with width 9 and one with width 3.

This picture shows how the page is laid out.

There's one more row at the bottom for the guacamole.

```
@rendermode InteractiveServer
@page "/"

<PageTitle>Welcome to Sloppy Joe's</PageTitle>


..................................................................................
  @foreach (MenuItem menuItem in menuItems)
  {

    ..................................................................................

          ..................................................................................
          @menuItem.Description
      </div>

          ..................................................................................
          @menuItem.Price
      </div>
  </div>
  }

  ..................................................................................

    ..................................................................................
          <strong>Add guacamole for @guacamolePrice</strong>
      </div>
  </div>
</div>
```

Sharpen your pencil Solution

Here's the Sharpen Your Pencil solution with the missing lines of HTML filled in. We also added a @code section to the bottom. This is the **complete *Home.razor* file** for your app.

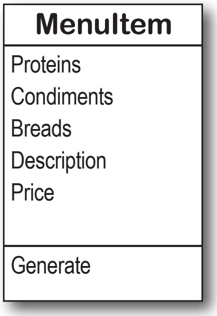
```
@rendermode InteractiveServer
@page "/"

<PageTitle>Welcome to Sloppy Joe's</PageTitle>

<div class="container">
  @foreach (MenuItem menuItem in menuItems)
   {
      <div class="row">
              <div class="col-9">
          @menuItem.Description
      </div>
              <div class="col-3">
          @menuItem.Price
      </div>
  </div>
  }
  <div class="row justify-content-center">
      <div class="col-6">
          <strong>Add guacamole for @guacamolePrice</strong>
      </div>
  </div>
</div>
```

You'll need this @using directive because you'll use a class in the SloppyJoe namespace.

```
@using SloppyJoe;
@code {
   MenuItem[] menuItems = new MenuItem[5];
   string? guacamolePrice;

   protected override void OnInitialized()
   {
      for (int i = 0; i < 5; i++)
      {
         menuItems[i] = new MenuItem();
         if (i >= 3)
             menuItems[i].Breads =
               [ "plain bagel", "onion bagel","pumpernickel bagel", "everything bagel" ];
         menuItems[i].Generate();

         MenuItem guacamoleMenuItem = new MenuItem();
         guacamoleMenuItem.Generate();
         guacamolePrice = guacamoleMenuItem.Price;
      }
   }
}
```

Here's the @code section with the C# code for your Home.razor file. It contains the two fields, menuItems and guacamolePrice, that were used by the Razor markup. It also has an OnInitialized method that sets up the page.

Exercise


MenuItem

Proteins
Condiments
Breads
Description
Price

Generate

Create a **new Blazor app called SloppyJoeBlazor**. Replace the Home.razor file with the code in the “Sharpen Your Pencil” solution.

Looking closely at the MenuItem class diagram. It has five fields: three arrays to hold the various sandwich parts, a description, and a price. The array fields use **collection expressions** that let you create an array by putting comma-separated values between [square brackets].

Add the MenuItem class to your project. Here's the code for the fields:

```
namespace SloppyJoe;

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 MenuItem class is in the SloppyJoe namespace.

The MenuItem class has three array fields that use collection expressions to set their values, just like the array you saw in Chapter 3 to store playing cards.

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 int 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

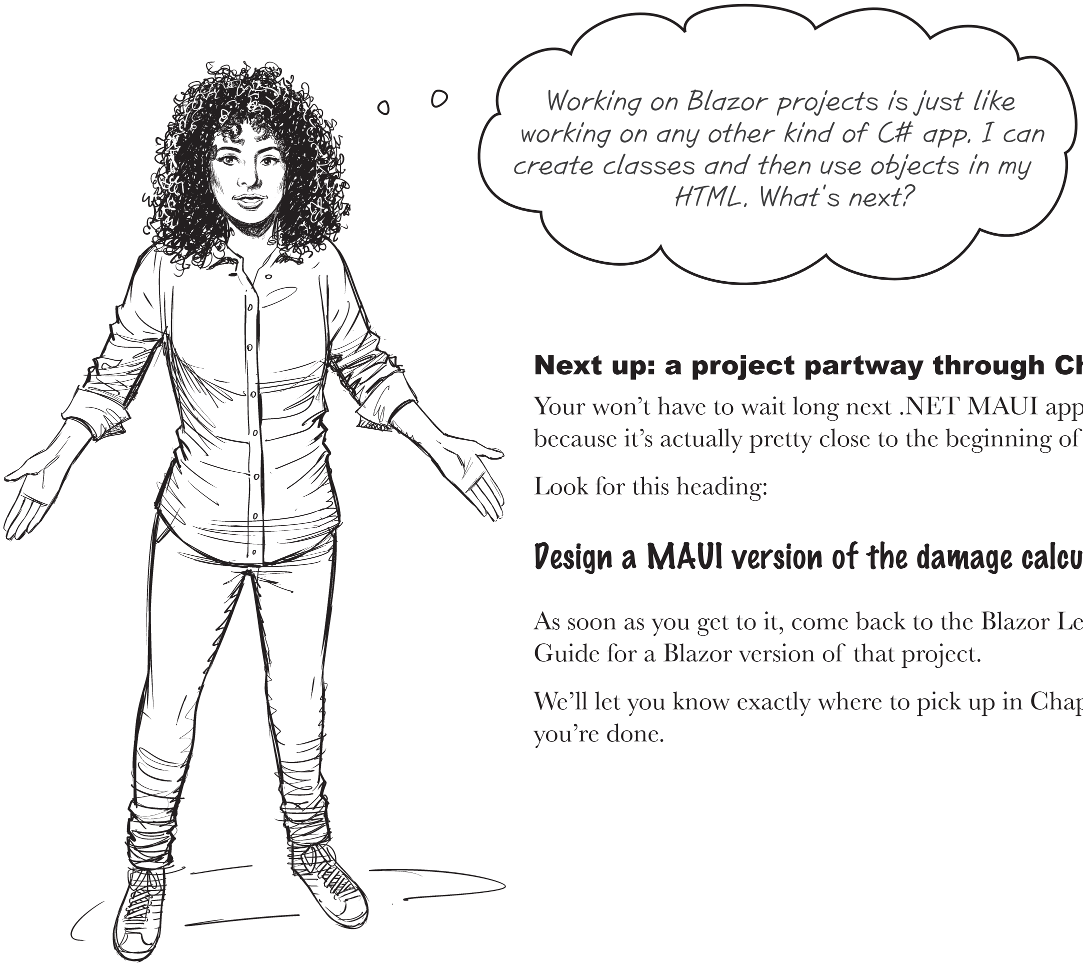
```
public void Generate()
{
    string protein = Proteins[Random.Shared.Next(Proteins.Length)];
    string condiment = Condiments[Random.Shared.Next(Condiments.Length)];
    string bread = Breads[Random.Shared.Next(Breads.Length)];
    Description = protein + " with " + condiment + " on " + bread;

    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");

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.
- An AI chatbot can read your code and **add comments**, including XML documentation (XMLDoc) comments.
- 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 to be null. Console.ReadLine can return null strings.
- You can use **Random.NextDouble** to create a random double value between 0 and 1. Multiply a random double to generate much larger random double values.
- Use **collection expressions** to initialize an array field by setting the field equal to a value starting with a square bracket, followed by a comma-delimited list of values, and ending with a square bracket.
- 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.


Working on Blazor projects is just like working on any other kind of C# app. I can create classes and then use objects in my HTML. What's next?

Next up: a project partway through Chapter 5.

Your won’t have to wait long next .NET MAUI app project, because it’s actually pretty close to the beginning of Chapter 5.

Look for this heading:

Design a MAUI version of the damage calculator app

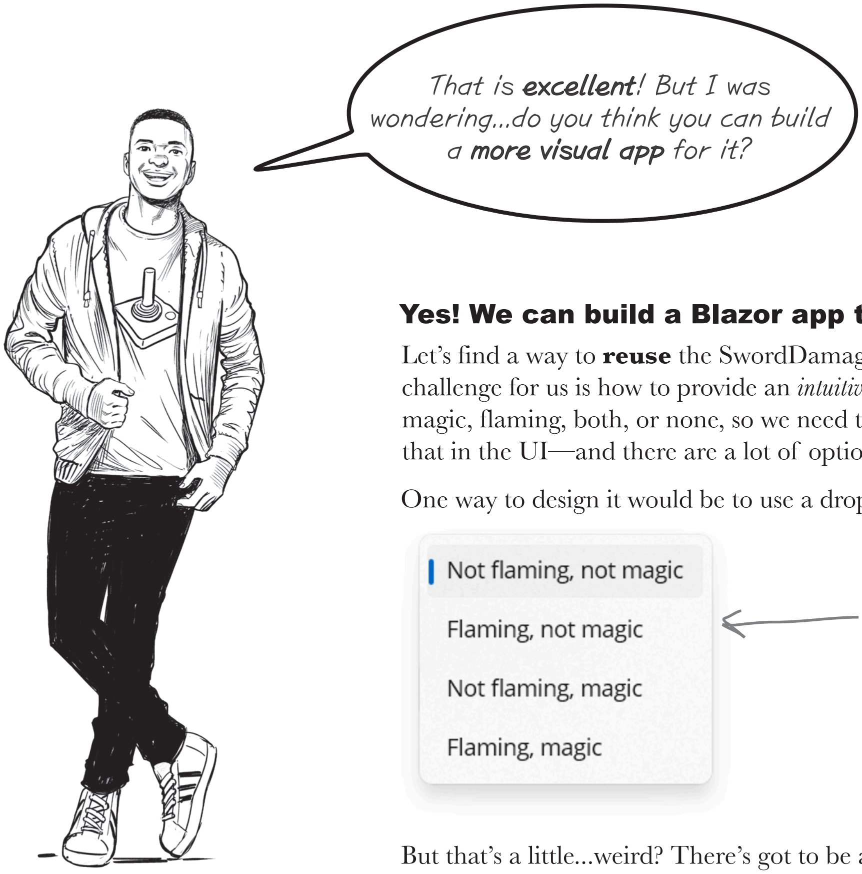
As soon as you get to it, come back to the Blazor Learner’s Guide for a Blazor version of that project.

We’ll let you know exactly where to pick up in Chapter 5 when you’re done.

The Blazor project starts after the first exercise solution

We gave you the code for a class called SwordDamage, and then challenged you with an exercise to write the code for a console app that uses it. You'll come back to the Learner's Guide right after you finish doing that exercise.

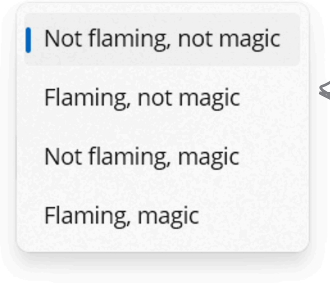
Here's what Owen told you after you finished the first exercise in Chapter 5.


That is **excellent**! But I was wondering...do you think you can build a **more visual app** for it?

Yes! We can build a Blazor app that uses the same class.

Let's find a way to **reuse** the SwordDamage class in a Blazor app. The first challenge for us is how to provide an *intuitive* user interface. A sword can be magic, flaming, both, or none, so we need to figure out how we want to handle that in the UI—and there are a lot of options.

One way to design it would be to use a dropdown with four options, like this:



We think using a dropdown for options would be a little weird. Do you agree?

But that's a little...weird? There's got to be a better way to design the app, right?

Design a Blazor version of the damage calculator app

Let's build a Blazor damage calculator app for Owen. We'll give you the @code section with C# code for the app. Your job will be to create the HTML that works with the C# code.

In this project, you'll be working with two new things that you haven't used yet:

- Your app will use two **checkboxes**. A checkbox is a control that should be very familiar to you—it's a box that displays a check when you click it, and is empty when you click it again. In MAUI, the Checkbox control has a Boolean value that's true if the box is checked and false if the box is unchecked.
- The C# code in your app will use **string interpolation** to build a string to display to the user. You've been using the + operator to build strings by concatenating values together. String interpolation does the same thing, but in a way that's easier to read.

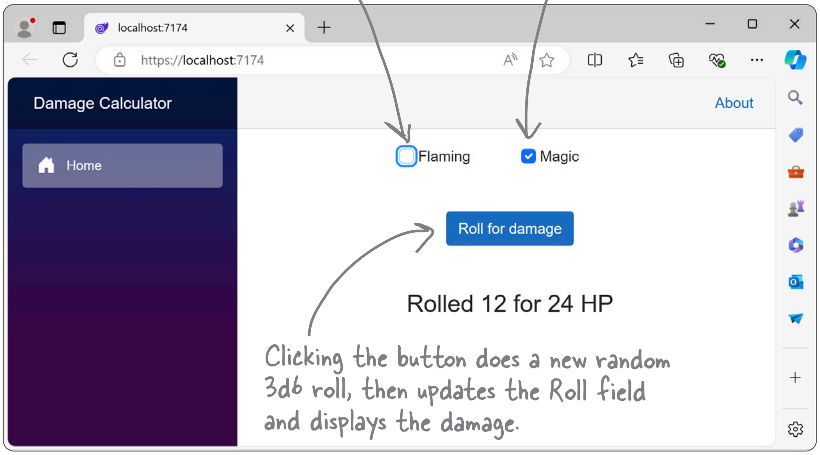
How your damage calculator app will work

Here's the main page for the damage calculator. It has two checkbox controls to turn flaming and magic on and off, a button to roll for damage, and an <h3> section to display the results:

- When you click the button, it generates three random numbers to do a 3d6 roll (just like the console app did), then uses the SwordDamage class to display the damage.
- Clicking on a checkbox causes the label to update automatically. When you check or uncheck either of the checkboxes, it updates the SwordDamage fields, recalculates the damage, and updates the label.

When you check the Flaming box, it calls the SwordDamage.SetFlaming, passing it true if the box is checked and false if it's unchecked, and then calls a method to update the label to display the damage.

The Magic checkbox works just like the Flaming one, except it calls SetMagic instead of SetFlaming.



Rolled 12 for 24 HP

Clicking the button does a new random 3d6 roll, then updates the Roll field and displays the damage.

Exercise

Create a new Blazor app called **BlazorDamageCalculator**.

Modify NavMenu.razor to change the menu title to “Damage Calculator” and remove the Counter and Weather menu items, just like we showed you in Chapter 2.

Next, **add the SwordDamage class** to the project. Don’t add a namespace directive—keep it in the global namespace. That will let your code in your Razor component access it without adding a @using directive.

Here’s the @code section for the *Home.razor* file:

```
@code {
    SwordDamage swordDamage = new SwordDamage();

    string damageText = "";

    private void UpdateFlaming(ChangeEventArgs e)
    {
        swordDamage.SetFlaming((bool)e.Value);
        DisplayDamage();
    }

    private void UpdateMagic(ChangeEventArgs e)
    {
        swordDamage.SetMagic((bool)e.Value);
        DisplayDamage();
    }

    protected override void OnInitialized()
    {
        swordDamage.SetMagic(false);
        swordDamage.SetFlaming(false);
        RollDice();
    }

    public void RollDice()
    {
        swordDamage.Roll = Random.Shared.Next(1, 7) +
            Random.Shared.Next(1, 7) + Random.Shared.Next(1, 7);
        DisplayDamage();
    }

    void DisplayDamage()
    {
       damageText = "Rolled " + swordDamage.Roll + " for " + swordDamage.Damage + " HP";
    }
}
```

The Flaming checkbox has this property: `@onchange="UpdateFlaming"` That will cause it to call the UpdateFlaming checkbox every time the user checks or unchecks the box. The page will send that value to the SwordDamage.SetFlaming method.

The Magic checkbox has a property that works the same way: `@onchange="UpdateMagic"`

The button has `@onclick="RollDice"` to generate a new random number and send the results to the SwordDamage object.

The DisplayDamage method updates the damageText field, which is displayed on the page like this: `<h3>@damageText</h3>`

There’s a *bug* in the code for this app! Can you spot it?

It’s not easy to find—don’t feel bad if you don’t see it yet!

Exercise

Your Blazor app will have two checkboxes to set the options for flaming and magic swords, a button to roll for damage, and text to display the results of the roll.

We'll lay it out with our familiar Bootstrap tags that we used in the previous projects:

- The page will have three rows. The bottom two rows have a top 5-space spacer (`mt-5`). Each row's content is centered (`justify-content-center`).
- The top row will have two `col-3` columns.
- The middle row will have a single `col-4` column.
- The bottom row will have a single `col-6` column.

Here's a new bit of Bootstrap to help lay things out.

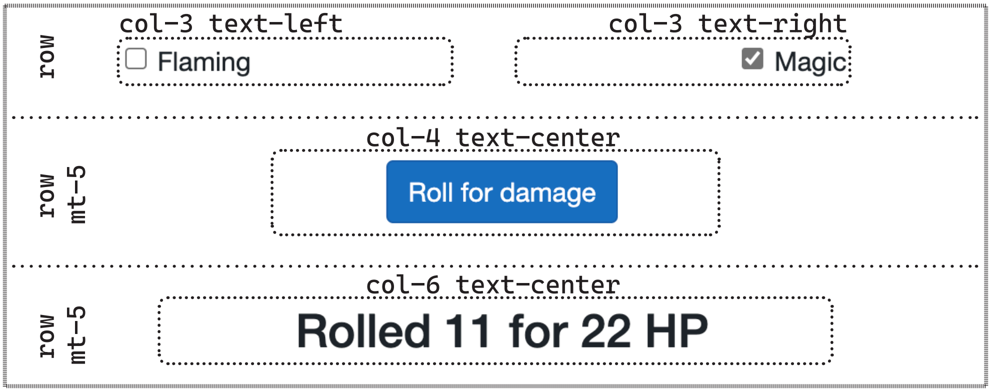
- The left column in the top row will use the **`text-left` class** to align its contents to the left side.
- The right column in the top row will use the **`text-right` class** to align its contents to the right side.
- The columns in the bottom rows will have the **`text-center` class**, which tells them to center their contents.

The last piece of the puzzle is the markup to create a checkbox. Here's the HTML for the flaming sword checkbox:

```
<div class="col-3 text-left">
    <input class="form-check-input" type="checkbox" id="flaming" />
    <label class="form-check-label" for="flaming">
        Flaming
    </label>
</div>
```

This markup uses the same <input> tag that you used for sliders and other controls. Setting `type="checkbox"` tells it to create a checkbox. We also added a <label> to add the "Flaming" text next to the checkbox. The `for` attribute on the label matches the `id` attribute on the input, which is how the page knows which input the label is associated with.

Here's what the page looks like:



Your job is to **add the HTML and Razor markup** to the *Home.razor* file. Make sure you also add the @code section that we gave you. Remember, it's ***not cheating*** to peek at the solution!

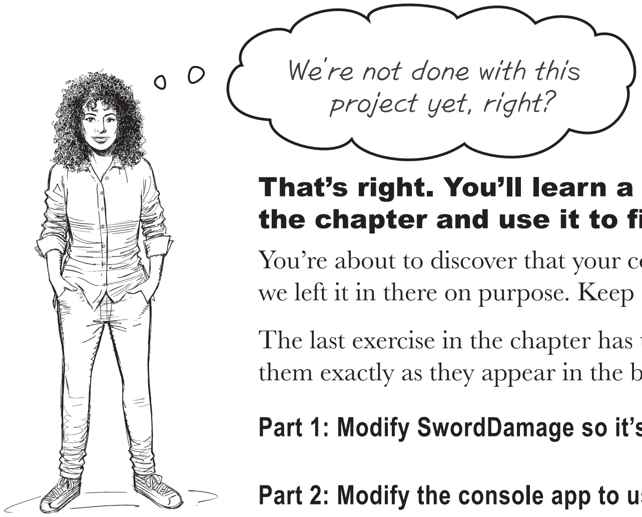
Exercise Solution

Your job was to **add the HTML and Razor markup** to the *Home.razor* file. Make sure you also add the @code section that we gave you to the bottom of the file.

```
@rendermode InteractiveServer
@page "/"

<PageTitle>Damage Calculator</PageTitle>

<div class="container">
    <div class="row justify-content-center">
        <div class="col-3 text-left">
            <input class="form-check-input" type="checkbox" id="flaming"
                   @onchange="UpdateFlaming" />
            <label class="form-check-label" for="flaming">
                Flaming
            </label>
        </div>
        <div class="col-3 text-right">
            <input class="form-check-input" type="checkbox" id="magic"
                   @onchange="UpdateMagic" />
            <label class="form-check-label" for="magic">
                Magic
            </label>
        </div>
    </div>
    <div class="row justify-content-center mt-5">
        <div class="col-4 text-center">
            <button type="button" class="btn btn-primary"
                    @onclick="RollDice">
                Roll for damage
            </button>
        </div>
    </div>
    <div class="row justify-content-center mt-5">
        <div class="col-6 text-center">
            <h3>@damageText</h3>
        </div>
    </div>
</div>

@code {

   // We gave you this code earlier

}
```

Tabletop talk (or maybe...dice discussion?)

Go back to Chapter 5 and find this heading—that's where to start again. Keep reading the chapter until you get to the very last exercise. Do parts 1 and 2 of that exercise. We'll give you a Blazor version of Part 3 here.



That's right. You'll learn a lot about encapsulation throughout the chapter and use it to fix the bug.

You're about to discover that your code has a bug in it! Don't worry, it's not your fault—we left it in there on purpose. Keep reading through the chapter until the very end.

The last exercise in the chapter has three parts. The chapter has the first two parts—do them exactly as they appear in the book:

Part 1: Modify SwordDamage so it's a well-encapsulated class

Part 2: Modify the console app to use the well-encapsulated SwordDamage class

Then do Part 3 below, which has you update your Blazor app.

Exercise

Part 3: Modify the Blazor app to use the well-encapsulated SwordDamage class

1. Copy the code from Part 1 into a new Blazor web app. Copy the HTML markup from earlier in the chapter.
2. Modify the markup:
   - Replace everything between <h3> and </h3> with to bind directly to the SwordDamage object <h3>**`Rolled @swordDamage.Roll for @swordDamage.Damage HP`**</h3>
   - Replace **`@onchange="UpdateFlaming"`** with **`@bind="swordDamage.Flaming"`**
   - Replace **`@onchange="UpdateMagic"`** with **`@bind="swordDamage.Magic"`**
3. Modify the code in the **`@code { }`** section at the bottom of your *Index.razor* file:
   - Your new SwordDamage class no longer has a CalculateDamage method, so remove lines that call it.
   - You removed the SetFlaming and SetMagic methods from your SwordDamage class, so remove all calls to those methods. You're not using the UpdateFlaming or UpdateMagic event handlers, so remove those too.
   - Now that you modified the "Rolled ... for ... HP" line at the bottom of the page to bind directly, you can delete the DisplayDamage method. Delete the damageText field, and all calls to the DisplayDamage method, too.
   - The new SwordDamage class has a constructor with one parameter—just pass it 10. It doesn't matter what value you use here, because you'll roll the dice when the page is initialized.

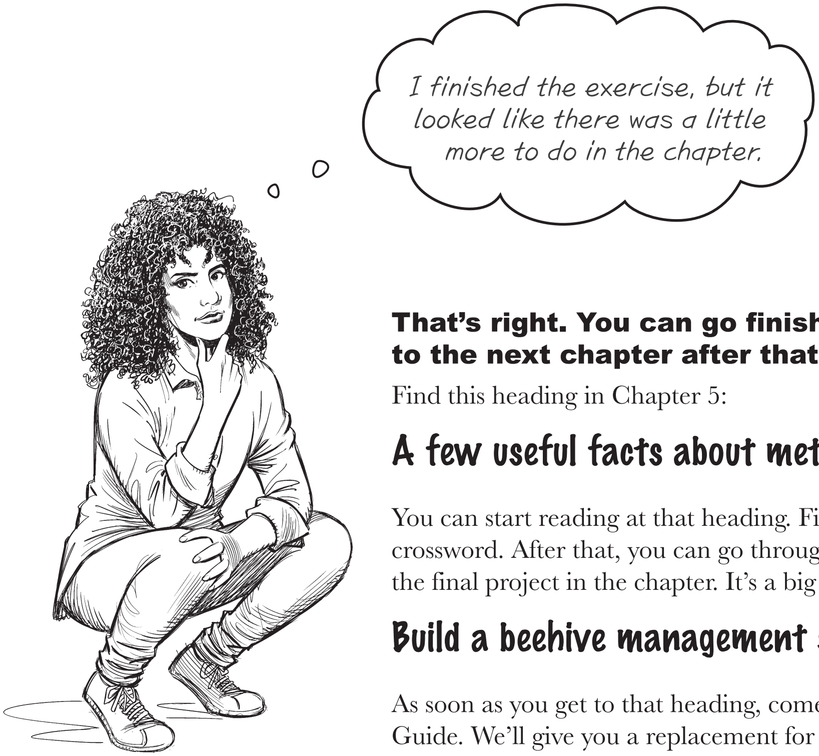
Test everything. Use the debugger or Debug.WriteLine statements to make sure that it all REALLY works.

Exercise Solution

Here’s the complete *Index.razor* file for your Blazor web app, including HTML markup and C# code. Did you notice how much less C# code you need in it? That's one way well-encapsulated classes help you write better code—you don't need to write as much additional code to use them.

The updated code for the SwordDamge class is in the solution in the book.

```
@rendermode InteractiveServer
@page "/"

<PageTitle>Damage Calculator</PageTitle>

<div class="container">
    <div class="row justify-content-center">
        <div class="col-3 text-left">
            <input class="form-check-input" type="checkbox" id="flaming"
                   @bind="swordDamage.Flaming" />
            <label class="form-check-label" for="flaming">
                Flaming
            </label>
        </div>
        <div class="col-3 text-right">
            <input class="form-check-input" type="checkbox" id="magic"
                   @bind="swordDamage.Magic" />
            <label class="form-check-label" for="magic">
                Magic
            </label>
        </div>
    </div>
    <div class="row justify-content-center mt-5">
        <div class="col-4 text-center">
            <button type="button" class="btn btn-primary"
                    @onclick="RollDice">
                Roll for damage
            </button>
        </div>
    </div>
    <div class="row justify-content-center mt-5">
        <div class="col-6 text-center">
            <h3>Rolled @swordDamage.Roll for @swordDamage.Damage HP</h3>
        </div>
    </div>
</div>

@code {
    SwordDamage swordDamage = new SwordDamage(10);

    protected override void OnInitialized()
    {
        RollDice();
    }

    public void RollDice()
    {
        swordDamage.Roll = Random.Shared.Next(1, 7) +
            Random.Shared.Next(1, 7) + Random.Shared.Next(1, 7);
    }
}
```


I finished the exercise, but it looked like there was a little more to do in the chapter.

That's right. You can go finish Chapter 5 and move on to the next chapter after that.

Find this heading in Chapter 5:

A few useful facts about methods and properties

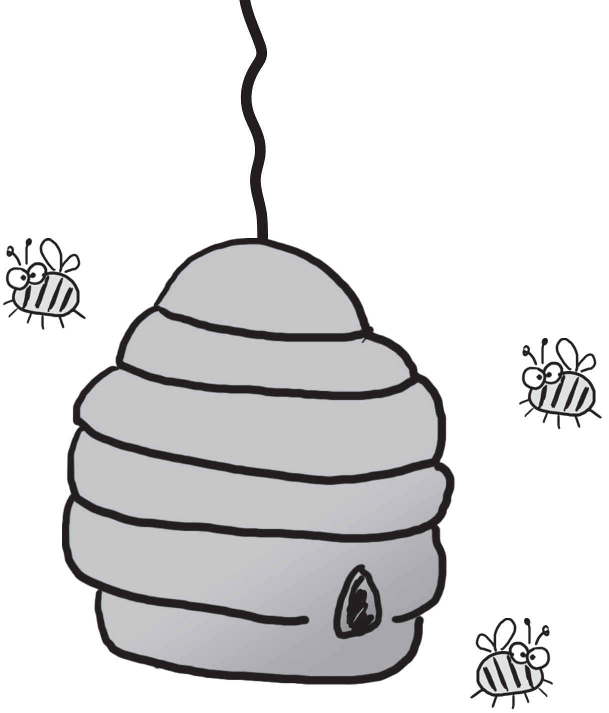
You can start reading at that heading. Finish the chapter, including the crossword. After that, you can go through all of Chapter 6 until you get to the final project in the chapter. It's a big project that starts with this heading:

Build a beehive management system

As soon as you get to that heading, come back to the Blazor Learner's Guide. We'll give you a replacement for the first part of the project to build a Blazor version of the UI, then you'll be able to return to the book to finish the project (with just one small section to skip).

Build a beehive management system

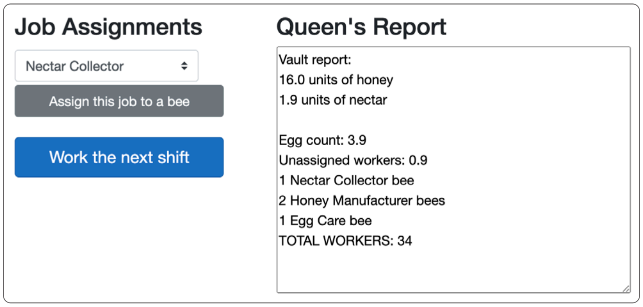
The queen bee needs your help! Her hive is out of control, and she needs a program to help manage her honey production business. She's got a beehive full of workers, and a whole bunch of jobs that need to be done around the hive, but somehow she's lost control of which bee is doing what, and whether or not she's got the beepower to do the jobs that need to be done. It's up to you to build a **beehive management system** to help her keep track of her workers. Here's how it'll work.



1. **The queen assigns jobs to her workers.**
There are three different jobs that the workers can do. **Nectar collector** bees fly out and bring nectar back to the hive. **Honey manufacturer** bees turn that nectar into honey, which bees eat to keep working. Finally, the queen is constantly laying eggs, and **egg care** bees make sure they become workers.

2. **When the jobs are all assigned, it's time to work.**
Once the queen's done assigning the work, she'll tell the bees to work the next shift. At the end of the shift, she gets a shift report that tells her how many bees are assigned to each job and the status of the nectar and honey in the **honey vault**.

3. **Help the queen grow her hive.**
Like all business leaders, the queen is focused on **growth**. The beehive business is hard work, and she measures her hive in the total number of workers. Can you help the queen keep adding workers? How big can she grow the hive before it runs out of honey and she has to file for bee-nkruptcy?



```
Vault report:
16.0 units of honey
1.9 units of nectar

Egg count: 3.9
Unassigned workers: 0.9
1 Nectar Collector bee
2 Honey Manufacturer bees
1 Egg Care bee
TOTAL WORKERS: 34
```

This is a *bigger project* than the ones in the last few chapters.

This is a big project. You can do this!

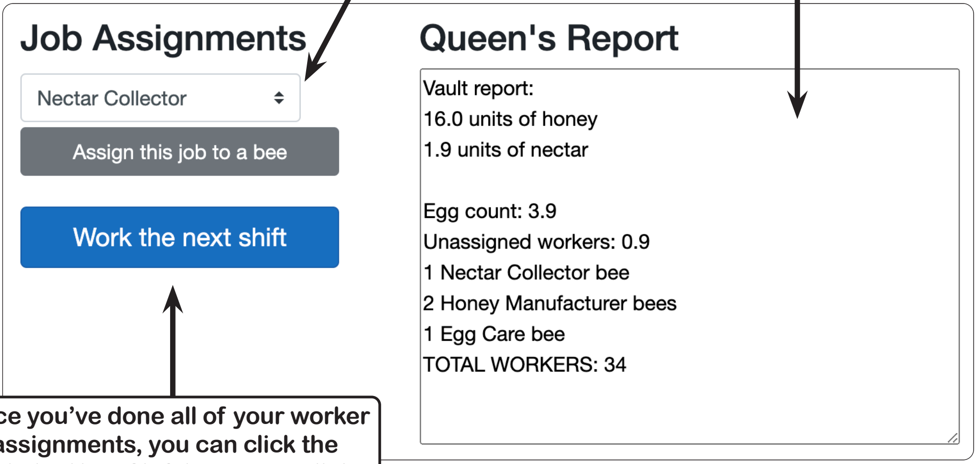
The main goal of this book is to help you learn C#. But we'll also teach important skills that can help you ***become a great developer***. One way to do that is to help show you how to work on—and finish!—larger projects. When you did the Animal Matching Game project in Chapter 1, you broke it down into smaller pieces. You'll do the same for the Beehive Management System project. **First** you'll create the XAML for the main page, **then** you'll do a "Sharpen your pencil" exercise to complete the code for several of the classes, and **finally** you'll do an exercise to finish the rest of the code for the project.

How the Beehive Management System app works

When the app starts, the honey vault has 25 units of honey and 100 units of nectar, and the hive has three workers: a nectar collector bee, a honey manufacturer bee, and an egg care bee. The first shift report delivered is displayed on the righthand side of the app.

If there are any unassigned workers at the start of the shift, you can use the dropdown to choose a job, then click the Assign button to give a worker that job (it's disabled if there aren't enough unassigned workers). If there are multiple unassigned workers, you can do this more than before the next shift.

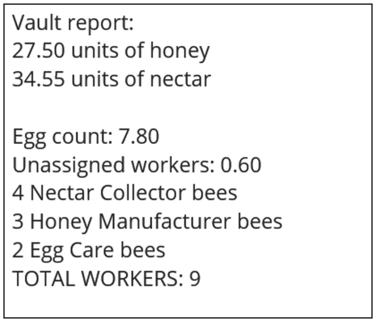
This is the shift report that the queen generates at the end of every shift. It shows the status of the honey vault, followed by the number of eggs, unassigned workers, and how many workers there are of each type.



```
Vault report:
16.0 units of honey
1.9 units of nectar

Egg count: 3.9
Unassigned workers: 0.9
1 Nectar Collector bee
2 Honey Manufacturer bees
1 Egg Care bee
TOTAL WORKERS: 34
```

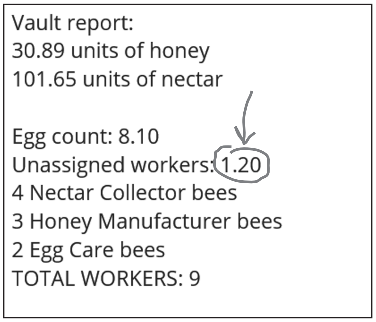
Once you've done all of your worker assignments, you can click the Work the Next Shift button to tell the workers to work the next shift.

Each worker consumes honey to do a job. The numbers change at the end of each shift to show what they did.



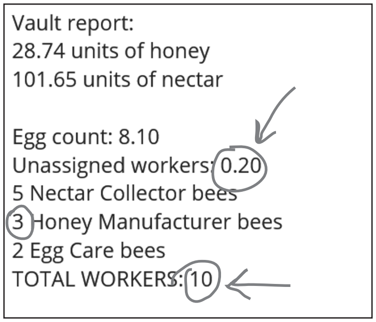
```
Vault report:
27.50 units of honey
34.55 units of nectar

Egg count: 7.80
Unassigned workers: 0.60
4 Nectar Collector bees
3 Honey Manufacturer bees
2 Egg Care bees
TOTAL WORKERS: 9
```

If the Unassigned Workers count is at least 1, clicking the Assign button assigns a worker to the selected job.



```
Vault report:
30.89 units of honey
101.65 units of nectar

Egg count: 8.10
Unassigned workers: 1.20
4 Nectar Collector bees
3 Honey Manufacturer bees
2 Egg Care bees
TOTAL WORKERS: 9
```

Assigning a worker makes the unassigned workers go down by 1 and the total workers increase by 1.

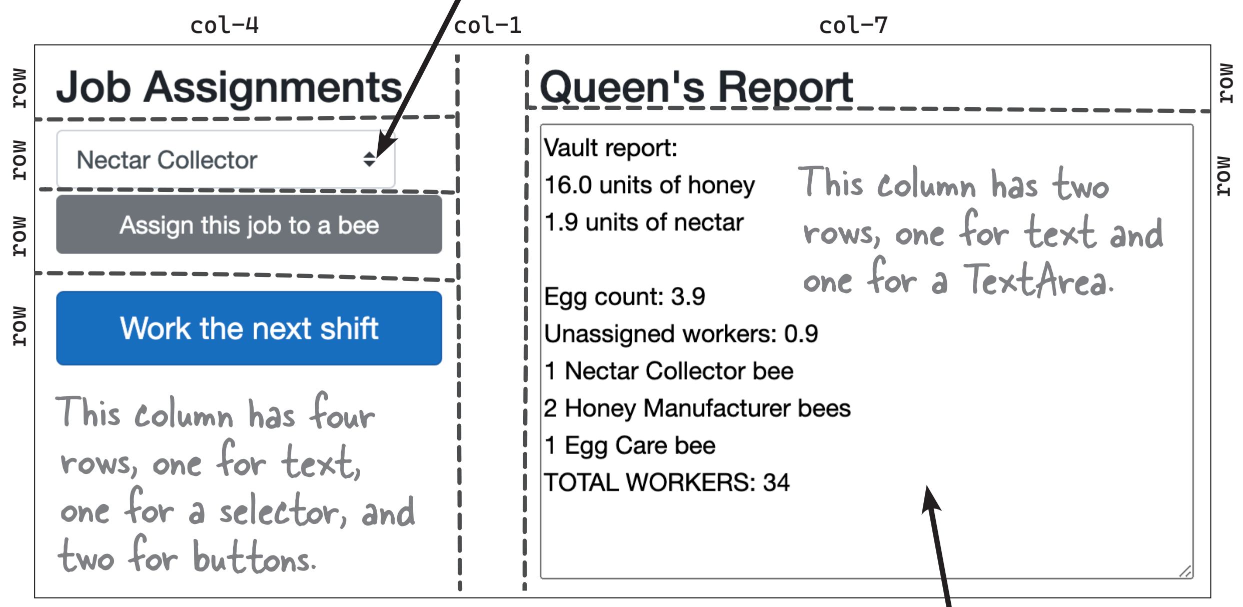


```
Vault report:
28.74 units of honey
101.65 units of nectar

Egg count: 8.10
Unassigned workers: 0.20
5 Nectar Collector bees
3 Honey Manufacturer bees
2 Egg Care bees
TOTAL WORKERS: 10
```

How the main window is designed

Create a **new Blazor web app called BlazorBeehiveManagementSystem**. Here's the HTML markup for the main window, including its **entire `@code` section**. It has three columns: one for job assignments, an empty divider column, and a column for the Queen's report. The left column contains four rows: one for the header, one for the job dropdown, one for the job assignment button, and one for the button to work the next shift

This is a dropdown control (or select control), just like you used in Chapter 2. It gives you a list of options to choose from. You'll use it to let the user choose a job to assign and `@bind` its value to a field called `selectJob`. In Chapter 2 you used a @foreach to create the options, but now you'll create seprate <option> tags for each of the three options.



```
Vault report:
16.0 units of honey
1.9 units of nectar

Egg count: 3.9
Unassigned workers: 0.9
1 Nectar Collector bee
2 Honey Manufacturer bees
1 Egg Care bee
TOTAL WORKERS: 34
```

This column has four rows, one for text, one for a selector, and two for buttons.

This column has two rows, one for text and one for a TextArea.

This is a TextArea control. It displays multiple lines of text. You'll use `@bind` data binding to make it display the Queen object's StatusReport property.

Watch it!

Put your classes in the right namespace (or add a @using directive).

You'll be creating a **project called BlazorBeehiveManagementSystem***, including classes that the code in your Home.Razor file will use. You have two options for which namespace to use. You can add* `namespace BlazorBeehiveManagementSystem;` *to the top of each class file to put it in the same namespace as your app, or you can put the classes in a different namespace and use a @using directive in your Home.razor to use that namespace.*

Exercise

Create a new Blazor web app called **BlazorBeehiveManagementSystem**. Your job in this exercise is to add the HTML markup for the main window.

Before you get started, you'll need to add this Queen class to your project—it has methods that the buttons will call and a property that you'll bind to. You'll fill in the rest of the class later on in the project:

```
public class Queen
{
    public void AssignBee(string selectedJob) { /* You'll fill this in later */ }

    public void WorkTheNextShift() { /* You'll fill this in later */ }

    public string StatusReport { get; private set; } = "";
}
```

You'll need to add this class to your project to get the @bind and @onclick data binding to build.

Here's the entire `@code` section for your *Home.razor* file:

```
@code {
    Queen queen = new Queen();
    string selectedJob = "Nectar Collector";
}
```

Your job is to create the HTML markup for the main window. Look carefully at the layout that we just showed you.

- The page has three columns: one for job assignments, an empty divider column, and a column for the Queen's report.
- The left column contains four rows: one for the <h3> header text, one for the job dropdown, one for the job assignment button, and one for the button to work the next shift.
- The right column has two rows, one for the <h3> header text and one for a TextArea control.

Here are a few useful tips that will help you create your form:

Any HTML element can be its own row.

- You can make a text tag like <h3> its own row like this: `<h3 class="row">This is a separate row</h3>`
- The dropdown uses a <select> control, with three options: Nectar Collector, Honey Manufacturer, and Egg Care. It works just like the <select> that you used in Chapter 2, except instead of using a @foreach loop to create its options, you'll add three separate <option> tags between the opening <select> and closing </select> tags. Give it the property `class="row mt-4"` to put it on its own row with a top margin. Look carefully at how the @foreach loop works. Can you figure out the properties to add to each <option> tag?
- The "Work the next shift" button is a **primary button**, which means that it represents the most important action for the user on the page, and it gets clicked when the user presses the Enter key. Here's the code for it:

```
<button type="button" class="col btn btn-lg btn-primary"
        @onclick="() => queen.WorkTheNextShift()">
```

- The "Assign this job to a bee" button is a **secondary button**. You can only have one primary button, but you can have many secondary buttons. Here's the code for it:

```
<button type="button" class="col btn btn-small btn-secondary"
        @onclick="() => queen.AssignBee(selectedJob)">
```

- A **TextArea control** lets you display or enter multi-line text. Use this HTML for a read-only text area with 12 rows:

```
<textarea class="row" rows="12" cols="50"
          value="@queen.StatusReport" readonly />
```

Exercise Solution

Here's the complete code for the *Home.razor* file.

```
@rendermode InteractiveServer
@page "/"
@using BlazorBeehiveManagementSystem;

<PageTitle>Beehive Management System</PageTitle>

<div class="container">
   <div class="row">

      <div class="col-4">
         <h3 class="row">Job Assignments</h3>

         <select type="row mt-4" @bind="selectedJob">
            <option value="Nectar Collector">Nectar Collector</option>
            <option value="Honey Manufacturer">Honey Manufacturer</option>
            <option value="Egg Care">Egg Care</option>
         </select>

         <div class="row mt-4">
            <button type="button" class="col btn btn-small btn-secondary"
                  @onclick="() => queen.AssignBee(selectedJob)">
               Assign this job to a bee
            </button>
         </div>

         <div class="row mt-4">
            <button type="button" class="col btn btn-lg btn-primary"
                  @onclick="() => queen.WorkTheNextShift()">
               Work the next shift
            </button>
         </div>
      </div>

      <div class="col-1" />

      <div class="col-7">
         <h3 class="row">Queen's Report</h3>
         <textarea class="row" rows="12" cols="50"
                    value="@queen.StatusReport" readonly />
      </div>

   </div>
</div>

@code {
    Queen queen = new Queen();
    string selectedJob = "Nectar Collector";
}
```

The left column has four rows: one with <h3> text, one with a dropdown, one with a secondary button, and one with a primary button.

The <select> has a class property to put it in its own row and give it a top margin.

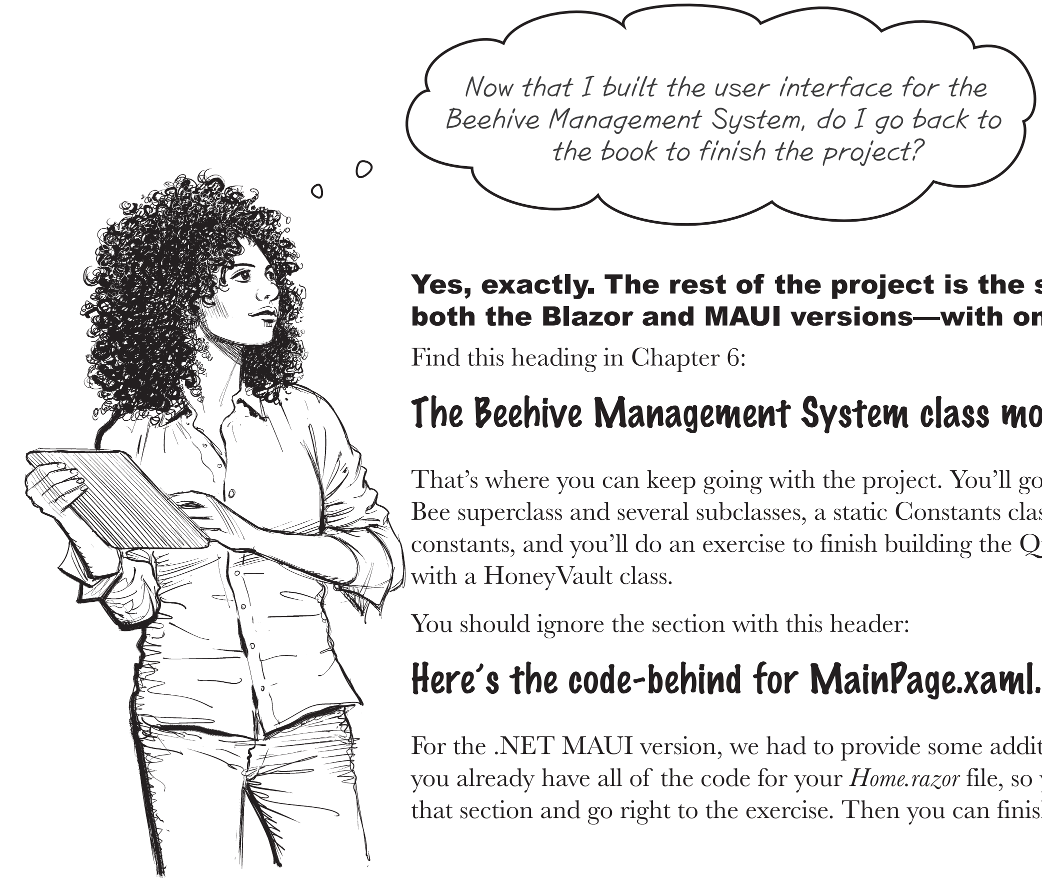
The dropdown uses these three <option> tags to determine the options the user can choose from.

We gave you the code for the two buttons. Look at the "class" properties—try experimenting with swapping them around. How does that change the buttons?

Here's the middle column.

We gave you the HTML code for this textarea. What happens if you change the rows and columns properties, or remove the readonly property?

We gave you the @code section and a minimal Queen class with empty methods so the project builds.


Now that I built the user interface for the Beehive Management System, do I go back to the book to finish the project?

Yes, exactly. The rest of the project is the same for both the Blazor and MAUI versions—with one tweak.

Find this heading in Chapter 6:

The Beehive Management System class model

That's where you can keep going with the project. You'll go on to create the Bee superclass and several subclasses, a static Constants class to hold your constants, and you'll do an exercise to finish building the Queen class, along with a HoneyVault class.

You should ignore the section with this header:

Here's the code-behind for MainPage.xaml.cs

For the .NET MAUI version, we had to provide some additional code. But you already have all of the code for your *Home.razor* file, so you can just skip that section and go right to the exercise. Then you can finish the chapter.

You also need to add this code to your page disable the "Work the next shift" button when the hive runs out of honey

The Queen.WorkTheNextShift method returns true if there is still honey in the honey vault, or false if it ran out of honey. Add a boolean field called outOfHoney to track when the hive runs out of honey:

```
@code {
bool outOfHoney = false;
Queen queen = new Queen();
```

This field will be set to true when the honey vault runs out of honey.

Then modify the HTML for the button to set the field when the button calls the WorkTheNextShift method, and use its disabled property to disable the button when the honey vault is out of honey:

```
<div class="row mt-4">
    <button type="button" class="col btn btn-lg btn-primary"
        disabled="@outOfHoney"
        @onclick="() => outOfHoney = !queen.WorkTheNextShift()">
           Work the next shift
    </button>
</div>
```

This property disables the button if outOfHoney is true.

This is where the button calls the WorkTheNextShift button. This change uses ! to the response, and sets the outOfHoney field to that value.

Look for a page in the book with this heading. Here's a replacement for that page.

The Beehive Management System is turn-based… now let's convert it to real-time

A **turn-based game** is a game where the flow is broken down into parts—in the case of the Beehive Management System, into shifts. The next shift doesn't start until you click a button, so you can take all the time you want to assign workers. We can use a timer—like the one you used in Chapter 1—to **convert it to a real-time game** where time progresses continuously…and we can do it with just a few lines of code.

1 Start a timer in the OnInitialized method.

You've used the OnInitialized method to run code when the app first starts: Now do the same thing to start a time:

```
@code {
    bool outOfHoney = false;
    Queen queen = new Queen();
    string selectedJob = "Nectar Collector";

    @using System.Timers
    Timer timer;
    protected override void OnInitialized()
    {
        timer = new Timer(1500);
        timer.Elapsed += Timer_Elapsed;
        timer.Start();
    }
```

This method starts a timer that calls the Timer_Elapsed method every 1.5 seconds.

You used a timer just like this in Chapter 1 to add a timer to your animal matching game. This code is very similar to the code you used in Chapter 1. Take a few minutes and flip back to that project to remind yourself how the timer works.

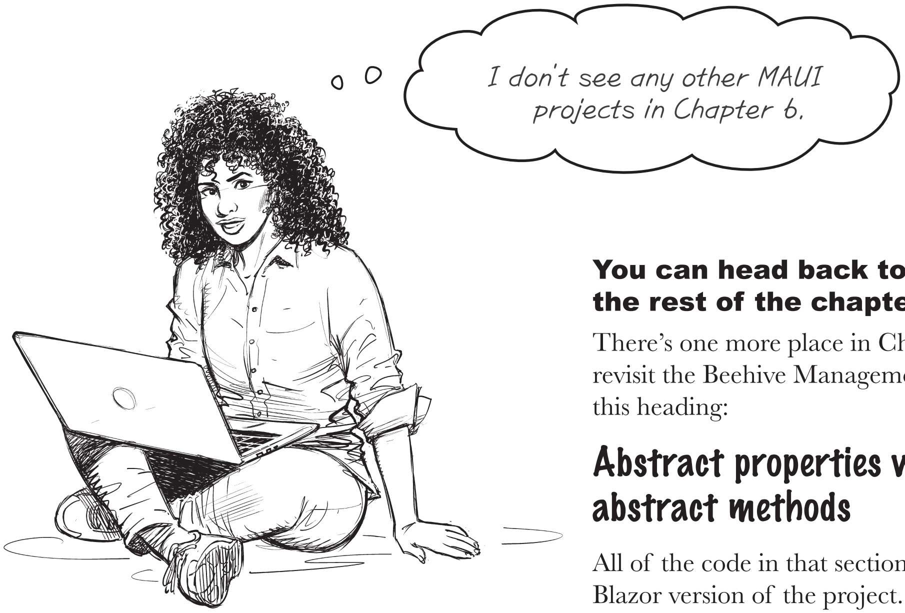
2 Add the Timer_Elapsed method to work the next shift on each "tick."

We want the timer to keep the game moving forward, so we can have it automatically trigger the next shift if the player hasn't done it already. Here's the code for the method:

```
    private void Timer_Elapsed(object? sender, ElapsedEventArgs e)
    {
        InvokeAsync(() =>
        {
            if (!outOfHoney)
            {
                outOfHoney = !queen.WorkTheNextShift();
                StateHasChanged();
            }
        });
    }
```

If the TimerTick method returns false, the timer stops running. This `if` statement keeps the timer from trying to work the next shift if the hive has run out of honey.

Now run your game. A new shift starts every 1.5 seconds, whether or not you click the button. This is a small change to the mechanics, but it ***dramatically changes the dynamics of the game***, which leads to a huge difference in aesthetics. It's up to you to decide if the game is better as a turn-based or real-time simulation.



You can head back to the book for the rest of the chapter.

There's one more place in Chapter 6 where you revisit the Beehive Management System. Look for this heading:

Abstract properties work just like abstract methods

All of the code in that section works with the Blazor version of the project. There's also an exercise in Chapter 7 where you add an interface called IWorker to the Beehive Management System, but that also works with the Blazor version.

Relax

There's no Blazor equivalent for the .NET MAUI section in Chapter 7.

There's a .NET MAUI project in Chapter 7 that starts with this heading:

Data binding updates MAUI controls automatically

*That project is about data binding in .NET MAUI. It's in that chapter because it uses an interface. You've used data binding with Blazor since Chapter 2, so there's no equivalent Blazor project. That means **you can skip the entire project** and start again at this heading near the end of the chapter:*

Polymorphism means that one object can take many different forms

Keep going until you reach this heading at the end of Chapter 8:

CollectionView is a MAUI control built for displaying collections

Then come back to the Blazor Learner's Guide for the next Blazor project.

A list box shows you a list of items that you can pick

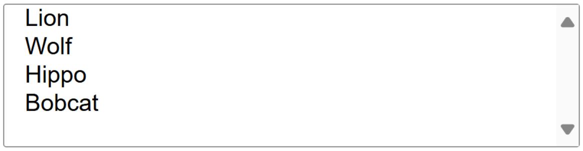
There's a close relative of the dropdown list called the **list box**. A list box lets you choose an item from a list, but unlike a dropdown it displays the items in a box. A list box ***uses the same `<select>` tag*** as the dropdown that you used to pick birds in Chapter 2 or bee jobs in Chapter 6. To turn a `<select>` into a list box, all you have to is **give it a `size` property** that tells it how many items to display before showing scrolling them.

Here's an example of a list box that could display animals in your zoo simulator from Chapter 6. It uses `<select>` tag with a **`size="5"`** property to turn it into a list box that displays five items:

```
<select size="5">
    <option value="Lion">Lion</option>
    <option value="Wolf">Wolf</option>
    <option value="Hippo">Hippo</option>
    <option value="Bobcat">Bobcat</option>
</select>
```

This size property makes the <select> appear as a list box that displays five items.

Here's how the list box is displayed on the page—it's only got four items, so there's space underneath Bobcat:

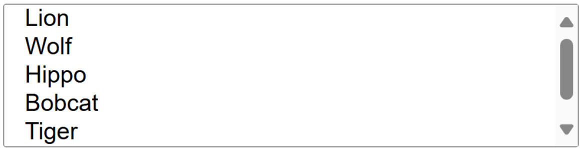


Lion
Wolf
Hippo
Bobcat

Let's add two more items to the list:

```
<select size="5">
    <option value="Lion">Lion</option>
    <option value="Wolf">Wolf</option>
    <option value="Hippo">Hippo</option>
    <option value="Bobcat">Bobcat</option>
    <option value="Tiger">Tiger</option>
    <option value="Dog">Dog</option>
</select>
```

The `<select>` tag still has **`size="5"`** so the list box only displays five items at a time. Since there are six items in the list, the list box now has a scroll bar so you can scroll down to see the list item:



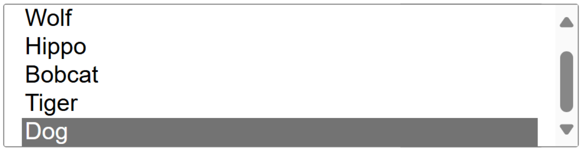
Lion
Wolf
Hippo
Bobcat
Tiger

When the number of options is bigger than the size, the list box adds a scroll bar.

Data binding works just like it did with the dropdown: `<select size="5" @bind="animal">`

Blazor uses **two-way binding**, which means you can ***change the selected item*** by setting the bound variable:

```
    animal = "Dog";
```



Wolf
Hippo
Bobcat
Tiger
Dog

The <select> is bound to the animal variable. Setting the variable to one of the items in the list causes the list box to select that item. If the list is currently scrolled so the item isn't currently displayed, the list box will scroll to make it visible.

Create a new Blazor Web app that uses a list box

Let's create a new app that we'll use to learn about how list boxes work, and get some practice working with collections. We'll start with a list box that displays the familiar list of birds. Later in the chapter we'll modify it to use your Card class to work with decks of cards instead.

Do this!

1. **Create a new Blazor Web App called BlazorCards.**
For now, we'll display a familiar array of birds. But later on, we'll replace them with cards, so ***make sure you use the name BlazorCards*** for your app. Modify *NavMenu.razor* to remove the Counter and Weather sections, and **set the page title** to `"Blazor cards"`.

2. **Replace the HTML with a page that has a list box.**
Open *Home.razor* and delete everything. Replace it with this code:

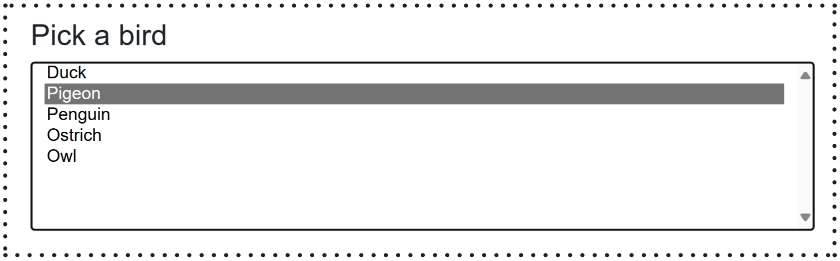
```
@page "/"
@rendermode InteractiveServer
<PageTitle>Blazor Cards</PageTitle>

<div class="container">
    <h3 class="row">Pick a bird</h3>
    <div class="row">
        <select class="col-12" size="8">
            @foreach (string bird in birds)
            {
                <option value="@bird">@bird</option>
            }
        </select>
    </div>
</div>

@code {
    private List<string> birds = [
        "Duck",
        "Pigeon",
        "Penguin",
        "Ostrich",
        "Owl"
    ];
}
```

The <select> tag in this app looks almost identical to the one you used for a dropdown. The only difference is the size="8" property.

This @foreach loop is just like the one you used in Chapter 2.

In Chapter 2 you used a string array for the list of birds. Now you can use a List<string> instead, but you can still use the same collection expression.

3. **Run your app.**
Try clicking on different birds in the list box—it selects each bird that you clicked on.


Pick a bird

Duck
Pigeon
Penguin
Ostrich
Owl

Make your app work with Card objects

Let's make your **list box work with objects**, not just strings—specifically, card objects. You created a Card class, Suits and Values enums, and a CardComparerByValue class. Now you'll reuse them in a Blazor app.

1. **Add your Card class, CardComparerByValue class, Suits enum, and Values enum.**
   The list box automatically calls the ToString method for any item that it displays, so make sure you use the version of the Card class ***that has the ToString method***. Add the existing files to your project just like you did in Chapter 3 (in Visual Studio: right-click on the project and choose Add >> Existing Item and add each file; in VSCode: right-click on the project and choose Reveal in Explorer/Finder and drag the files onto the project). Make sure the classes and enums are in the **BlazorCards namespace**.

2. **Change your HTML to work with cards instead of birds.**
   Here's the updated Home.razor file. We **added a row** to show the selected card and **added a button** that calls the the AddBird method, which adds a random card to the list and updates selectedCard to select it:

```
@page "/"
@rendermode InteractiveServer
@using BlazorCards
<PageTitle>Blazor Cards</PageTitle>

<div class="container">
    <h3 class="row">Pick a card</h3>
    <div class="row">
        <select class="col-12" size="8" @bind="selectedCard">
            @foreach(Card card in cards)
            {
                <option value="@card">@card</option>
            }
        </select>
    </div>

    <h4 class="row">You selected: @selectedCard</h4>

    <button type="button" class="row mt-2 btn btn-primary"
            @onclick="AddCard">Add a card</button>
</div>

@code {
    string? selectedCard;
    private List<Card> cards = new List<Card>();

    private void AddCard()
    {
        cards.Add(new Card((Values)Random.Shared.Next(1, 14),
                           (Suits)Random.Shared.Next(0, 4)));

        selectedCard = cards[cards.Count - 1].ToString();
    }
}
```

Bind your list box to a new field callde selectedCard.

We renamed the variable in the foreach loop to card.

Add two rows to the page. The first row is an <h4> that displays the selected card, just like you did in Chapter 2 with the dropdown.

This row adds a button that calls a new method called AddCard.

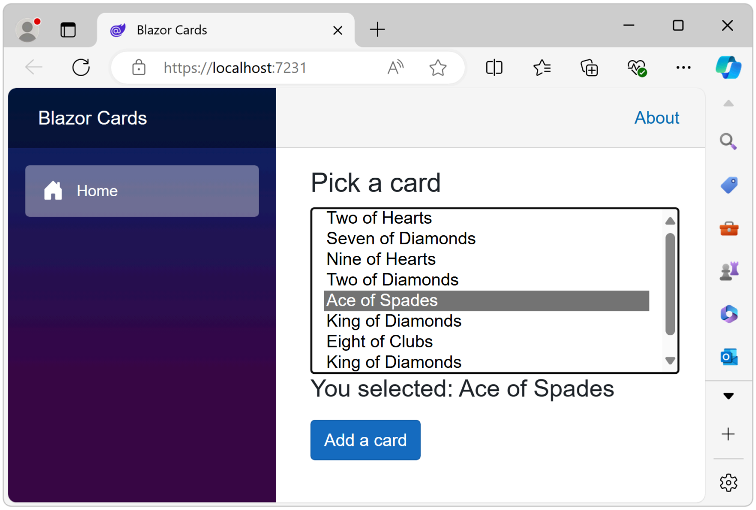
Here are the fields for binding: the selected card, and the list of cards (which starts out empty).

The AddCard method adds a new random card to the list, then sets the selectedCard field to the string value of the card that was just added to the list.

Does your list box show "BlazorCards.Card" instead of card names? Make sure your Card class overrides the ToString method so it returns Name.

3 Run your app and add a bunch of cards.

Every time you click the button, the app adds a card to the list. But hold on—***something's wrong***.



Blazor Cards

https://localhost:7231

Blazor Cards

About

Home

Pick a card

Two of Hearts
Seven of Diamonds
Nine of Hearts
Two of Diamonds
Ace of Spades
King of Diamonds
Eight of Clubs
King of Diamonds

You selected: Ace of Spades

Add a card

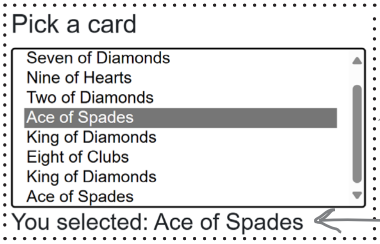
We clicked the Add a card button, but instead of adding a card and scrolling to the bottom of the list our app selected a card in the middle of the list.

Most of the time, clicking the card causes the list box to scroll all the way to the bottom. But sometimes it ***jumps to the middle of the list***. What's going on?

The Case of the Card that Jumped

You've been sleuthing out bugs throughout the book—and as Sherlock Holmes once said, "You know my methods, Watson." Luckily, this bug won't be hard to track down.

Start your app and keep clicking the *Add a card* button until the selection jumps to the middle of the list. In our screenshot above, it selected the Ace of Spades. Now use the scroll bar to scroll to the bottom of the list box:



Pick a card

Seven of Diamonds
Nine of Hearts
Two of Diamonds
Ace of Spades
King of Diamonds
Eight of Clubs
King of Diamonds
Ace of Spades

You selected: Ace of Spades

Here's the card that got selected when we pressed the button.

Here's the card that just got added to the end of the list. It's the same card!

Aha! The card that got selected in the middle of the list is the **same card that was just added**. When the AddCard method updated the selectedCard field, the list box found the first card that matched the updated value. Since there was already an Ace of Spades in the list, ***it selected that card instead of the one added to the end***.

We have a culprit! But how will we get the list box to select a *specific* Ace of Spades, not just any Ace of Spades?

Give options unique values so the list box can select a specific card

It's not unusual for a list box to display two items that appear the same, but when that happens your app needs to figure out which item it should select. To help with that problem, you can use the <option> tag's **`value` property** to assign a unique value to every item in the list.

This is the value that the bound variable gets set to.

This is the text that's displayed in the list.

All of the options you've used so far have looked like this: `<option value="Lion">Lion</option>` – the `value` property has always been the same as the content of the tag.

You'll **fix the bug in your app** by setting the `value` property to something unique to that specific item in the list. Since this is a list, you can use the index of the card in the List<Card>. We'll introduce some **new Razor markup** to help you make the change: **`@for`** lets you include a for loop in your HTML, and **`@if`** lets you add an if/else conditional.

First, **change the @foreach** that creates the options ***to a @for loop***:

```
<select class="col-12" size="8" @bind="selectedCard">
     @for(int i = 0; i < cards.Count; i++)
     {
         <option value="@i">@cards[i]</option>
     }
</select>
```

A @for loop in Razor markup works just like a C# for loop. Just like with @foreach, any HTML inside the loop is repeated for each iteration of the loop.

Instead of setting the selectedCard field to the name of the card, the list box will set it to the index of the card in the list. There's a problem, though—the selectedCard field is a string? type. **Change it to an int** instead of a string?:

```
int selectedCard;
```

Now we've got another problem. The <h4> row displays the value of the selectedCard field, but now that value is a number and not the name of a card. Try replacing `@selectedCard` with `@cards[selectedCard]` – when you run your app, you'll get an exception because the @cards list is empty.

Replace your <h4> row with this @if that makes sure selectedCard won't throw an exception:

```
@if (selectedCard >= cards.Count)
{
    <h4 class="row">No card selected</h4>
}
else
{
    <h4 class="row">You selected: @cards[selectedCard]</h4>
}
```

An @if directive lets you do a conditional test and add HTML to the page based on the results of the test. You can include an else, just like with a C# if/else.

Finally, update your AddCard method to **set selectedCard to the index of the card** that was just added:

```
private void AddCard()
{
    cards.Add(new Card((Values)Random.Shared.Next(1, 14),
                       (Suits)Random.Shared.Next(0, 4)));

    selectedCard = cards.Count - 1;
}
```

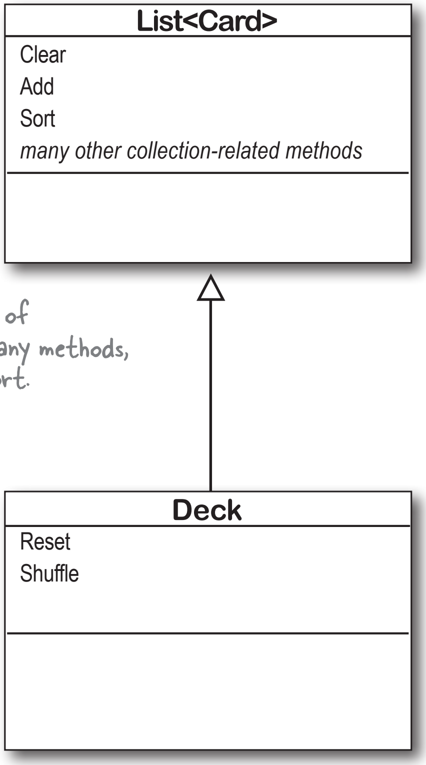
Run your app again. Click the button many times—now it always scrolls to the card that was just added.

Exercise

Create a Deck class that extends List<Card>.

You learned all about inheritance in Chapter 6. Now it's time to apply that knowledge to create a class that represents a deck of cards.

Add a Deck class to your project that extends List<Card> so it inherits all of the collection-related methods, including the Clear and Add methods.



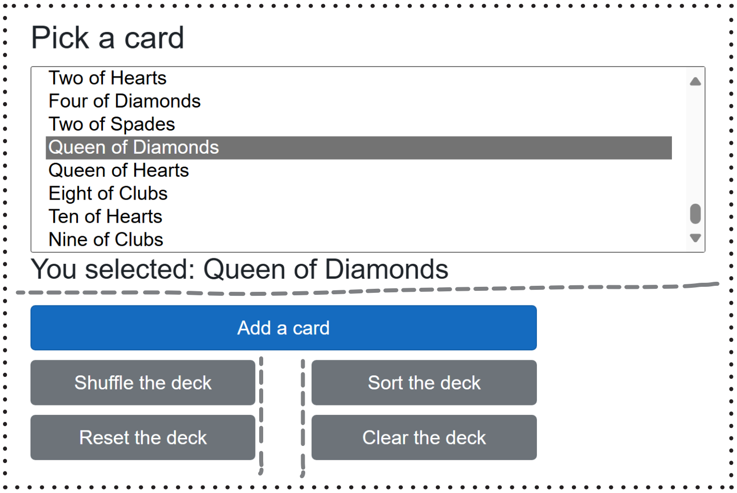
```
class Deck : List<Card>
{
  /// <summary>
  /// The constructor resets the 52-card deck
  /// </summary>
  public Deck() {
     Reset();
  }

  /// <summary>
  /// Clears the deck, then loops through suits and
  /// values, and adds each card to the 52-card deck
  /// </summary>
  public void Reset() { ... }

  /// <summary>
  /// Creates a copy of the deck, clears the deck, and
  /// uses a while loop to move a random card from
  /// the copy to the deck and remove it from the copy
  /// </summary>
  public void Shuffle() { ... }
}
```

The Deck class is a subclass of List<Card> so it inherits many methods, including Clear, Add, and Sort.

Write the code for the Reset and Shuffle methods. Carefully read the XMLDoc to see what the methods need to do.

Modify your *Home.razor* to use the deck add buttons to the bottom of the page that call its methods

Here's what how the buttons are laid out. Can you create the HTML so your app matches our screenshot?



Pick a card

Two of Hearts
Four of Diamonds
Two of Spades
Queen of Diamonds
Queen of Hearts
Eight of Clubs
Ten of Hearts
Nine of Clubs

You selected: Queen of Diamonds

Surround your current <button> tag with a <div> that adds a row with a top margin:

Change the button's class property so it contains "col-9" to make it span 9 of the 12 columns in the Bootstrap grid layout.

The next two buttons are also contained in a <div class="row mt-2">. Each of those buttons spans 4 columns, and there's a col-1 between them to add space:

```
<div class="col-1" />
```

Only the top button is a primary button. Make the rest of the buttons secondary.

It's not cheating to peek at the solution!

Exercise Solution

Here's the full Home.razor for your Blazor Cards page with buttons to shuffle, sort, reset, and clear the deck:

```
@page "/"
@rendermode InteractiveServer
@using BlazorCards
<PageTitle>Blazor Cards</PageTitle>

<div class="container">
    <h3 class="row">Pick a card</h3>
    <div class="row">
        <select class="col-12" size="8" @bind="selectedCard">
            @for (int i = 0; i < cards.Count; i++)
            {
                <option value="@i">@cards[i]</option>
            }
        </select>
    </div>

    @if (selectedCard >= cards.Count)
    {
        <h4 class="row">No card selected</h4>
    }
    else
    {
        <h4 class="row">You selected: @cards[selectedCard]</h4>
    }

    <div class="row mt-2">
        <button type="button" class="col-9 mt-2 btn btn-primary"
                @onclick="AddCard">
            Add a card
        </button>
    </div>

    <div class="row mt-2">
        <button type="button" class="col-4 btn btn-secondary"
                @onclick="ShuffleDeck">
            Shuffle the deck
        </button>
        <div class="col-1" />
        <button type="button" class="col-4 btn btn-secondary"
                @onclick="SortDeck">
            Sort the deck
        </button>
    </div>

    <div class="row mt-2">
        <button type="button" class="col-4 btn btn-secondary"
                @onclick="ResetDeck">
            Reset the deck
        </button>
        <div class="col-1" />
        <button type="button" class="col-4 btn btn-secondary"
                @onclick="ClearDeck">
            Clear the deck
        </button>
    </div>
</div>
```

The card classes and enums are in the BlazorCards namespace.

Each option displays the name of the card, but the actual value that gets bound to selectedCard is @i, or the index of the card in the List.

This @if checks that the selected card is a valid index in the List, so it never tries to get the value of cards[selectedCard] if it will throw an "out of range" exception.

This row has a button that spans 9 of the 12 Bootstrap columns. We made it the primary button.

The other two rows have two buttons that span 4 of the 12 Bootstrap columns, with a 1-column margin between them so they add up to a total of 9 columns. That will make them the same width as the "Add a card" button.

Exercise Solution

```
@code {
    int selectedCard;
    private Deck cards = new Deck();

    private void AddCard()
    {
        cards.Add(new Card((Values)Random.Shared.Next(1, 14),
                           (Suits)Random.Shared.Next(0, 4)));

        selectedCard = cards.Count - 1;
    }

    private void ShuffleDeck() { cards.Shuffle(); }
    private void SortDeck() { cards.Sort(new CardComparerByValue()); }
    private void ResetDeck() { cards.Reset(); }
    private void ClearDeck() { cards.Clear();  }
}
```

The cards field is now a Deck object, which extends List<Card>.

The event handler methods for the four buttons call the corresponding methods on the Deck class.

Here's the Deck class that extends List<Card> and adds Reset and Shuffle methods:

```
namespace BlazorCards;

class Deck : List<Card>
{
    public Deck()
    {
        Reset();
    }

    public void Reset()
    {
        Clear();
        for (int suit = 0; suit <= 3; suit++)
            for (int value = 1; value <= 13; value++)
                Add(new Card((Values)value, (Suits)suit));
    }

    public void Shuffle()
    {
        List<Card> copy = new List<Card>(this);
        Clear();
        while (copy.Count > 0)
        {
            int index = Random.Shared.Next(copy.Count);
            Card card = copy[index];
            copy.RemoveAt(index);
            Add(card);
        }
    }
}
```

This nested for loop goes through each of the suits, and for each suit it loops through all of the cards and adds them to the Deck.

This while loop picks a random card from the copy, adds it to the Deck, and then removes it from the copy, repeating until the copy is empty.

In the next exercise, you'll take the ideas and tools you just used and apply them to a new project. This is a great way to get them to stick in your brain.

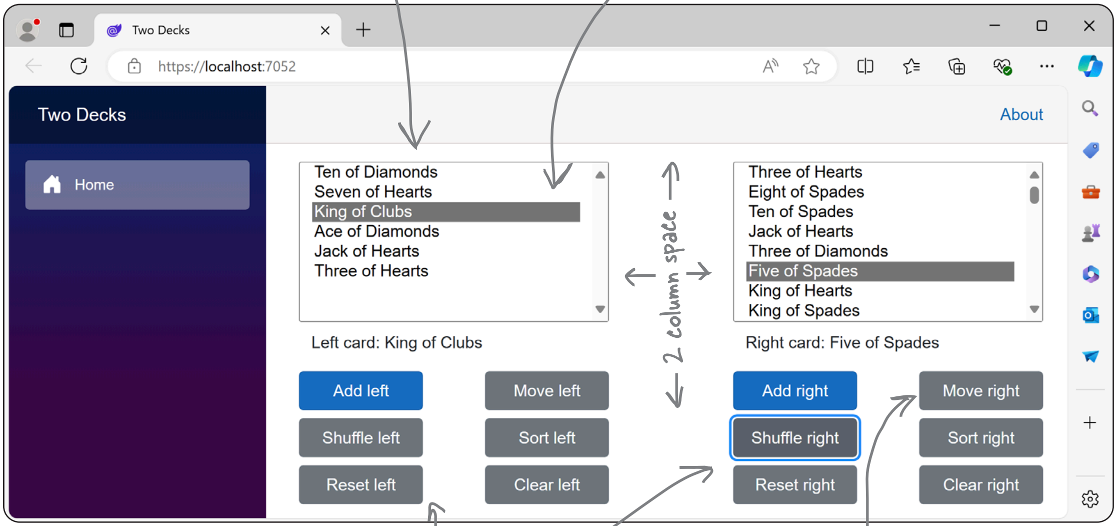
Exercise

You've just used a list box control, options with unique values, @for, and @if, and you used inheritance to create a Deck class that extends List<Card>, and you've learned more about laying out pages using the 12-clolumn Bootstrap grid system. In this exercise, you'll use all of those things to create an app that has two decks of cards, with buttons that let you shuffle, sort, reset, and clear the deck, and two more buttons to move cards from one deck to the other.

You can do this! Just take it step by step, and remember that **it's not cheating** to peek at the solution.

We cleared the left deck, then used the "Add left" button to add six cards to it.

Click a card in the left deck to select it, then click the Move Right button to remove it from this deck and add it to the other one.



Ten of Diamonds
Seven of Hearts
King of Clubs
Ace of Diamonds
Jack of Hearts
Three of Hearts

Left card: King of Clubs

Three of Hearts
Eight of Spades
Ten of Spades
Jack of Hearts
Three of Diamonds
Five of Spades
King of Hearts
King of Spades

Right card: Five of Spades

Now there are two sets of buttons to add, shuffle, sort, reset, and clear the decks.

The "Move right" button removes the selected card from the right deck and adds it to the left deck. The "Move left" button removes the selected card from the left deck and adds it to the right deck.

Step 1: Copy the BlazorCards code and card classes, then replace the cards field with two Deck fields

Create a **new project called TwoDecksBlazor**. Modify *NavMenu.razor* to update the app name and remove the extra menu items. Add the **Deck, Card, and CardComparerByValue classes and the Suits and Values enums** into your project and **add a @using directive** to your *Home.razor* file because they're in the BlazorCards namespace.

Add a @code section to and add two Deck fields called leftDeck and RightDeck:

```
Deck leftDeck = new Deck();
Deck rightDeck = new Deck();
```

You'll also need two fields to store the indexes of the selected left and right cards:

```
int selectedCardLeft;
int selectedCardRight;
```

Getting argument out of range exceptions you're having trouble tracking down? Make sure you're not using selectedCardLeft with rightDeck or vice versa.

Exercise

Step 2. Add click event handler methods that work just like the ones in your Blazor Cards project

Your app has familiar buttons to add, shuffle, sort, reset, and clear the right decks. These buttons work just like the ones in your Blazor Cards project. Add methods named AddCardLeft, ShuffleLeftDeck, SortLeftDeck ResetLeftDeck, and ResetLeftDeck that work just like the ones in Blazor Cards, except they use the leftDeck field. Then add AddCardRight, ShuffleRightDeck, SortRightDeck ResetRightDeck, and ResetRightDeck methods for the right deck.

Step 3. Create the MoveLeftToRight and MoveRightToLeft methods

Add a MoveLeftToRight event handler method for the “Move left” button that moves a card from the left deck to the right deck. It adds leftDeck[selectedCardLeft] to the right deck, sets selectedCardRight to rightDeck.Count - 1, and then calls `leftDeck.RemoveAt(selectedCardLeft)` – that’s a method inherited from List<Card> that removes an element at a specific index. “Move right” button that moves a card from the right deck to the left deck.

Then add the MoveRightToLeft method, which does exactly the same thing except left and right are reversed.

Step 4. Add LeftCardNotSelected and RightCardNotSelected properties

Create two boolean properties called LeftCardNotSelected and RightCardNotSelected that do the same check as the @if directive in the BlazorCards project. The LeftCardNotSelected property has a getter that returns `selectedCardLeft >= leftDeck.Count`. RightCardNotSelected does the same thing, except for the right deck.

Step 5. Lay out the HTML for the page

The page has **five rows**, each with a top margin of 2 (`<div class="row mt-2">`) except for the top row, whcih doesn’t have `mt-2` in the `class` property. Here’s what you’ll add to each row:

- The first row has a list box for the left deck, a two-column spacer (`<div class="col-2" />`), and a list box for the right deck. Both decks are 5 columns wide. They work just the list boxes in the BlazorCards project, except the left list box @for loop reads the cards from the leftDeck field, and the right list box reads from the rightDeck field.
- The second row has an @if for the text that displays the card selected in the left deck, a two-column spacer, and an @if for the text that displays the card selected in the right deck. The @if directives call the properties to check if a card is not selected (`@if (LeftCardNotSelected)`). They work just like the similar @if in BlazorCards, except instead of adding `<h4 class="row">` they add `<div class="col-5">` to the page.
- The third row has the Add left, Move left, Add right, and Move right buttons, in that order. The buttons all are 2 columns wide. The Add buttons are primary (`class="col-2 btn btn-primary"`), all of the other buttons are secondary, including the buttons in the other rows (`class="col-2 btn btn-primary"`). There's a 1-column spacer (`<div class="col-1" />`) between each pair of Add and Move buttons, and a 2-column spacer between the buttons on the left and and the ones on the right. The Add buttons call the AddCardLeft and AddCardRight Mehtods, and the Move buttons call the MoveLeftToRight and MoveRightToLeft methods. Add the property `disabled="@LeftCardNotSelected"` to the Move left button to disable it if a card is not selected. Add the property `disabled="@RightCardNotSelected"` to the Move right card too.
- The fourth row has the Shuffle left, Sort left, Shuffle right, and Sort right buttons. They are all 2-column wide secondary buttons, with the same spacers as in the third row. They call the ShuffleLeftDeck, SortLeftDeck, ShuffleRightDeck, and SortRightDeck methods.
- The fourth row has the Reset left, Clear left, Reset right, and Clear right buttons. They are all 2-column wide secondary buttons, with the same spacers as in the third row. They call the ResetLeftDeck, ClearLeftDeck, ResetRightDeck, and ClearRightDeck methods.

If there are more than 12 columns in a row Bootstrap will wrap to the next row, so if you see buttons in the wrong place, you may just be missing a `<div>` and `<div class="row mt-2">` between two buttons.

Exercise Solution

Here's the full *Home.razor* file for the Two Decks app.

```
@page "/"
@rendermode InteractiveServer
@using BlazorCards

<PageTitle>Two Decks</PageTitle>

<div class="container">
    <div class="row">
        <select class="col-5" size="8" @bind="selectedCardLeft">
            @for (int i = 0; i < leftDeck.Count; i++)
            {
                <option value="@i">@leftDeck[i]</option>
            }
        </select>

        <div class="col-2"/>

        <select class="col-5" size="8" @bind="selectedCardRight">
            @for (int i = 0; i < rightDeck.Count; i++)
            {
                <option value="@i">@rightDeck[i]</option>
            }
        </select>
    </div>
```

The list boxes for the left and right decks work just like the ones in your BlazorCards app. The only difference is that they read cards from the leftDeck and rightDeck fields.

```
    <div class="row mt-2">
        @if (LeftCardNotSelected)
        {
            <div class="col-5">No left card selected</div>
        }
        else
        {
            <div class="col-5">Left card: @leftDeck[selectedCardLeft]</div>
        }
        <div class="col-2" />
        @if (RightCardNotSelected)
        {
            <div class="col-5">No right card selected</div>
        }
        else
        {
            <div class="col-5">Right card: @rightDeck[selectedCardRight]</div>
        }
    </div>
```

The second row has the two @if directives for the text that display the selected card in each deck. They use the properties that return true if a card is selected.

```
    <div class="row mt-2">
        <button type="button" class="col-2 btn btn-primary"
                @onclick="AddCardLeft">
            Add left
        </button>
        <div class="col-1" />
        <button type="button" class="col-2 btn btn-secondary"
                @onclick="MoveLeftToRight" disabled="@LeftCardNotSelected">
            Move left
        </button>

        <div class="col-2" />
```

We gave you this property to disable the button if there's no left card selected.

```
        <button type="button" class="col-2 btn btn-primary"
                @onclick="AddCardRight">
            Add right
        </button>
        <div class="col-1" />
        <button type="button" class="col-2 btn btn-secondary"
                @onclick="MoveRightToLeft" disabled="@RightCardNotSelected">
            Move right
        </button>
    </div>

    <div class="row mt-2">
        <button type="button" class="col-2 btn btn-secondary"
                @onclick="ShuffleLeftDeck">
            Shuffle left
        </button>
        <div class="col-1" />
        <button type="button" class="col-2 btn btn-secondary"
                @onclick="SortLeftDeck">
            Sort left
        </button>

        <div class="col-2" />

        <button type="button" class="col-2 btn btn-secondary"
                @onclick="ShuffleRightDeck">
            Shuffle right
        </button>
        <div class="col-1" />
        <button type="button" class="col-2 btn btn-secondary"
                @onclick="SortRightDeck">
            Sort right
        </button>
    </div>

    <div class="row mt-2">
        <button type="button" class="col-2 btn btn-secondary"
                @onclick="ResetLeftDeck">
            Reset left
        </button>
        <div class="col-1" />
        <button type="button" class="col-2 btn btn-secondary"
                @onclick="ClearLeftDeck">
            Clear left
        </button>

        <div class="col-2" />

        <button type="button" class="col-2 btn btn-secondary"
                @onclick="ResetRightDeck">
            Reset right
        </button>
        <div class="col-1" />
        <button type="button" class="col-2 btn btn-secondary"
                @onclick="ClearRightDeck">
            Clear right
        </button>
    </div>

</div>
```

We gave you this property to disable the button if there's no right card selected.

These buttons all work just like the ones in the BlazorCards app. The only difference is that they all have class properties that make them 2 columns wide and secondary buttons, and they use call the methods for the left and right decks.

Exercise Solution

```
@code {
    int selectedCardLeft;
    int selectedCardRight;

    Deck leftDeck = new Deck();
    Deck rightDeck = new Deck();

    private void AddCardLeft()
    {
        leftDeck.Add(new Card((Values)Random.Shared.Next(1, 14),
                              (Suits)Random.Shared.Next(0, 4)));

        selectedCardLeft = leftDeck.Count - 1;
    }
    private void ShuffleLeftDeck() { leftDeck.Shuffle(); }
    private void SortLeftDeck() { leftDeck.Sort(new CardComparerByValue()); }
    private void ResetLeftDeck() { leftDeck.Reset(); }
    private void ClearLeftDeck() { leftDeck.Clear(); }

    private void AddCardRight()
    {
        rightDeck.Add(new Card((Values)Random.Shared.Next(1, 14),
                              (Suits)Random.Shared.Next(0, 4)));

        selectedCardRight = rightDeck.Count - 1;
    }
    private void ShuffleRightDeck() { rightDeck.Shuffle(); }
    private void SortRightDeck() { rightDeck.Sort(new CardComparerByValue()); }
    private void ResetRightDeck() { rightDeck.Reset(); }
    private void ClearRightDeck() { rightDeck.Clear(); }

    void MoveLeftToRight()
    {
        rightDeck.Add(leftDeck[selectedCardLeft]);
        selectedCardRight = rightDeck.Count - 1;
        leftDeck.RemoveAt(selectedCardLeft);
    }

    void MoveRightToLeft() {
        leftDeck.Add(rightDeck[selectedCardRight]);
        selectedCardLeft = leftDeck.Count - 1;
        rightDeck.RemoveAt(selectedCardRight);
    }

    public bool LeftCardNotSelected {
                get { return selectedCardLeft >= leftDeck.Count; }
    }

    public bool RightCardNotSelected {
                get { return selectedCardRight >= rightDeck.Count; }
    }
}
```

There are two sets of fields to track the deck and selected cards, one set for the left deck and one set for the right deck.

These event handler methods work just like the ones in the BlazorCards app, except they use the leftDeck and rightDeck fields.

The event handler methods for the buttons to move cards from one deck to the other first add the card being moved, then update the field that tracks the selected card to make sure the moved card is selected, and finally remove the card from the old deck.

These properties return true of a valid card is selected in the deck.

If MoveLeftToRight is called when there's no card selected, it will throw an exception. We could added an if statement that checks LeftCardNotSelected and only moves the card if it is, but instead we used the disabled property to disable the button. Did we make the right choice?

There's no right or wrong answer to this question, because there are lots of ways to write code to do the same thing.

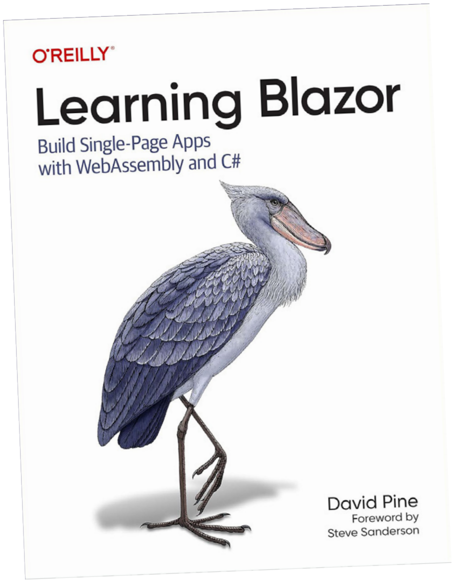
You've got the tools you need to build some really great web applications with Blazor.

The last four chapters in the book don't have .NET MAUI projects, so...

Congratulations! You're done with the *Blazor Learner's Guide*. You can go back to the book and resume with Unity Lab 8 (or if you're skipping the Unity Lab projects, Chapter 9).

But you don't have to stop your Blazor journey here. You've laid down a great foundation. If you want to learn more, we love *Learning Blazor*, a fantastic book by Blazor expert David Pine that gives you a great in-depth guide to building web powerful applications using Blazor.

We love this book! If you want to learn more about Blazor, this is where to go next.


O'REILLY
Learning Blazor
Build Single-Page Apps with WebAssembly and C#
David Pine
Foreword by Steve Sanderson