O'REILLY®

~~Fourth~~ Fifth Edition

Head First C#

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

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

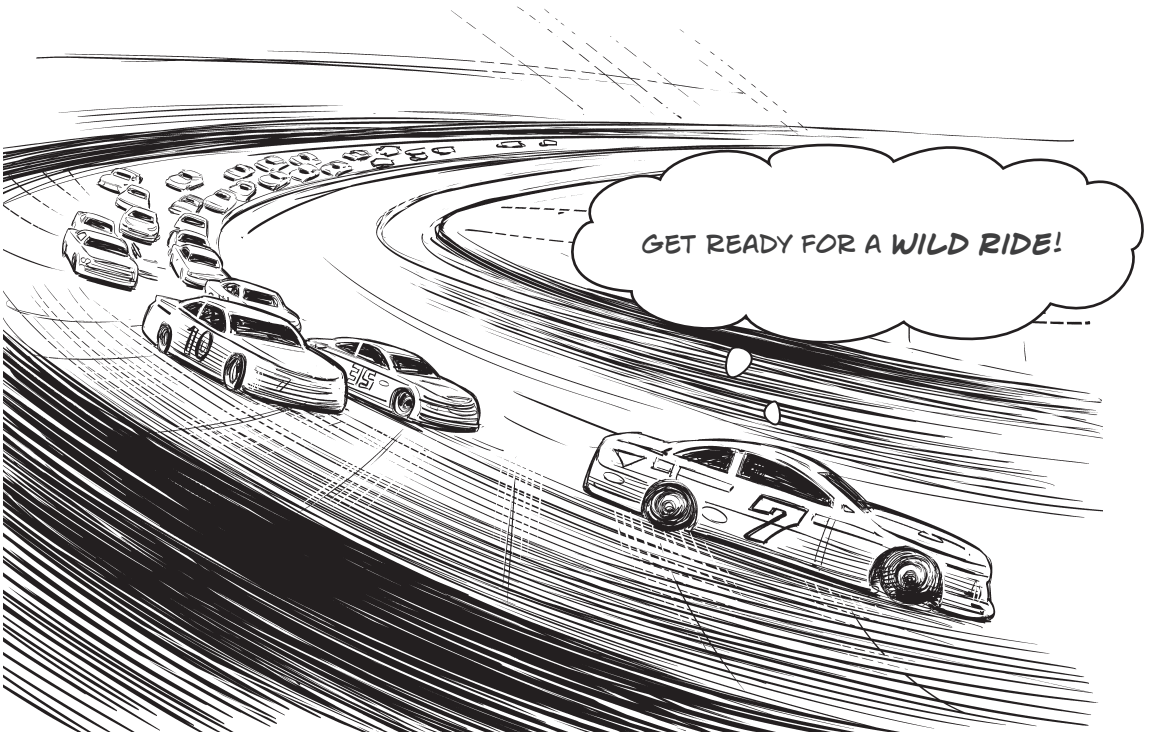
Andrew Stellman
& Jennifer Greene

The Animal Matching Game project is the first .NET MAUI project in the book. The entire first chapter is dedicated to using this project to introduce the user to .NET MAUI and the basics of working with C# projects in Visual Studio.

A Brain-Friendly Guide

1 start building apps with C#

Build something great...fast!



Want to build great apps…right now?

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

Why you should learn C#

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

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

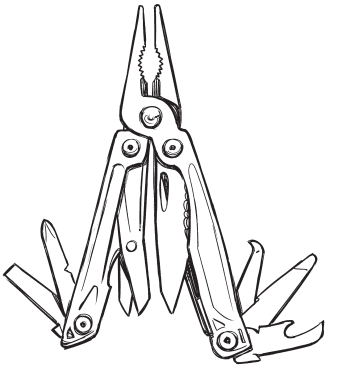
- **C# is really popular… and for good reason.** Microsoft released the first version of C# over 20 years ago, and the C# team been working on it since then—and it shows! They've kept it up to date and added lots of new features, which is why it's one of the most flexible and powerful programming languages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

Write code and explore C# with Visual Studio

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

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

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



Visual Studio is your gateway to C#

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

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

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

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

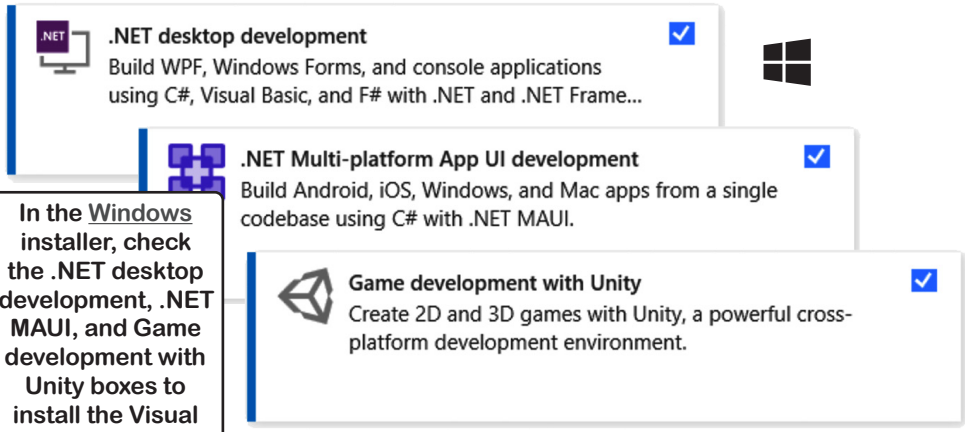
Visual Studio is an powerful development environment, and it's an amazing learning tool to help you explore C#.

IDE Tips

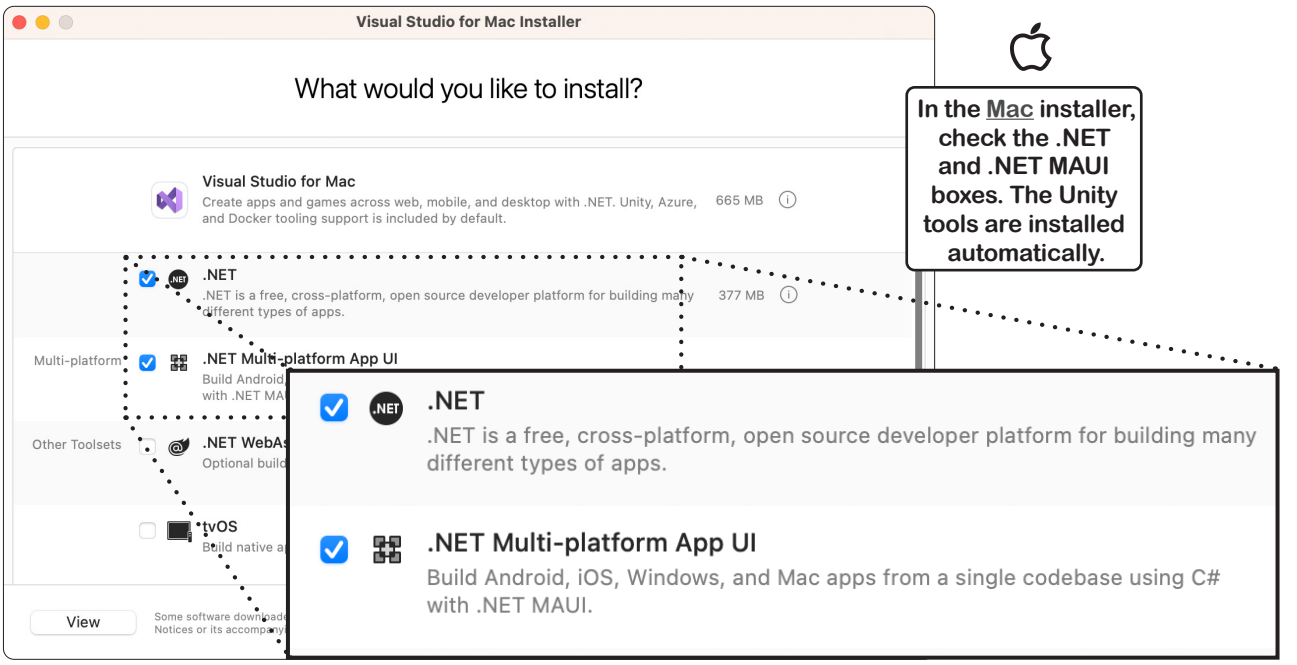
We'll often refer to Visual Studio as **"the IDE"** throughout this book. Keep an eye out for handy IDE tips that help you become a more efficient coder.

Install Visual Studio Community Edition

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



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



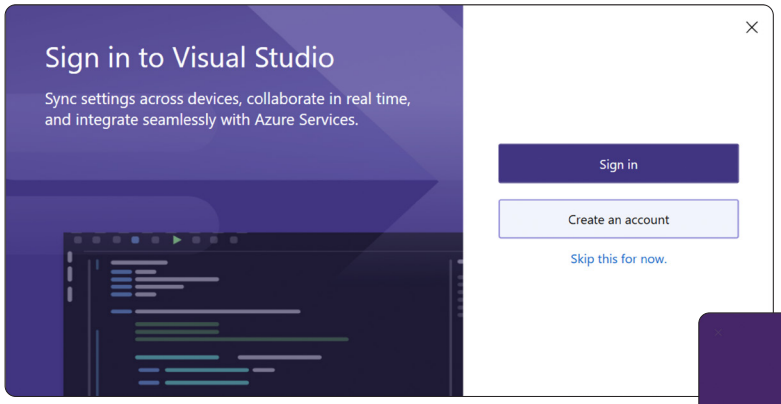
In the Mac installer, check the .NET and .NET MAUI boxes. The Unity tools are installed automatically.

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

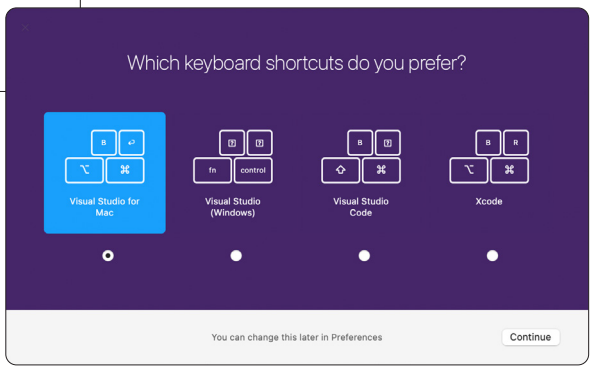
Visual Studio Code is an amazing open source, cross-platform code editor, but it's not tailored to .NET development the way Visual Studio is. That's why we can use Visual Studio throughout this book as a tool for learning and exploration.

Run Visual Studio

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

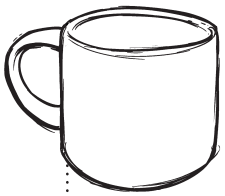


The first time you start Visual Studio, it may ask you to sign into your Microsoft account—make sure you do that, and if you don't have an account, you can create one. This is how Microsoft activates you your free license for the Community edition of Visual Studio.



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

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



Relax

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

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

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

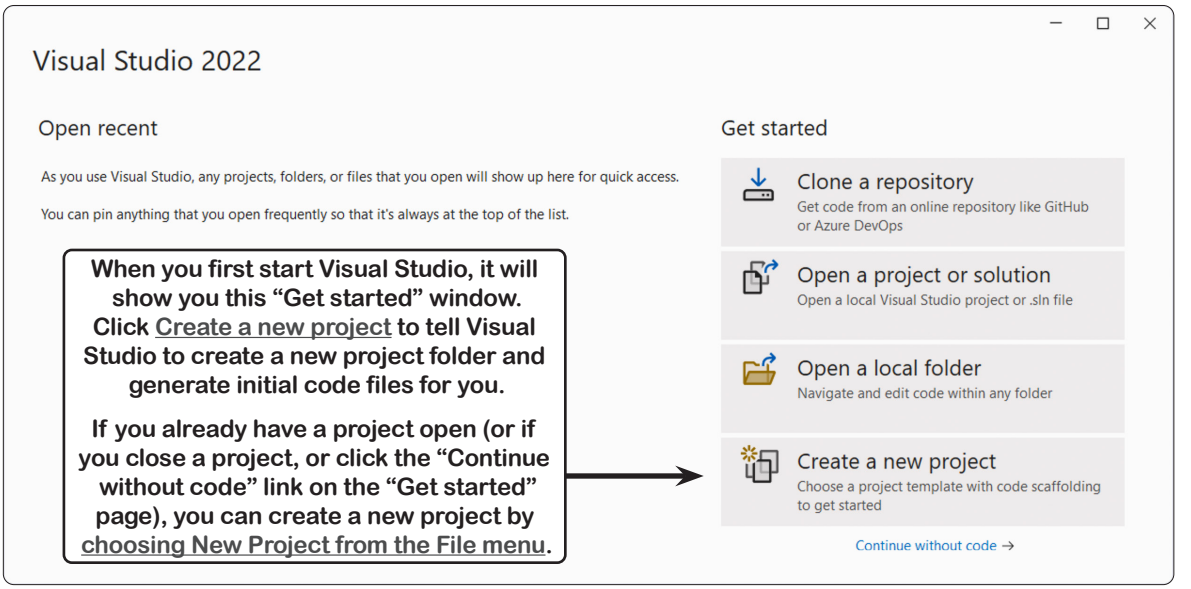
If you run into trouble installing Visual Studio, head to our YouTube channel to see videos of the entire installation process: https://www.youtube.com/@headfirstcsharp

Create your first project in Visual Studio – Windows edition

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

1 Tell Visual Studio to create a new project.

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



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

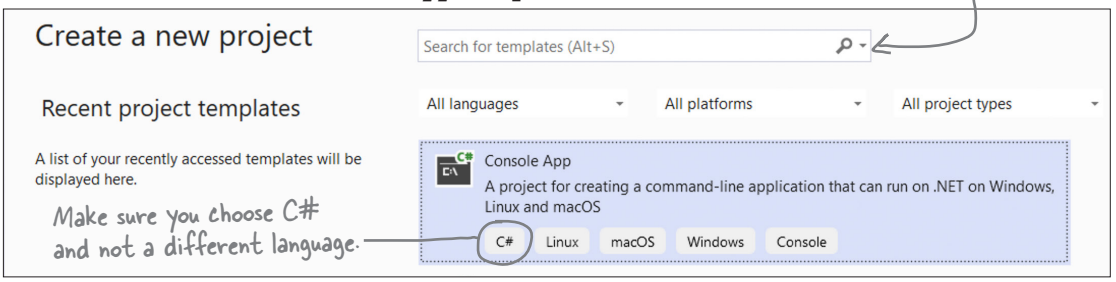
This applies to both Windows and Mac projects! We'll show you how to create and run an app in Visual Studio for Mac, too.

Geek Note

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

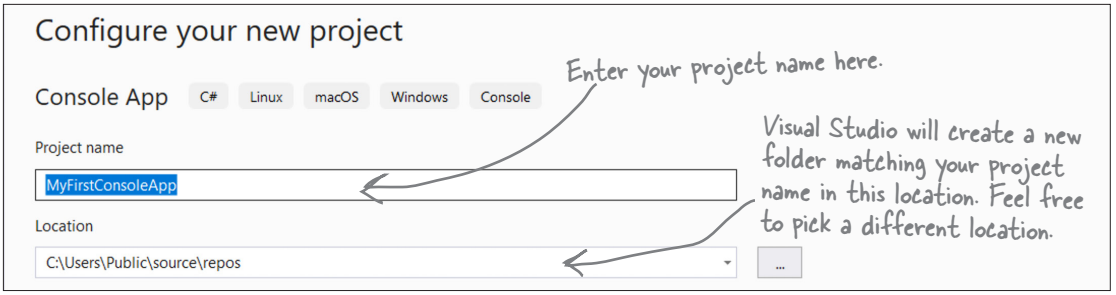
2 Choose a project template for Visual Studio to use.

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



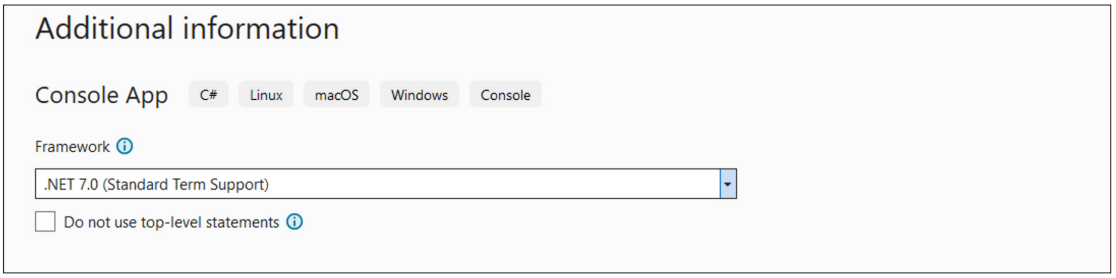
3 Enter a name for your project and click Next.

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

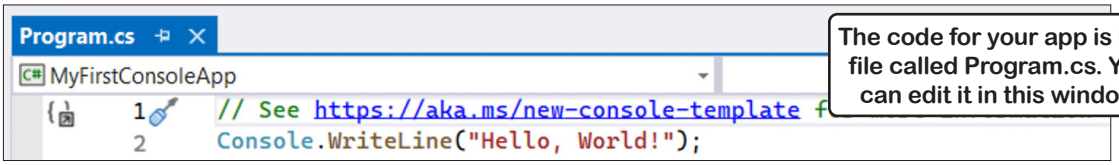


4 Make sure you're using the current version of .NET.

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



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

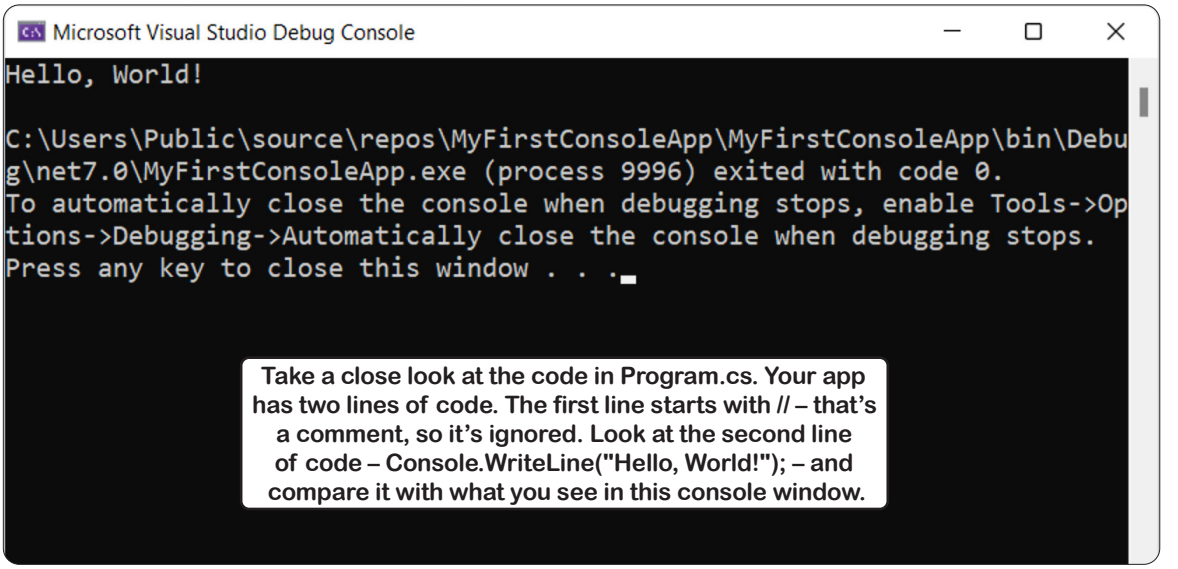


5 Run your app.

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

6 Look at your app's output.

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



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

```
Hello World!
```

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

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

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

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

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

there are no Dumb Questions

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

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

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

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

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

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

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

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

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

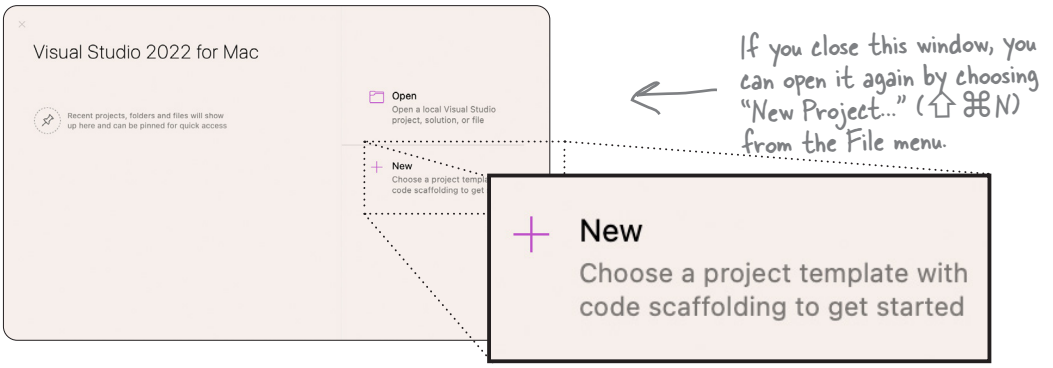
Visual Studio will generate code you can use as a starting point for your applications. Making sure the app does what it's supposed to do is entirely up to you.

Create your first project in Visual Studio – Mac edition

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

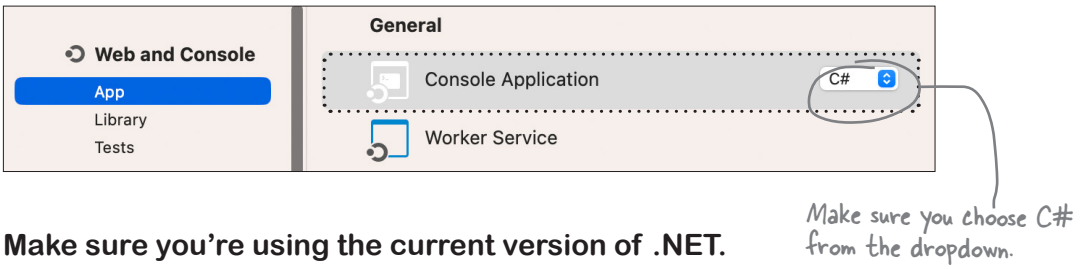
1 Start creating a new project a new project.

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



2 Choose a project template for Visual Studio to use.

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



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

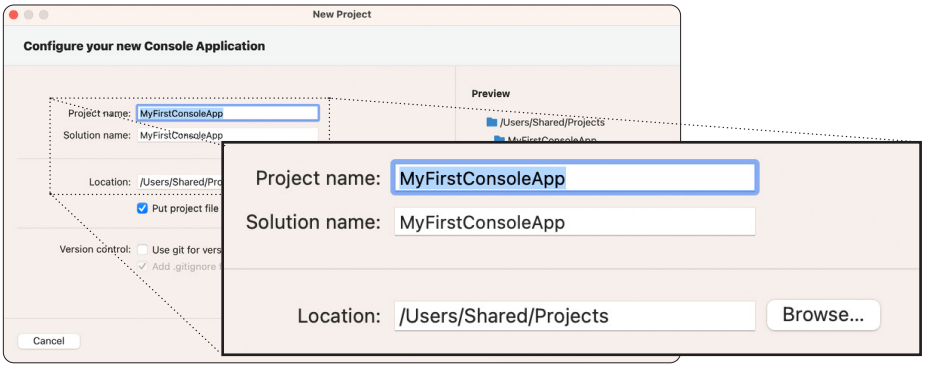
The current version of .NET at the time we're writing this is 7.0 – make sure the version that you're using is 7.0 or later. Click the Continue button.

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

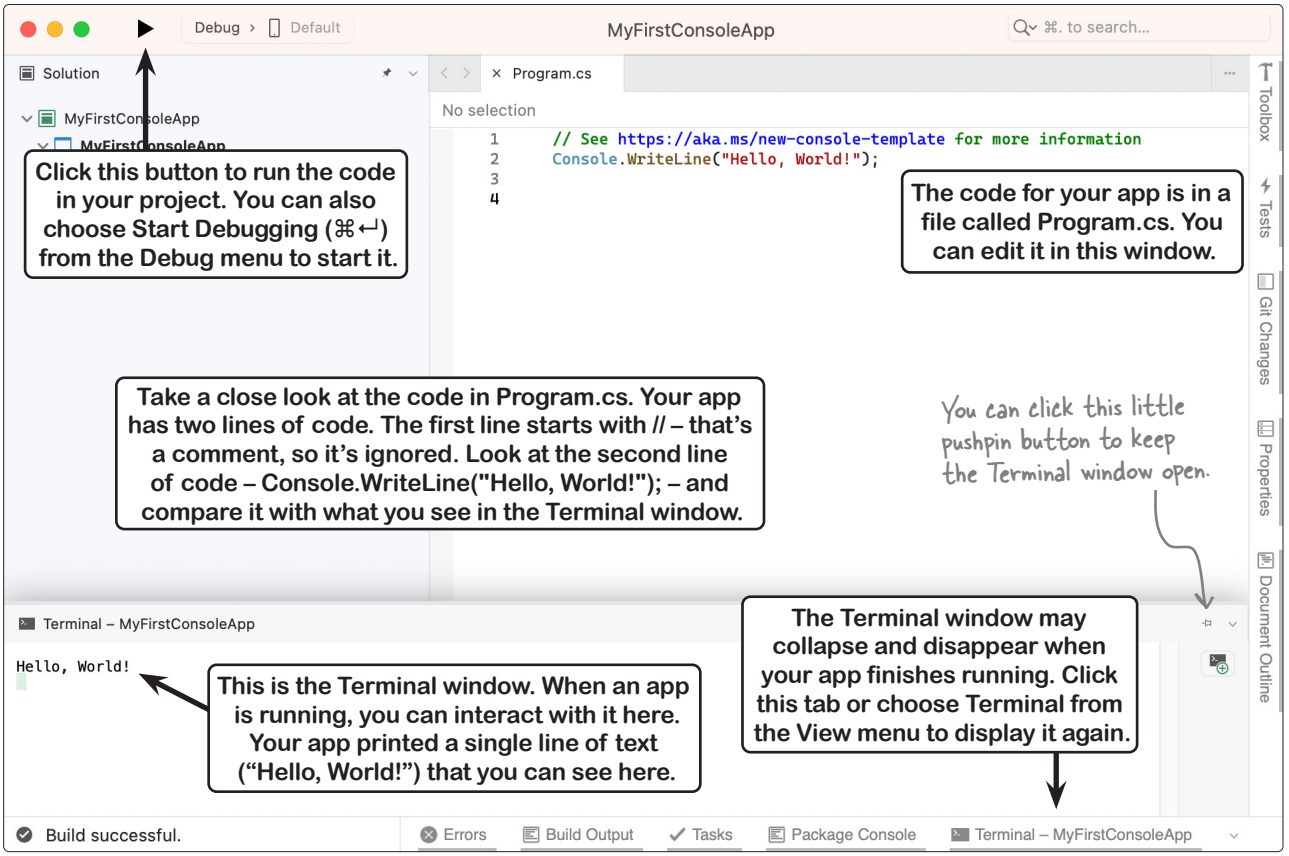
4

Enter a name for your project and click Create.

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

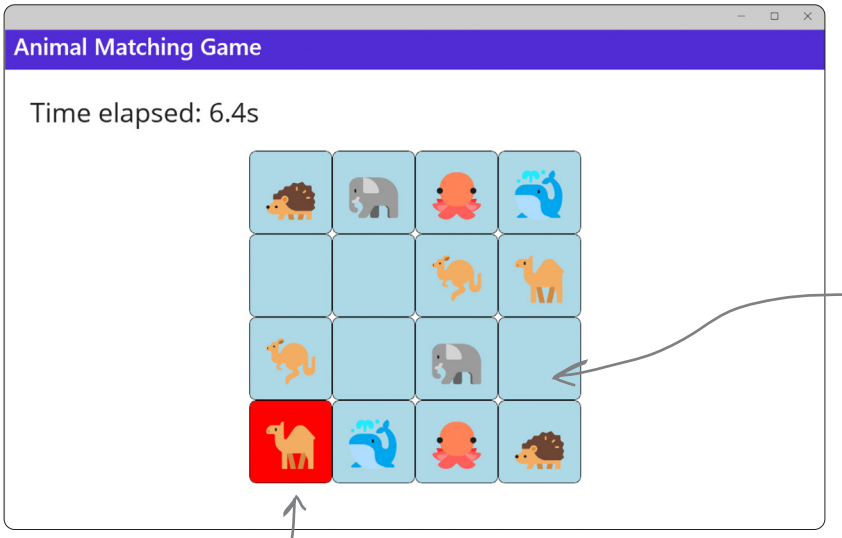


Once Visual Studio creates your project, open a file called Program.cs that has the code for your new app. You can **press the run button ▶ to run the app** and view the results in the Terminal window.



Let's build a game!

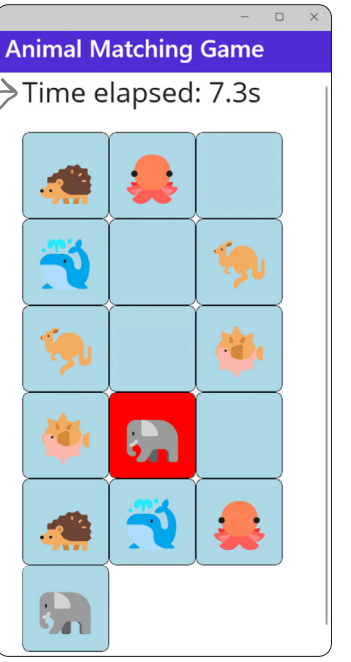
You've built your first C# app, and that's great! Now that you've done that, let's build something a little more complex. We're going to build an **animal matching game**, where a player is shown 8 pairs of animals and needs to click on them in pairs to make them disappear.



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

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

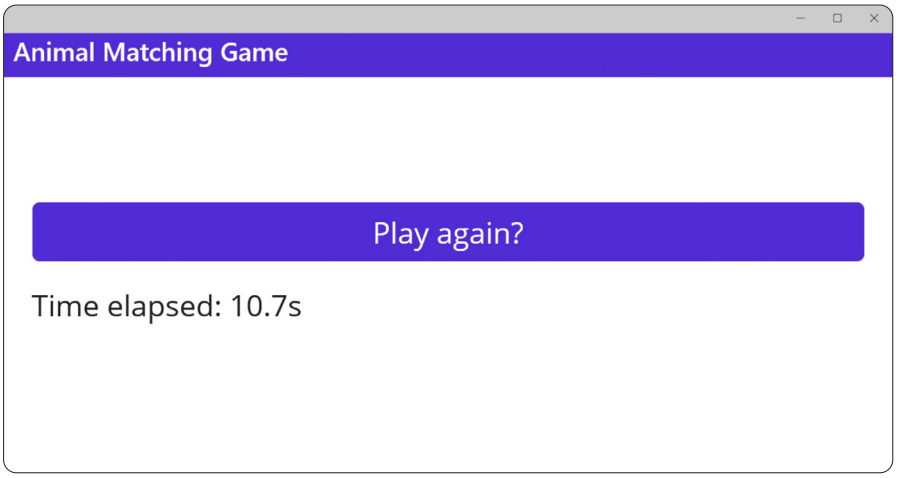
When you click the first button, it changes color. If you click on its match, then both animals disappear. If you click any other animal, the color of the first button changes back and you have to start over with a new pair.



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

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

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



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

Game design... and beyond

What is a game?

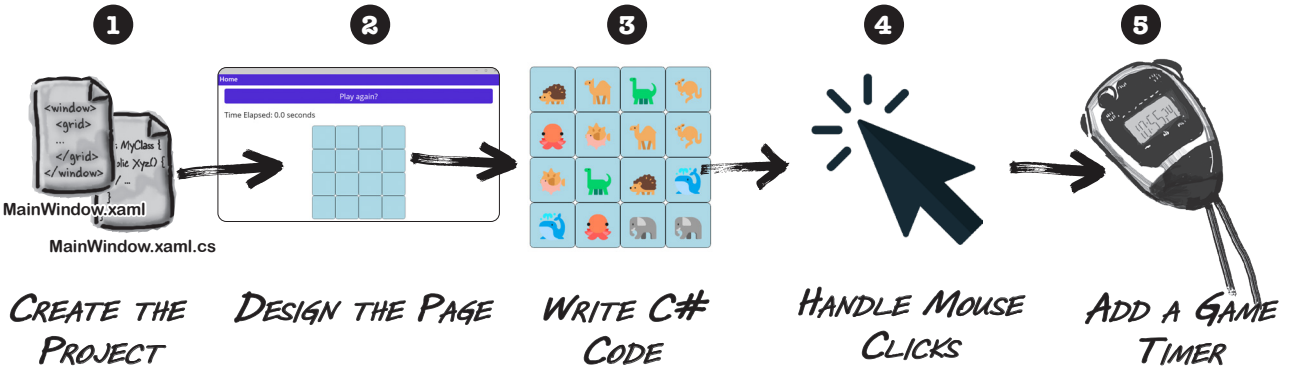
It may seem obvious what a game is. But think about it for a minute—it's not as simple as it seems.

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

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

Break up large projects into smaller parts

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



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

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

Relax

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

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

Here's how you'll build your game

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

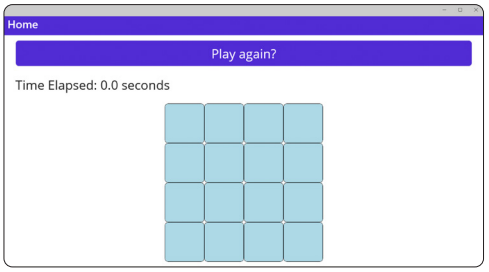
The rest of this chapter will walk you through building your animal matching game, and you'll be doing it in a series of separate parts:

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

MainWindow.xaml

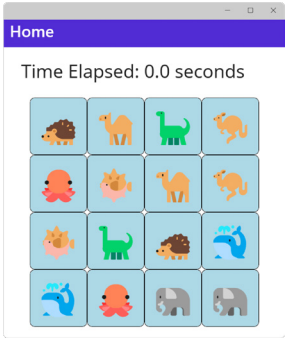
MainWindow.xaml.cs

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



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

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

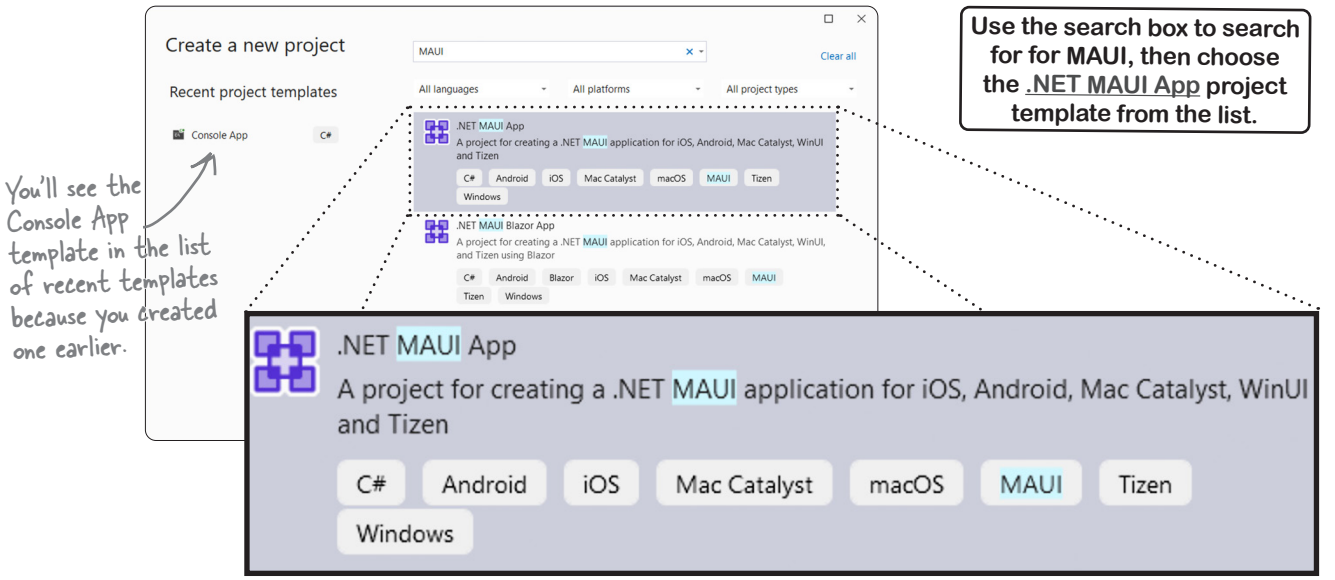


4. **You'll make the gameplay work.**
   The game needs to detect when the user clicks on pairs of emoji, keep track of the pairs, and end the game when they've found all of the matches. You'll write that code too.

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

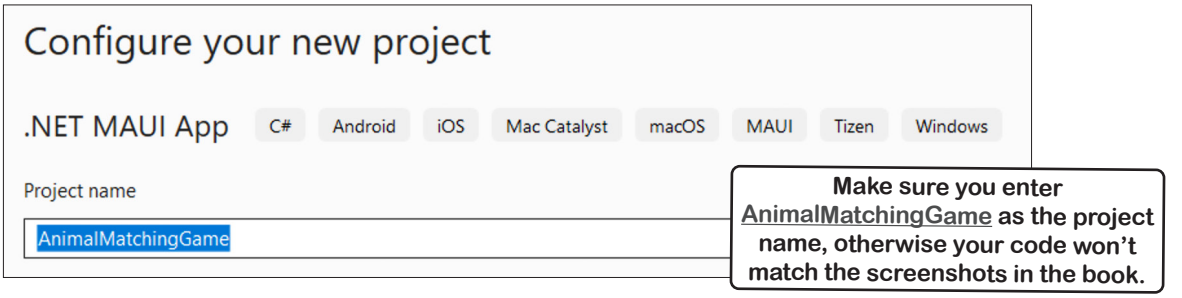
Create a .NET MAUI project in Visual Studio – Windows edition

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



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

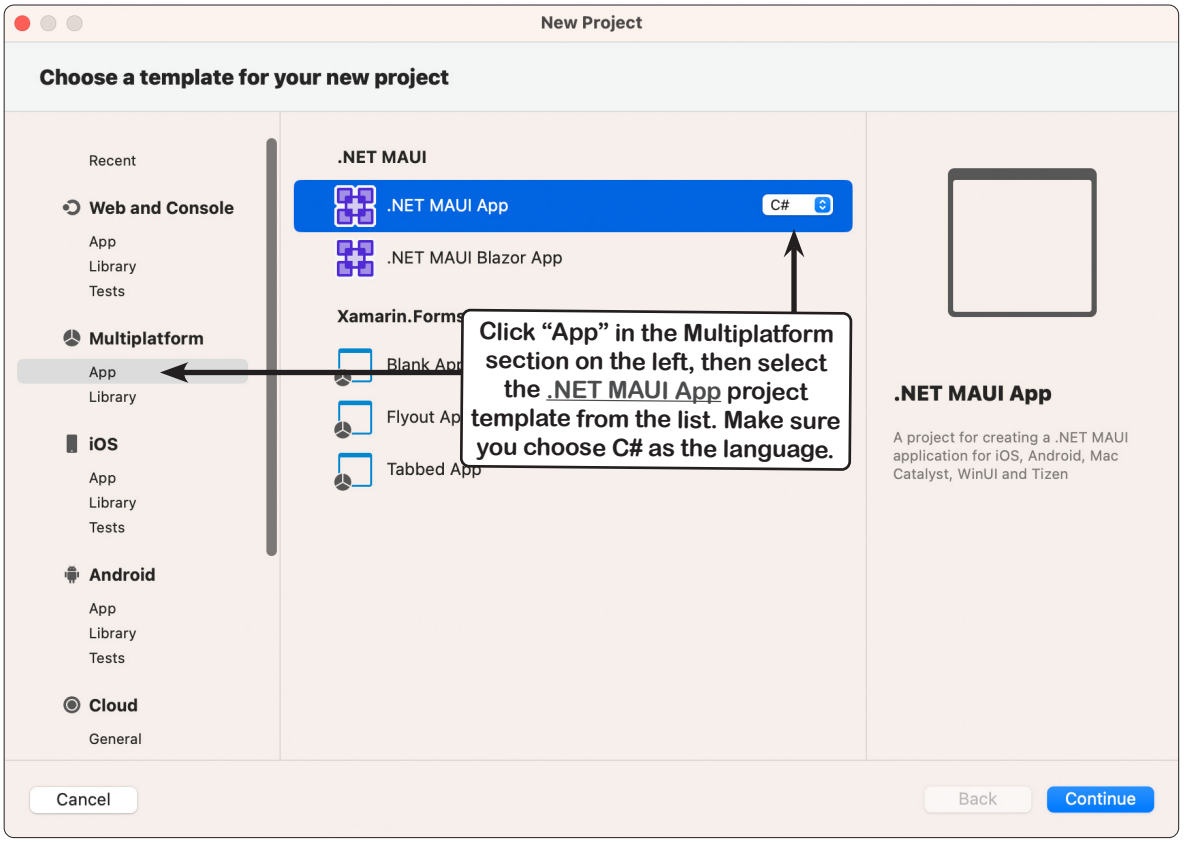
Enter **AnimalMatchingGame** as the project name and click Next.



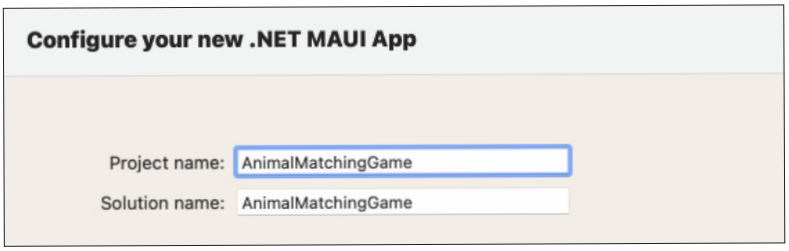
Finally, Visual Studio will ask you to choose a version of .NET – choose the latest version, just like you did when you created the Console App project. Then click the Create button to create your new .NET MAUI project.

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

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



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

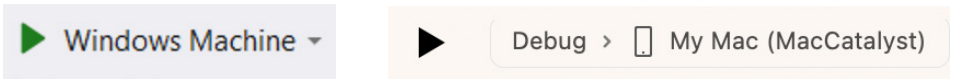


Make sure you enter AnimalMatchingGame as the project name, otherwise your code won't match the screenshots in the book.

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

Run your new .NET MAUI app

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

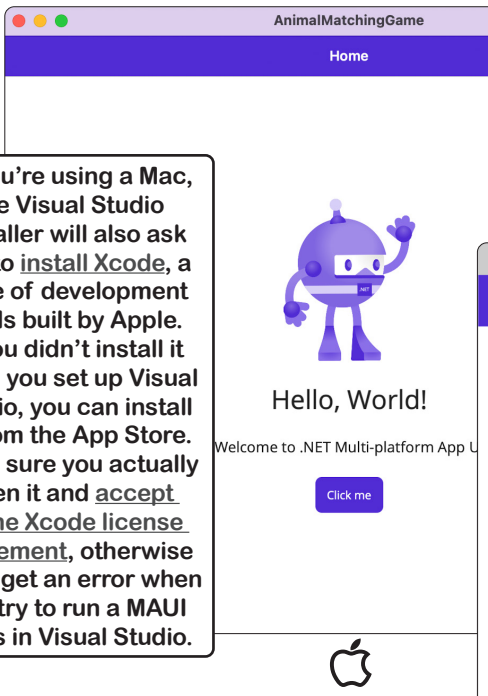


Do this!

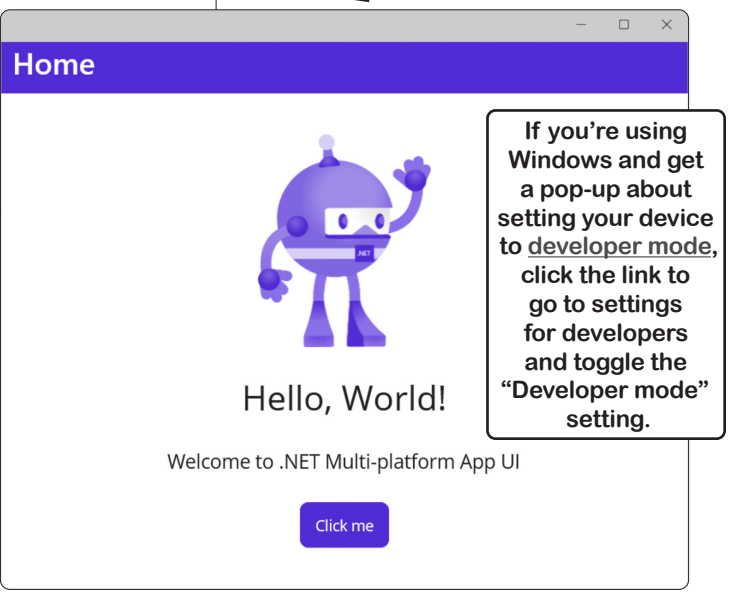
Or you can choose Start Debugging (F5 or ⌘↵) from the Debug menu.

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

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



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



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

Stop your MAUI app

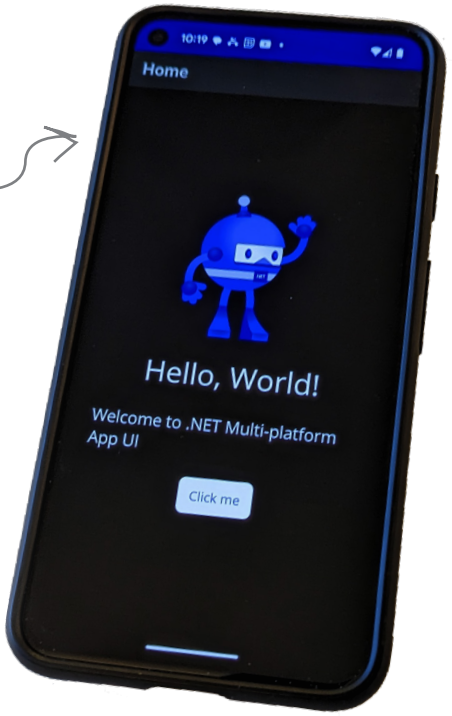
You can stop your app by closing the app window. You can also choose Stop Debugging (Shift+F5 or ⇧⌘↵) from the Debug menu, or click the square Stop button in the Visual Studio toolbar.

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

MAUI apps work on all of your devices

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

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



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

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

MAUI apps are designed with XAML

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

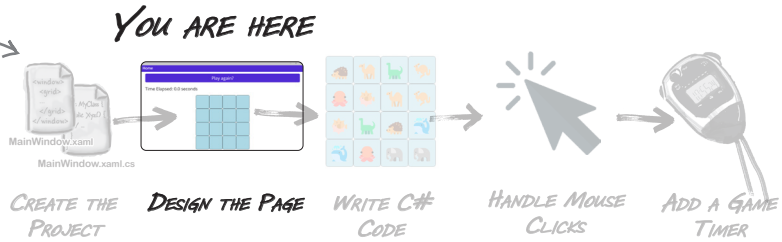
```
<Button Text="Click" Clicked="Button_Click" />
```

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

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

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

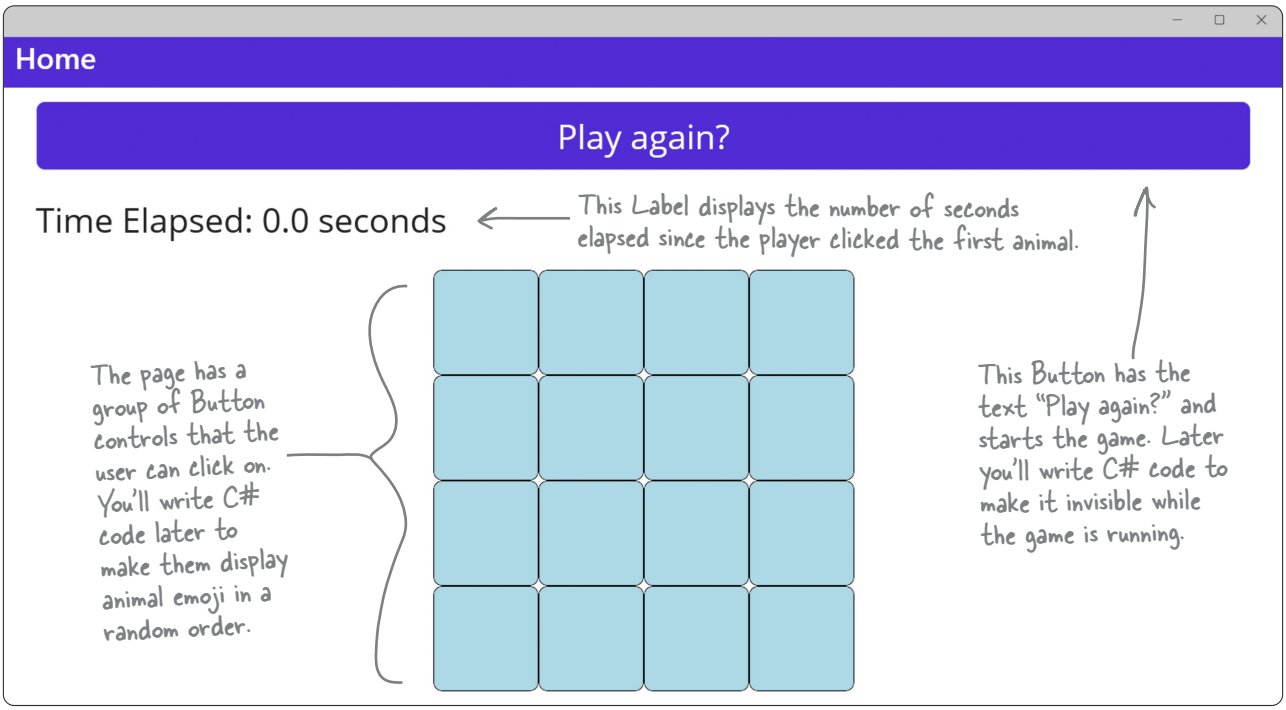
We'll include a "map" like this at the start of each of the sections of this project to help you keep track of the big picture.



Here's the page that you'll build

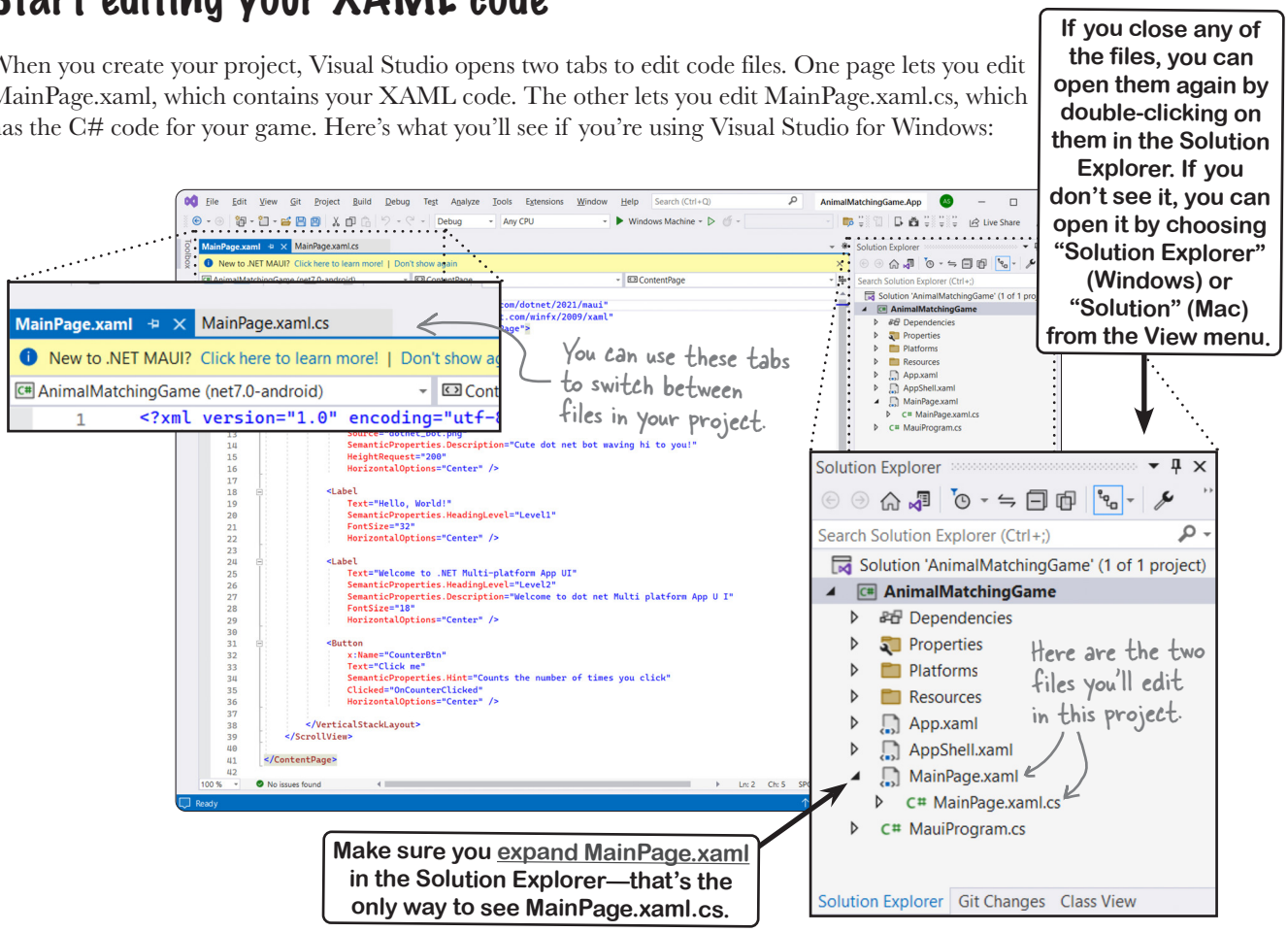
When you start a project, the first thing you always want to do is take a few minutes to understand the big picture. What are you going to create? How will it work? Let's take a look at the page you're about to build.

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

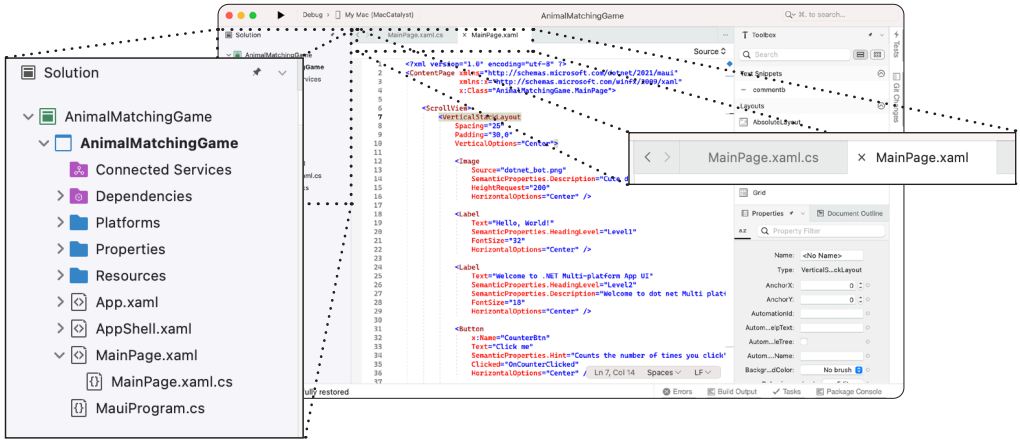


Start editing your XAML code

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



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



Add the XAML for a Button and a Label

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

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

```
<ScrollView>
```

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

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

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

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

Next, find the closing VerticalStackLayout tag on line 38:

```
        </VerticalStackLayout>
```

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

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

    <ScrollView>

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


        </VerticalStackLayout>
    </ScrollView>

</ContentPage>
```

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

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

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

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

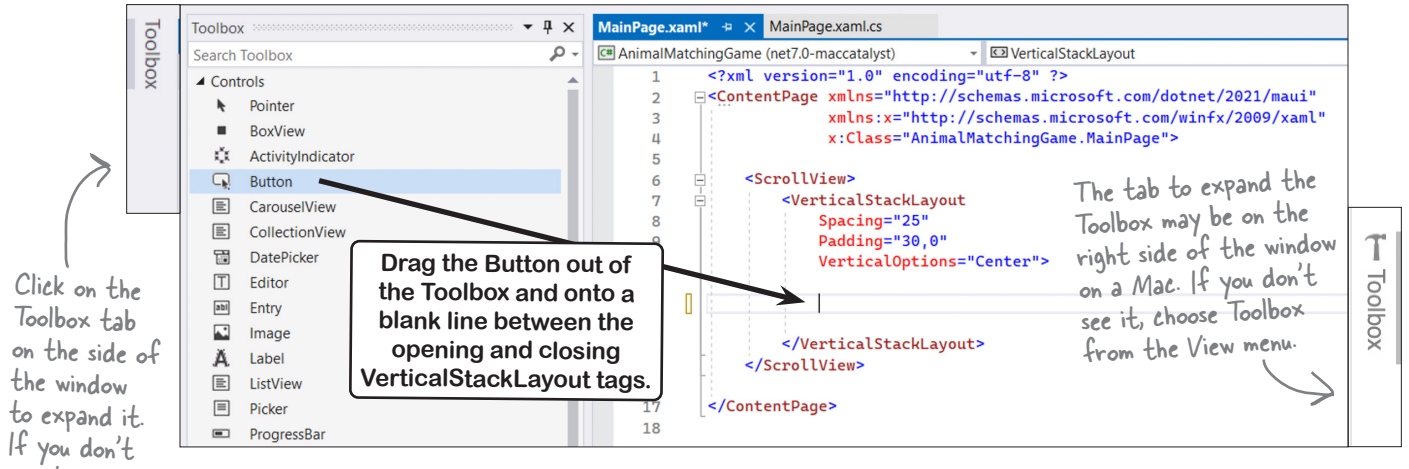
Delete it, and the next 10 lines of code, up to and including the closing curly brace `}` on line 22. Be careful not to delete the final closing `}` on line 24. Then delete the line of code on line 5: `int count = 0;`

Your C# code should now look like this:

```
namespace AnimalMatchingGame;

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

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

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



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

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

    <Button Text="" />

</VerticalStackLayout>
```

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

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

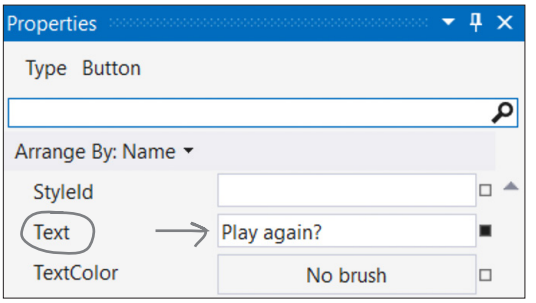
XAML tags have **properties** that let you set options to customize how they're displayed on the page. The **Properties window** in Visual Studio makes it easier to edit them.

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

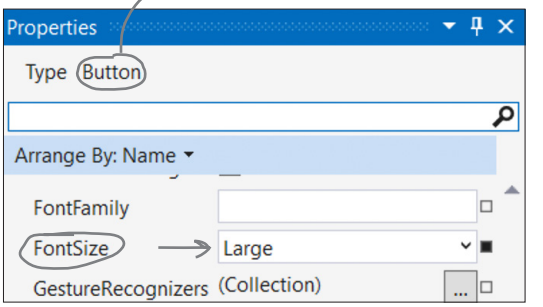
Find the ***Text*** property and **set it to `Play again?`**

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

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



Make sure you're editing the Button.



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

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

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

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

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

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

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

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

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

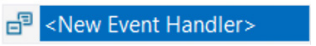
⑥ Add an event handler so your button does something.

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

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

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

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



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

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

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

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

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

}
```

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

Exercise

Add a Label control to your XAML page.

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

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

Exercise Solution

Add a Label control to your XAML page.

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

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

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

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

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

        </VerticalStackLayout>
    </ScrollView>

</ContentPage>
```

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

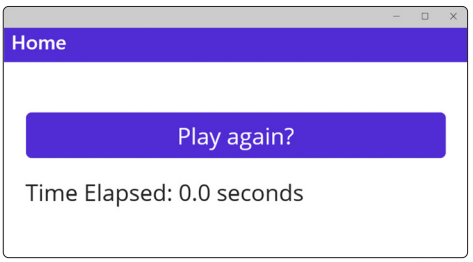
Here's the Label that you added in the Exercise.

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

```
namespace AnimalMatchingGame;

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

    private void PlayAgainButton_Clicked(object sender, EventArgs e)
    {

    }
}
```

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

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



there are no Dumb Questions

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

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

Q: So those buttons and labels are controls?

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

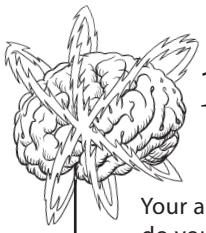
Q: It looks like some controls contain others, like the VerticalStackLayout in my app contains Button and a Label. What's going on there?

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

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

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

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



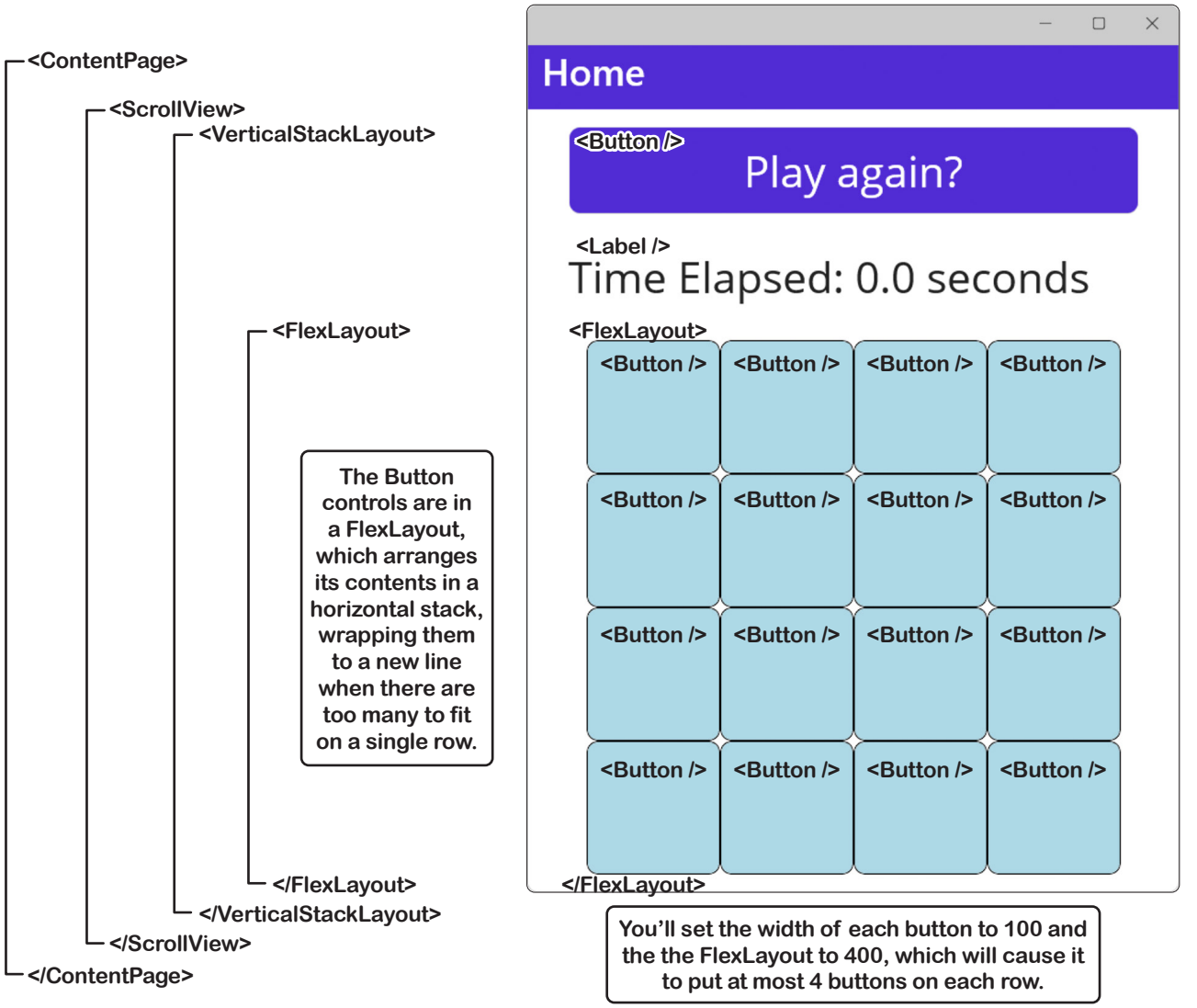
Brain Power

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

Use a FlexLayout to make a grid of animal buttons

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

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



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

Exercise

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

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

Add extra space for your FlexLayout control

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

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

Add the FlexLayout control just below the Label

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

Add the first Button inside the FlexLayout

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

Add the rest of the Buttons

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

Exercise Solution

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

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

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

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

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

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

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

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

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

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

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

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

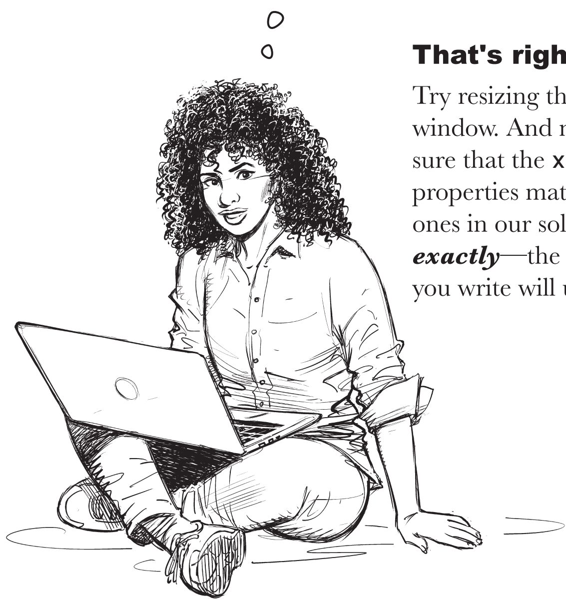
Exercise Solution

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

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

    </VerticalStackLayout>
  </ScrollView>

</ContentPage>
```

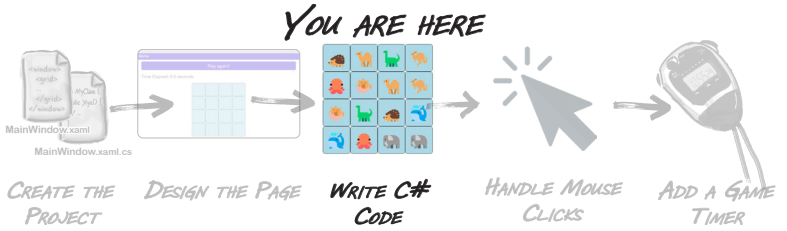
Make sure **every button** has the `Clicked="Button_Clicked"` property. If the Clicked event handler has a different name, your C# code won't match ours. You can delete the Clicked property from all of the buttons, then re-add it with the correct name. Once you add the event handler, it will show up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

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



That's right!

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



Write C# code to add the animals to the buttons

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

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

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

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

Start editing the PlayAgainButton event handler method

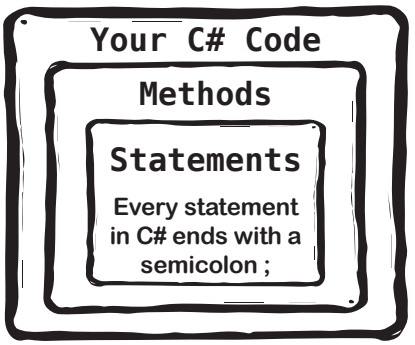
When you were writing the XAML code for the "Play again?" button, you added an event handler:

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

<New Event Handler>

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

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

}
```



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

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

Add a C# statement to the event handler method

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

Do this!

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

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

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

Tab Tab to accept | Alt+. next suggestion

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

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

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

Add more statements to your event handler

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

Do this!

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

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

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

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

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

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

2 Make the animal buttons invisible when the app starts.

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

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

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

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

    AnimalButtons.IsVisible = false;
}
```

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

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

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

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

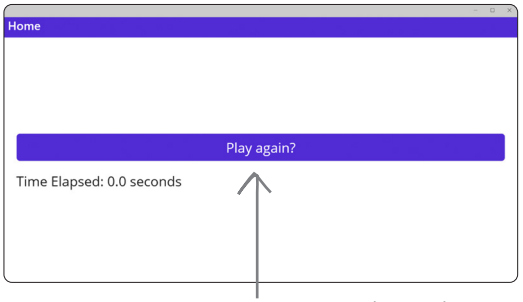
When you're writing code, you don't just write a complete app from beginning to end, then run it to see if it works. That's not how it works at all! ***Writing code is a creative process.*** There are many, many ways to make your code do a specific thing, and in a lot of cases, the only way you can really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

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

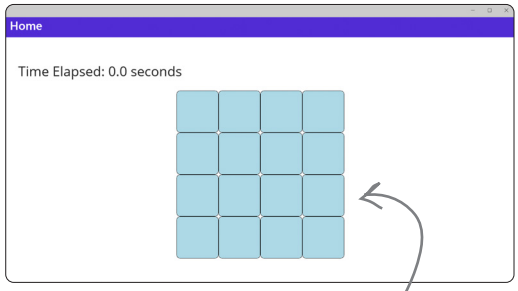
What does all that mean?

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

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



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



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

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

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

Add animals to your buttons

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

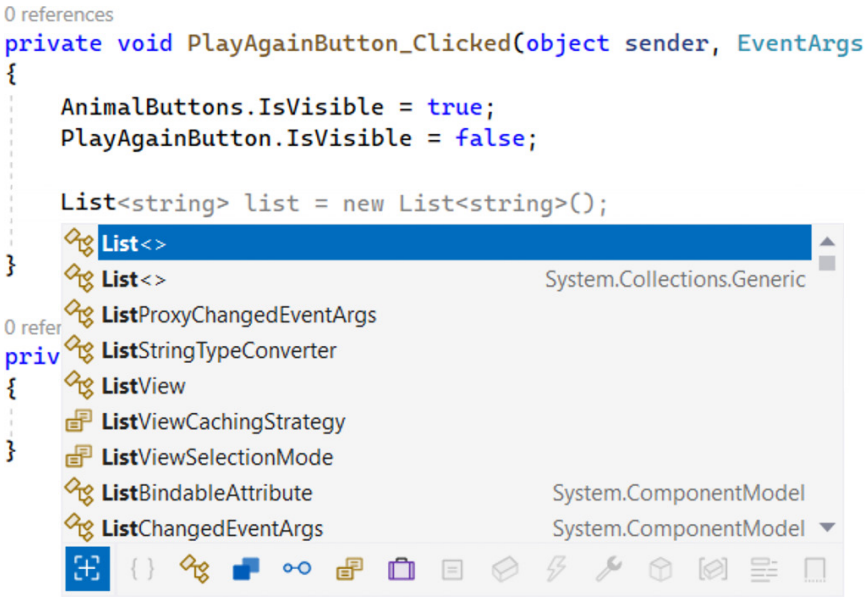
1 Start creating a List of animal emoji.

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

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

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

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



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

Your PlayAgainButton_Clicked event handler method should now look like this:

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

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

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

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

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

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

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

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

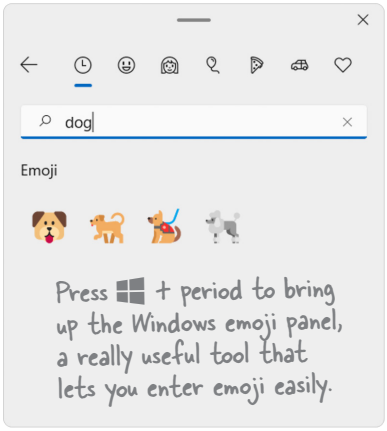
If you're using Windows, use the **Windows emoji panel** (press Windows logo key + period). If you're using a Mac, use the Character Viewer panel (choose "Show Emoji & Symbols" from the Input menu).

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

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

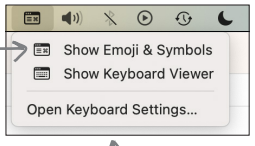
How to enter emoji

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



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

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



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

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

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

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

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

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

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

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

4 Finish the method.

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

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

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

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

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

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

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

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

```
namespace AnimalMatchingGame;
```

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

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

        AnimalButtons.IsVisible = false;
    }

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

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

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

    private void Button_Clicked(object sender, EventArgs e)
    {

    }
}
```

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

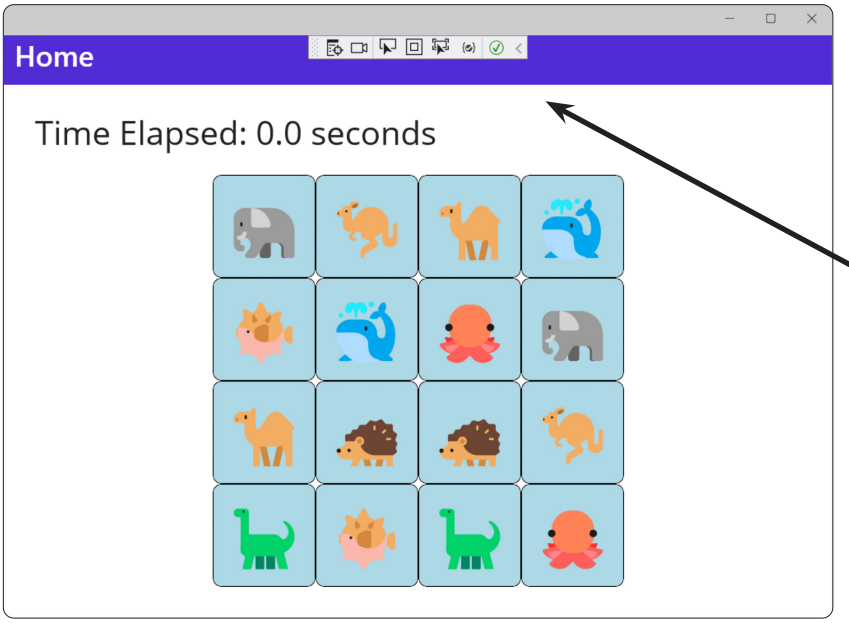
Make sure there are exactly eight matching pairs of emoji. That's part of what makes the game work.

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

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

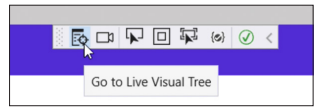
Run your app!

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

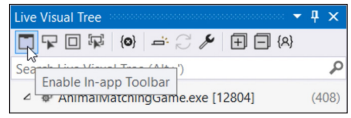


Stop it and run it again a few times. The animals should get reshuffled in a different order every time you click the "Play again?" button.

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



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



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

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

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

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

Who Does What?

C# statement

What it does

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

Make the button display the selected emoji

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

Make the "Play again?" button invisible

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

Create a list of eight pairs of emoji

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

Make the FlexLayout with the emoji buttons visible

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

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

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

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

Remove the chosen emoji from the list

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

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

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

Who Does What? Solution

C# statement

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

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

animalEmoji.RemoveAt(index);

button.Text = nextEmoji;

string nextEmoji = animalEmoji[index];

AnimalButtons.IsVisible = true;

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

PlayAgainButton.IsVisible = false;
```

What it does

Make the button display the selected emoji

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

Make the "Play again?" button invisible

Create a list of eight pairs of emoji

Make the FlexLayout with the emoji buttons visible

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

Remove the chosen emoji from the list

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

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

MINI Sharpen your pencil

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

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

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

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

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

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

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

Bullet Points

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

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

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

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

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

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

Git can help you do all of those things!

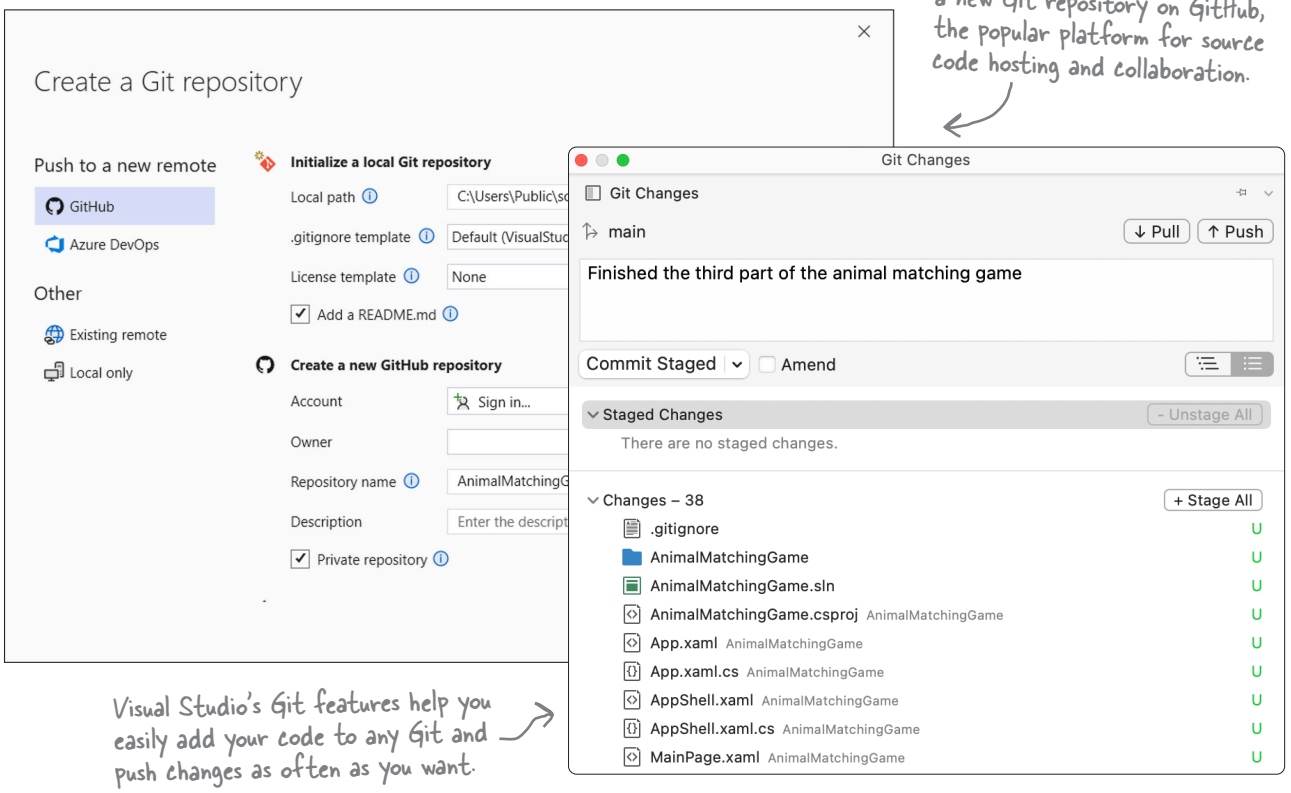
Here are just a few things Git can do for you

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

Visual Studio makes it easy to use Git

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

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

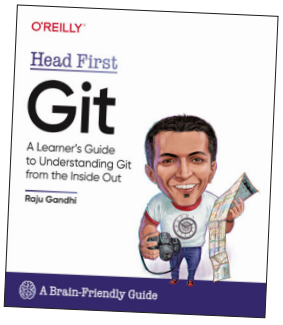


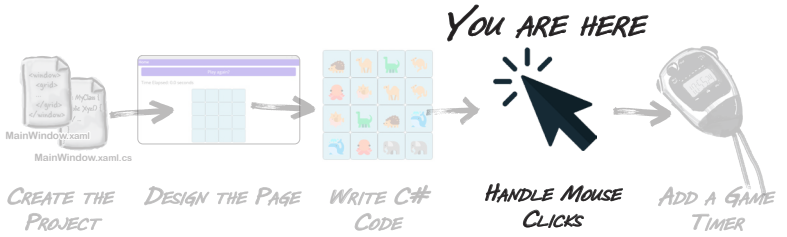
Visual Studio's Git features help you easily add your code to any Git and push changes as often as you want.

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

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

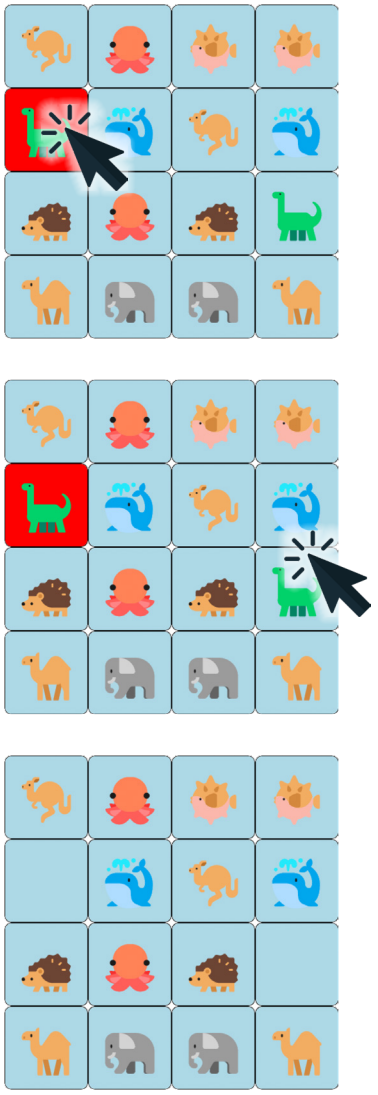
We'll give you everything you need to use Visual Studio to save and share your projects. But there is a lot more that you can do with Git, especially if you're working with large teams! If you're fascinated by what you see and want to do a deep dive into Git, check out Head First Git by Raju Gandhi.





Add C# code to handle mouse clicks

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



The player clicks the first button.

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

The player clicks the second button.

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

The game checks for a match.

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

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

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

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

Sharpen your pencil

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

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

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

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

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

1. What does `matchesFound` do?

2. What do these three lines of code do?

3. What does this block of code do?

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

Sharpen your pencil Solution

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

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

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

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

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

1. What does `matchesFound` do?

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

2. What do these three lines of code do?

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

3. What does this block of code do?

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

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

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

Enter the code for the event handler

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

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

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

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

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

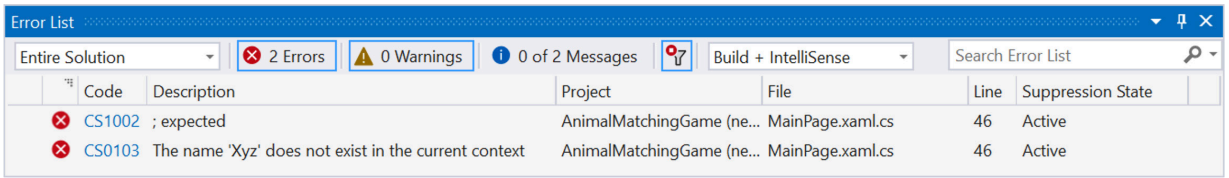
IDE Tip: The Error List

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

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

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

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



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

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

Run your app and find all the pairs

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

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

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

Uh-oh – there's a bug in your code

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

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

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

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

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

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

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

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

Sleuth it Out

The Case of the Unexpected Match

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

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

Every bug has a culprit

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

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

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

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

Read the code carefully and search for clues

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

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

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

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

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

Why did the bug happen?

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

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

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

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

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

Go back to the scene of the crime

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

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

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

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

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

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

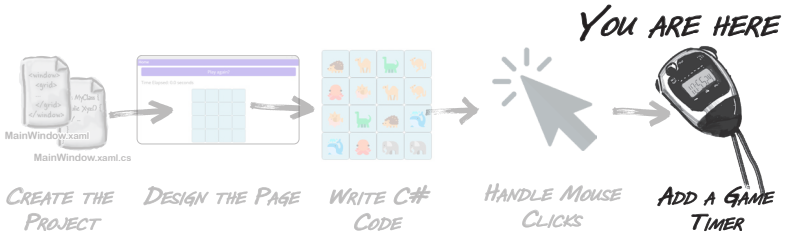
Position your cursor between the last two closing parentheses `))` in the if statement and press enter to add a line. Then enter the following code: `&& (buttonClicked.Text != "")`

Here's what your code should now look like:

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

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

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



Finish the game by adding a timer

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

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

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

Add a timer to your game's code

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

Do this!

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

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

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

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

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

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

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

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

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

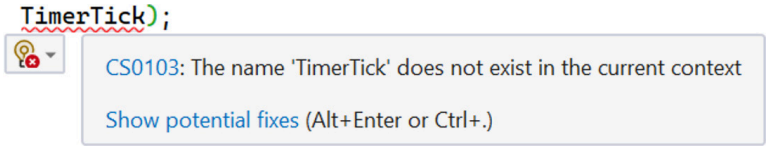
```
35              button.Text = nextEmoji;
36              animalEmoji.RemoveAt(index);
37          }
38
39          Dispatcher.StartTimer(TimeSpan.FromSeconds(.1), TimerTick);
40      }
41
```

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

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

③ Use Visual Studio to generate a new TimerTick method.

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



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

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

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

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

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

Finish the code for your game

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

Do this!

Add a field to hold the time elapsed

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

Add this line of code:

```
int tenthsOfSecondsElapsed = 0;
```

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

```
private bool TimerTick()
```

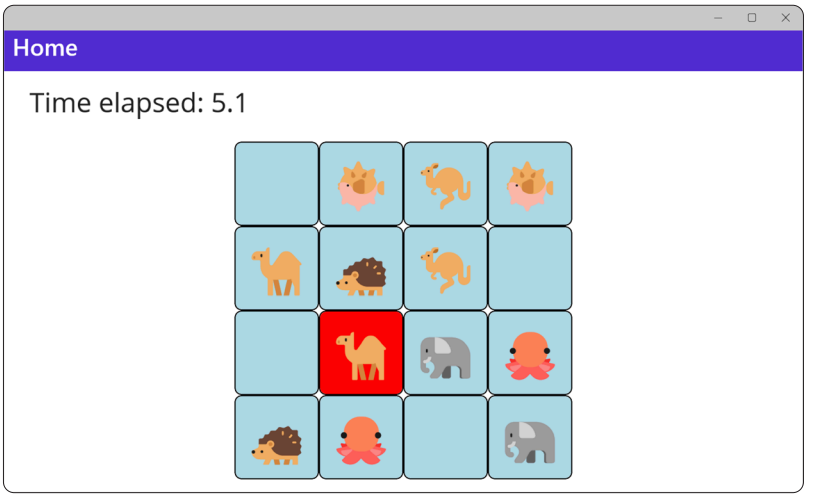
Finish your TimerTick method

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

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

    tenthsOfSecondsElapsed++;

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

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

    return true;
}
```

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

Run your game. Now the timer works!



Your TimerTick Method Up Close

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

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

    tenthsOfSecondsElapsed++;

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

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

    return true;
}
```

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

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

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

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

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

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

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

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

Even better ifs...

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

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

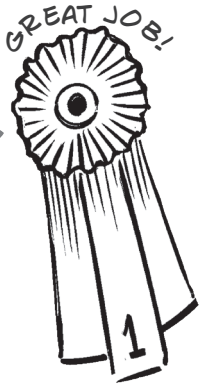
MINI

Sharpen your pencil

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

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

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



Did you save your code Git?

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

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

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

Bullet Points

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