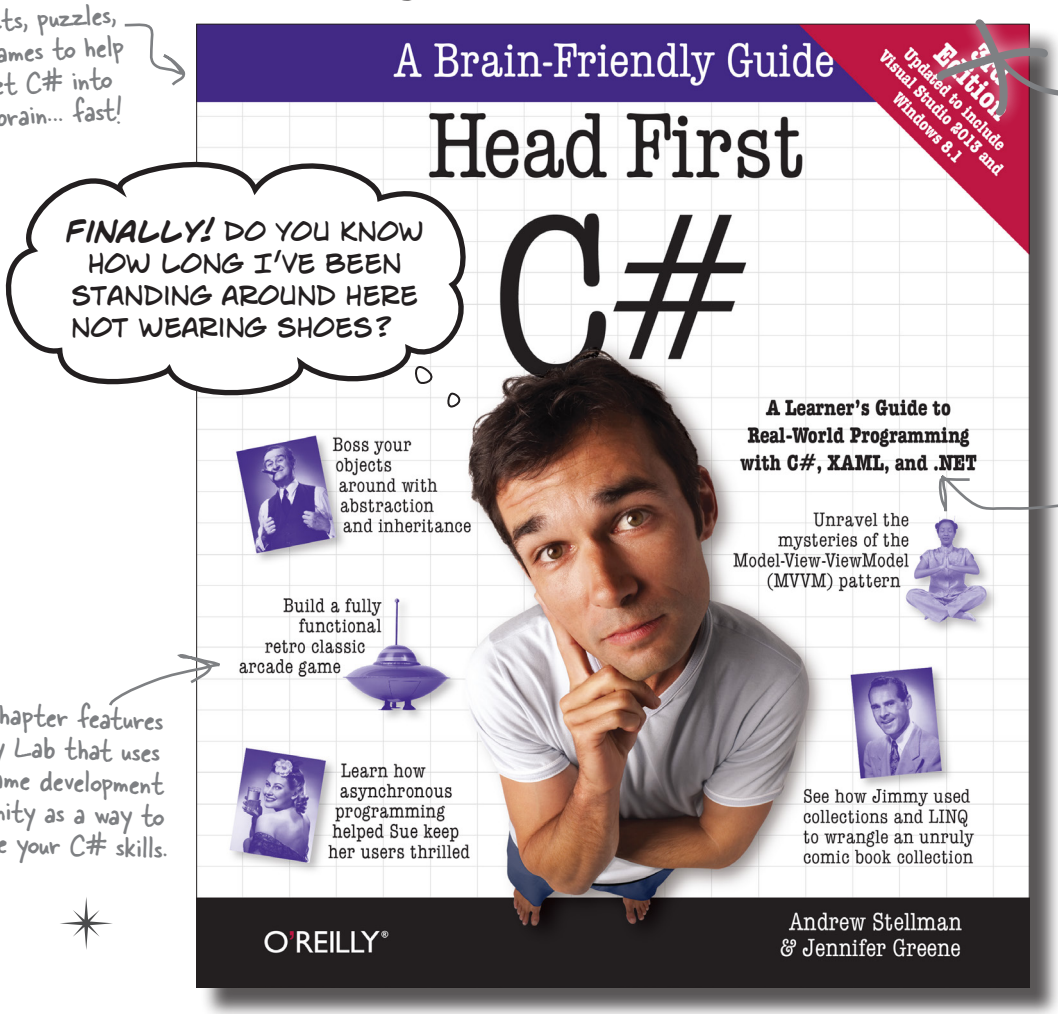
Download the latest version of this PDF from our GitHub repo:
https://github.com/head-first-csharp/fourth-edition/

Head First C# 4th Edition

Early Release Chapters

Packed full of projects, puzzles, and games to help you get C# into your brain... fast!



4th edition updated for Windows 10 and Visual Studio 2019 coming really soon!

Now featuring cross-platform development with .NET Core, and a full MacOS learning path featuring Visual Studio Mac.

Each chapter features a Unity Lab that uses 3-D game development with Unity as a way to practice your C# skills.

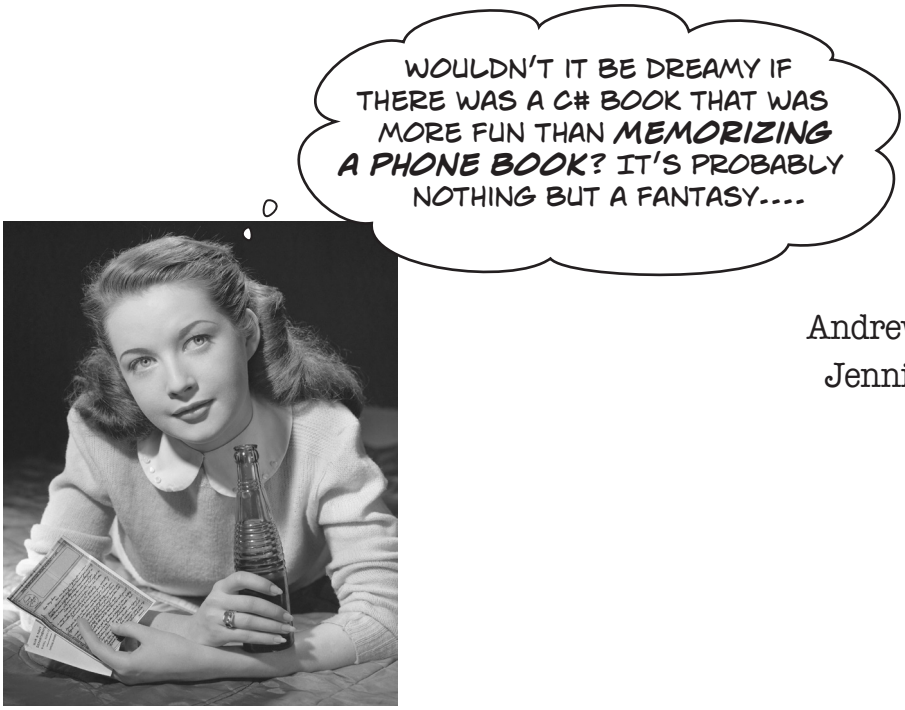
We're back, and better than ever.

There's never been a better time to learn C#, and the **4th edition of Head First C#** may just be be the easiest, most fun way to for you to do it. But don't take our word for it! Have a look at these **early release chapters** to see what's coming soon!

Follow us on Twitter: @HeadFirstCsharp • @AndrewStellman • @JennyGreene

Head First C#

Fourth Edition



Andrew Stellman
Jennifer Greene

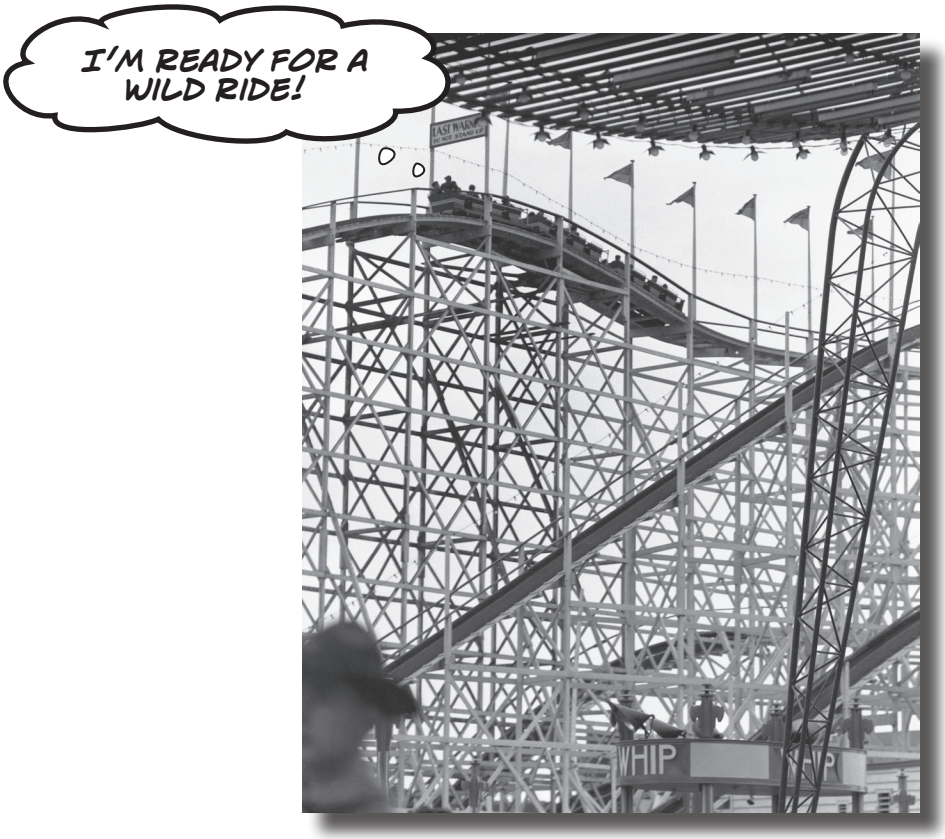
O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Early Release Chapters

1 start building with c#

Build something great... fast!



Want to build great apps... right now?

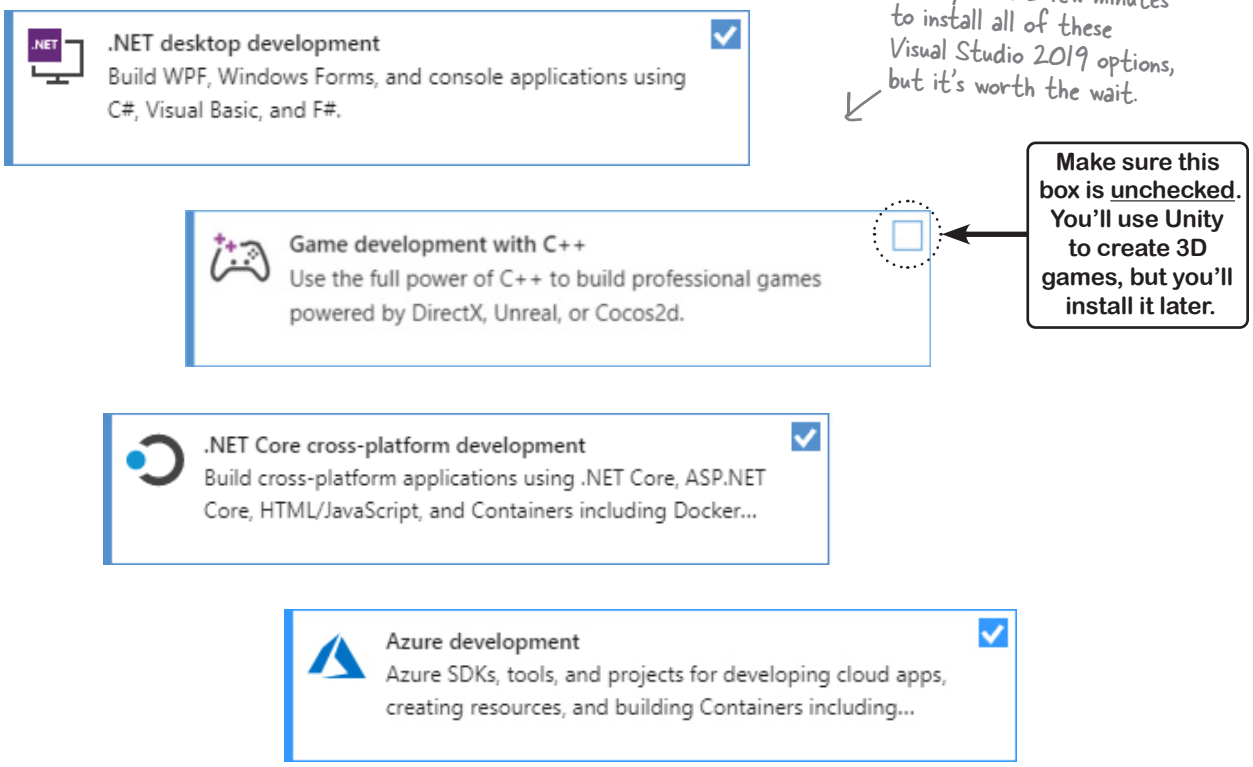
With C#, you've got a **modern programming language** and a **valuable tool** at your fingertips. And with the **Visual Studio IDE**, you've got an **amazing development environment** with highly intuitive features that make coding as easy as possible. But not only is Visual Studio a great tool for writing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

Why you should learn C#

C# is a simple, modern language that lets you do incredible things. And 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 programs: desktop, web, and mobile apps; cloud computing; machine learning and artificial intelligence; 2D and 3D gaming; and much, much more.

Visual Studio is your gateway to C#

If you haven't installed Visual Studio 2019 yet, this is the time to do it. Go to https://visualstudio.microsoft.com/ and **download the Visual Studio IDE community edition**. (If it's already installed, run the Visual Studio Installer to update your installed options.) Make sure you check these four options to install support for .NET desktop development, 2D and 3D game development with Unity, .NET Core cross-platform development, and cloud projects with Azure:



It may take a few minutes to install all of these Visual Studio 2019 options, but it's worth the wait.

Make sure this box is unchecked. You'll use Unity to create 3D games, but you'll install it later.

And while you're there, scroll down and read through all of the other options. Learning C# is the first step in doing all of those things. While it's installing, take go to https://dotnet.microsoft.com/ and learn more about the exciting kinds of apps, tools, and programs that you can build with C#.

Visual Studio is a tool for writing code and exploring C#

You could use Notepad or another text editor to write your C# code, but there's a better way. 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.

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

1. **Build an application, FAST.** The C# language is flexible and easy to learn, and the Visual Studio IDE makes it easier by doing a lot of manual work for you automatically. Here are just a few things that Visual Studio does for you:

   - Manages all your project files
   - Makes it easy to edit your project's code
   - Keeps track of your project's graphics, audio, icons, and other resources
   - Helps you debug your code by stepping through it line by line

2. **Design a great-looking user interface.** The Visual Designer in the Visual Studio IDE is one of the easiest-to-use design tools out there. It does so much for you that you'll find that creating user interfaces for your programs is one of the most satisfying parts of developing a C# application. You can build full-featured professional programs without having to spend hours tweaking your user interface (unless you want to).

3. **Build visually stunning programs.** When you **combine C# with XAML**, the visual markup language for designing user interfaces for WPF desktop applications,

   The user interface (or UI) for any WPF is built with XAML (which stands for eXtensible Application Markup Language). Visual Studio makes it really easy to work with XAML.

   you're using one of the most effective tools around for creating visual programs... and you'll use it to build software that looks as great as it acts.

4. **Learn and explore C# and .NET.** Visual Studio is a world-class development tool, but lucky for us it's also a fantastic learning tool. ***We're going to use the IDE to explore C#***, which gives us a fast track for getting important programming concepts into your brain *fast*.

Visual Studio is an amazing development environment, but we're also going to use it as a learning tool to explore C#.

Create your first project in Visual Studio

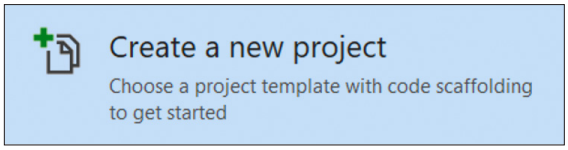
The best way to learn C# is to start writing code, so we're going to use Visual Studio to **create a new project**... and start writing code immediately!

Do this!

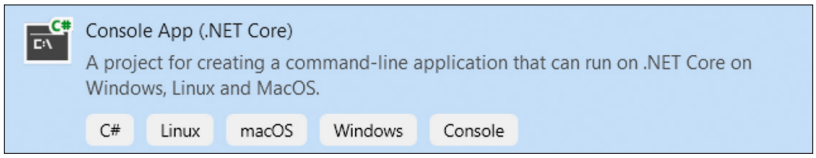
When you see Do this! (or Now do this!, or Debug this!, etc.) go to Visual Studio and follow along. We'll tell you exactly what to do, and point out what to look for to get the most out of the example we show you.

1 Create a new Console App (.NET Core) project.

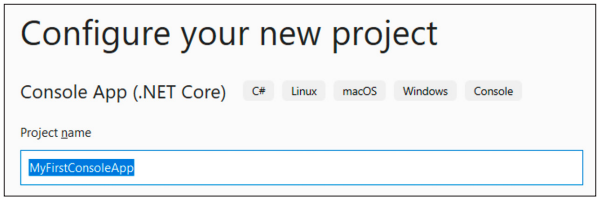
Start up Visual Studio 2019. When it first starts up, it shows you a "Get Started" page with a few different options. Choose **Create a new project**.



If hit a key and now you see a File menu at the top of the screen, you can create a new project by choosing New >> Project... from the File menu (that is, click File, then click New, then click Project...). Choose the **Console App (.NET Core)** project type, then press the **Next button**.



Name your project MyFirstConsoleApp.



2 Look at the code for your new app.

When Visual Studio creates a new project, it gives you a starting point that you can build on. As soon as it finishes creating the new files for the app, it should open display a file called Program.cs with this code:

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

When Visual Studio creates a new Console App project, it automatically adds a class called Program.

The class starts out with a method called Main, which contains a single statement that writes a line of text to the console.

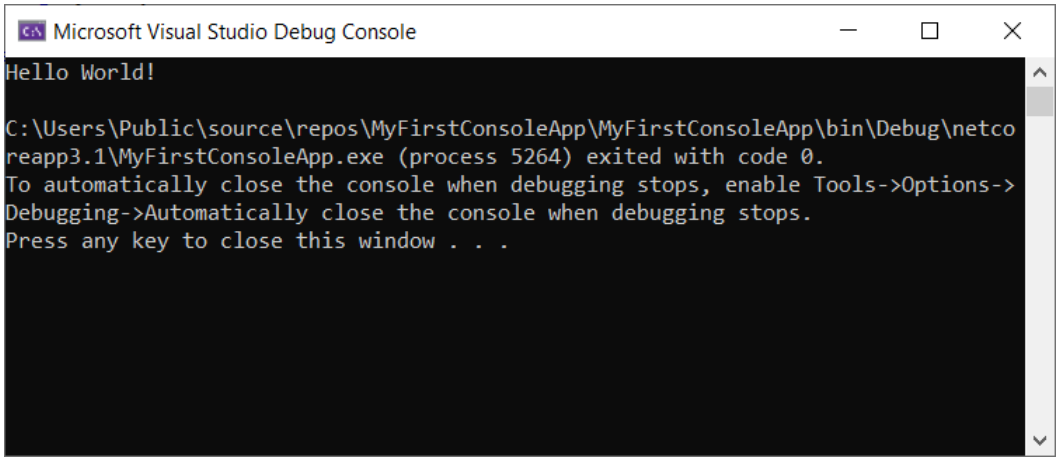
3 Run your new app.

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

4 Look at your program's output.

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

When you ran your app it executed the Main method, which wrote this line of text to the console.



The best way to learn a language is to write a lot of code in it, so you're going to build a lot programs in this book. Many of them will be .NET Core Console App projects, so let's have a closer look at what you just did.

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\
netcoreapp3.1\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 . . .
```

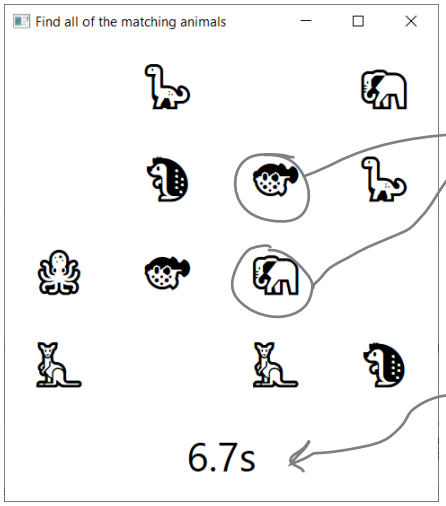
You'll see the same message at the bottom of every Debug Console window. Your program printed a single line of text ("`Hello World!`") and then exited. Visual Studio is keeping the output window open until you press a key to close it so you can see the output before the window disappears.

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.

Let's build a game!

You've built your first C# app, and that's great! But it doesn't do much, does it? So let's dive right in and **create a game**. This will give you a solid foundation to start exploring C#—and in the process, you'll work with important Visual Studio tools that you'll use throughout the book.

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



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

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

There are several different technologies that let you build desktop apps for Windows. We chose WPF because this particular type of project gives you tools to design highly detailed user interfaces that run on many different versions of Windows, even very old editions like Windows XP.

But C# isn't just for Windows!

Are you a Mac user? Well, then you're in luck! We added a learning path just for you, featuring Visual Studio 2019 for Mac.. See the Mac Learning Path appendix at the end of this book—it has a complete replacement for this chapter, and Mac versions of all of the WPF projects that appear throughout the book.

Your animal match game is a WPF app

Console apps are great if you just need to input and output text. But if you want a visual app that's displayed in a window, you'll need to use a different technology. That's why your animal match game will be a **WPF app**. WPF—or Windows Presentation Foundation—lets you create desktop applications that can run on any version of Windows. Most of the chapters in this book will feature one WPF app. The goal of this project is to introduce you to WPF and give you tools to build visually stunning desktop applications as well as console apps.

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

Here's how you'll build your game

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

1. First you'll create a new desktop application project in Visual Studio
2. Then you'll use XAML to build the window
3. You'll write C# code to add random animal emoji to the window
4. The game needs to let the user click on pairs of emoji to match them
5. Finally, you'll make the game more exciting by adding a timer

XAML – or eXtensible Application Markup Language – is a powerful tool that you'll use throughout the book to build user interfaces for your apps.

Create the Project | Design the Window | Write C# Code | Handle Mouse Clicks | Add a Game Timer

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.

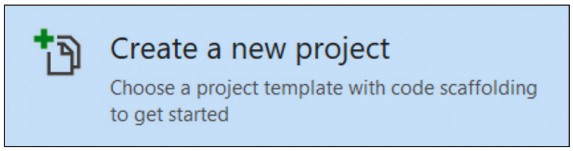
- Do all games have a **winner**? Do they always end? Not necessarily. What about a flight simulator? A game where you design an amusement park? What about a game like The Sims?
- Are games always **fun**? Not for everyone. Some players like a "grind" where they do the same thing over and over again; 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. And 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 at least as entertaining to play as it is to build.

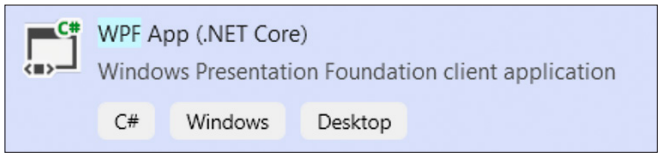
Create a WPF project in Visual Studio

We're going to build an **animal matching game**, where a player is shown a grid of 16 animals and needs to click on pairs to make them disappear. Go ahead and **start up a new instance of Visual Studio 2019** and create a new project:

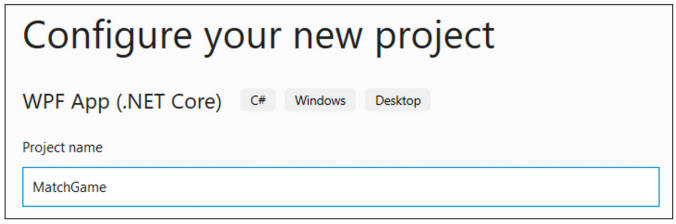
We're done with the Console App project you created in the first part of this chapter, so feel free to close that instance of Visual Studio.



We're going to build our game as a desktop app using WPF, so **select WPF App (.NET Core)** and click Next:



Visual Studio will ask you to configure your project. **Enter `MatchGame` as the project name** and click Next (you can also change the location to create the project if you'd like):



Click the Create button. Visual Studio will create a new project called MatchGame.

Visual Studio created a project folder full of files for you

As soon as you created the new project, the IDE added a new folder called MatchGame and filled it with all of the files and folders that your project needs. You'll be making changes to two of these files, MainWindow.xaml and MainWindow.xaml.cs.

This file contains the XAML code that defines the user interface of the main window.

MainWindow.xaml

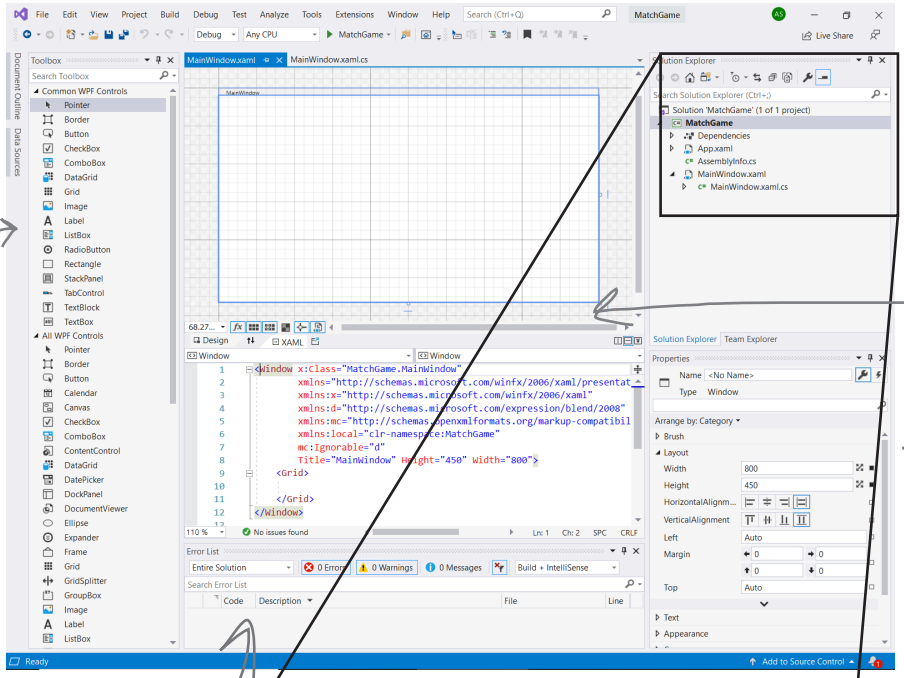
MainWindow.Xaml.cs

The C# code that makes your game work will go in here.

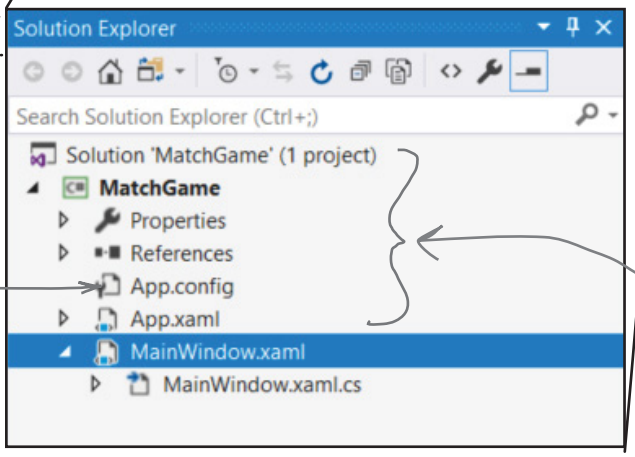
Sharpen your pencil

The pencil-and-paper exercises throughout the book aren't optional. They're an important part of learning, practicing, and leveling up your C# skills.

Adjust your IDE to match the screenshot below. First, **open *MainWindow.xaml*** by double-clicking on it in the Solution Explorer window. Then open the *Toolbox* and *Error List* windows by **choosing them from the View menu**. You can actually figure out the purpose of many of these windows and files based on their names and common sense! Take a minute and **fill in each of the blanks**—try to fill in a note about what each part of the IDE does. We've done one to get you started. See if you can take an educated guess at the others.



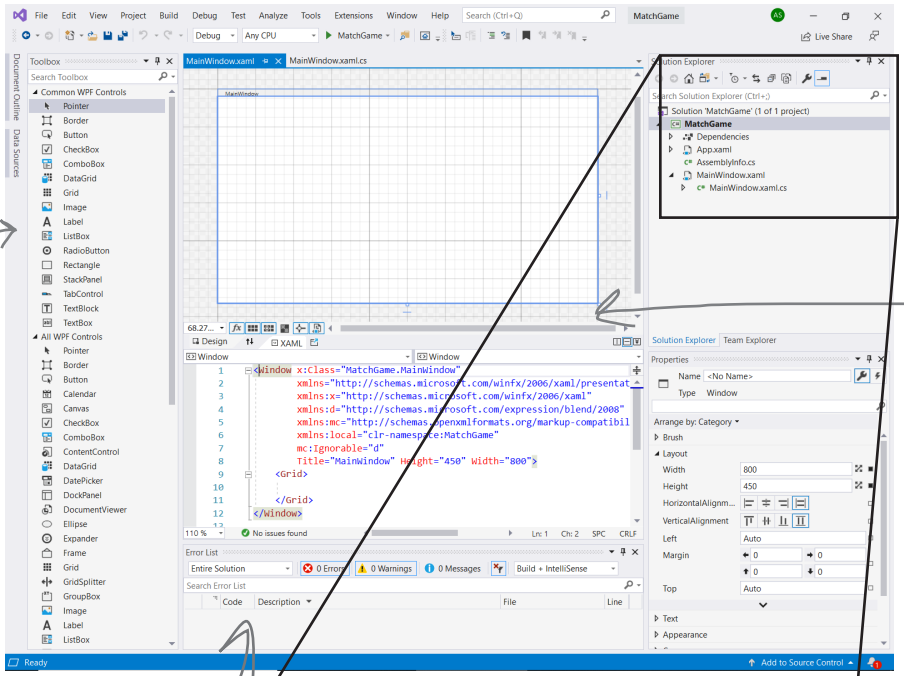
The designer lets you edit the user interface by dragging controls onto it.



We're using the Light color theme to make it easier to see our screenshots. Switch between color themes by choosing "Options..." from the Tools menu and clicking Environment.

Sharpen your pencil Solution

We've filled in the annotations about the different sections of the Visual Studio C# IDE. You may have some different things written down, but hopefully you were able to figure out the basics of what each window and section of the IDE is used for. Don't worry if you came up with a slightly different answer from us! You'll get LOTS of practice using the IDE.

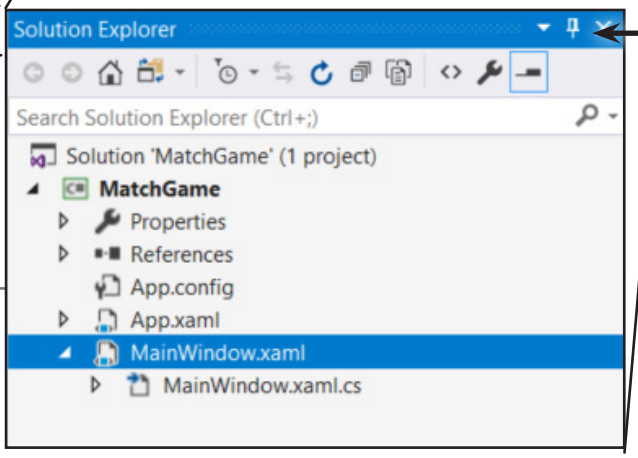


This is the toolbox. It has a bunch of visual controls that you can drag onto your window.

The designer lets you edit the user interface by dragging controls onto it.

This window shows properties of whatever is currently selected in your designer.

This Error List window shows you when there are errors in your code. This pane will show diagnostic info about your app.



See this little pushpin icon? If you click it, you can turn auto-hide on or off. The Toolbox window has auto-hide turned on by default.

The C# and XAML files that the IDE created for you when you added the new project appear in the Solution Explorer, along with any other files in your solution.

You can switch between files using the Solution Explorer in the IDE.

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. But the hard 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: Is it OK that I downloaded and installed Visual Studio Community Edition? Or do I 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 get in the way of writing C# and creating fully functional, complete applications.

Q: You said something about combining C# and XAML. What is XAML, and how does it combine with C#?

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

```
<Ellipse Fill="Gray"
  Height="100" Width="75"/>
```

If you type in that tag right after <Grid>, a grey ellipse will appear in the middle of your window. You can tell that that's a tag because it starts with a < followed by a word ("`Ellipse`"), which makes it a **start tag**. This particular `Ellipse` tag has three **properties**: one to set its fill color to gray, and two to set its height and width. This tag ends with `/>`, but some XAML tags can contain other tags. We can turn this tag into a **container tag** by replacing `/>` with a `>`, adding other tags (which can also contain additional tags), and closing it with an **end tag** that looks like this: `</Ellipse>`.

You'll learn a lot more about how XAML works and many different XAML tags throughout the book.

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 Toolbox and Error List windows. That will make your screen look like the ones in this chapter.

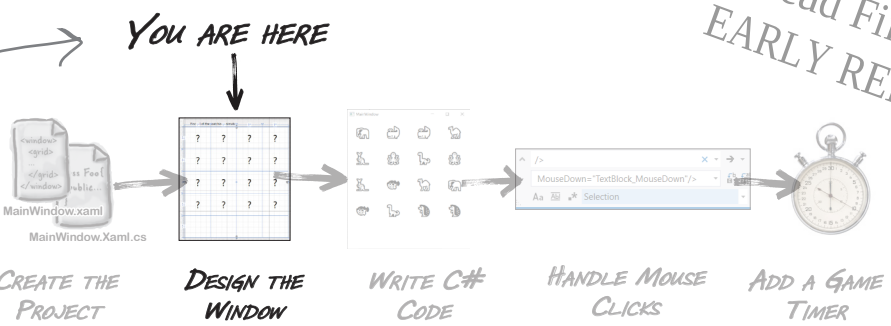
Visual Studio will generate code you can use as a starting point for your applications.

Making sure the app does what it's supposed to do is entirely up to you.

The Toolbox collapses by default. Use the pushpin button in the upper right corner of the Toolbox window to make it stay open.

xaml** rhymes with **camel

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



Use XAML to design your window

Now that Visual Studio created a WPF project for you, it's time to start working with **XAML**.

XAML, which stands for **Extensible Application Markup Language**, is a really flexible, XML-based markup language that C# developers use to design user interfaces. Yes, you'll be building an app with two different kinds of code. First you'll design the user interface (or UI) with XAML. Then you'll add C# code to make the game run.

If you've ever used HTML to design a web page, then you'll see a lot of similarities with XAML. Here's a really quick example of a simple window layout in XAML:

We added numbers to the parts of the XAML that defined text.

```
<Window x:Class="MyWPFApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="This is a WPF window" Height="100" Width="400">1
    <StackPanel HorizontalAlignment="Center" VerticalAlignment="Center">
        <TextBlock FontSize="18px" Text="XAML helps you design great user interfaces."/>2
        <Button Width="50" Margin="5,10" Content="I agree!"/>3
    </StackPanel>
</Window>
```

Look for the corresponding numbers in the screenshot below.

And here's what that window looks like when WPF **renders** it (or draws it on the screen). It draws a window with two visible **controls**, a TextBlock control that displays text and a Button control that the user can click on. They're laid out using an invisible StackPanel control, which causes them to be rendered one on top of the other. Look at the controls in the screenshot of the window, then go back to the XAML and find the TextBlock and Button tags.

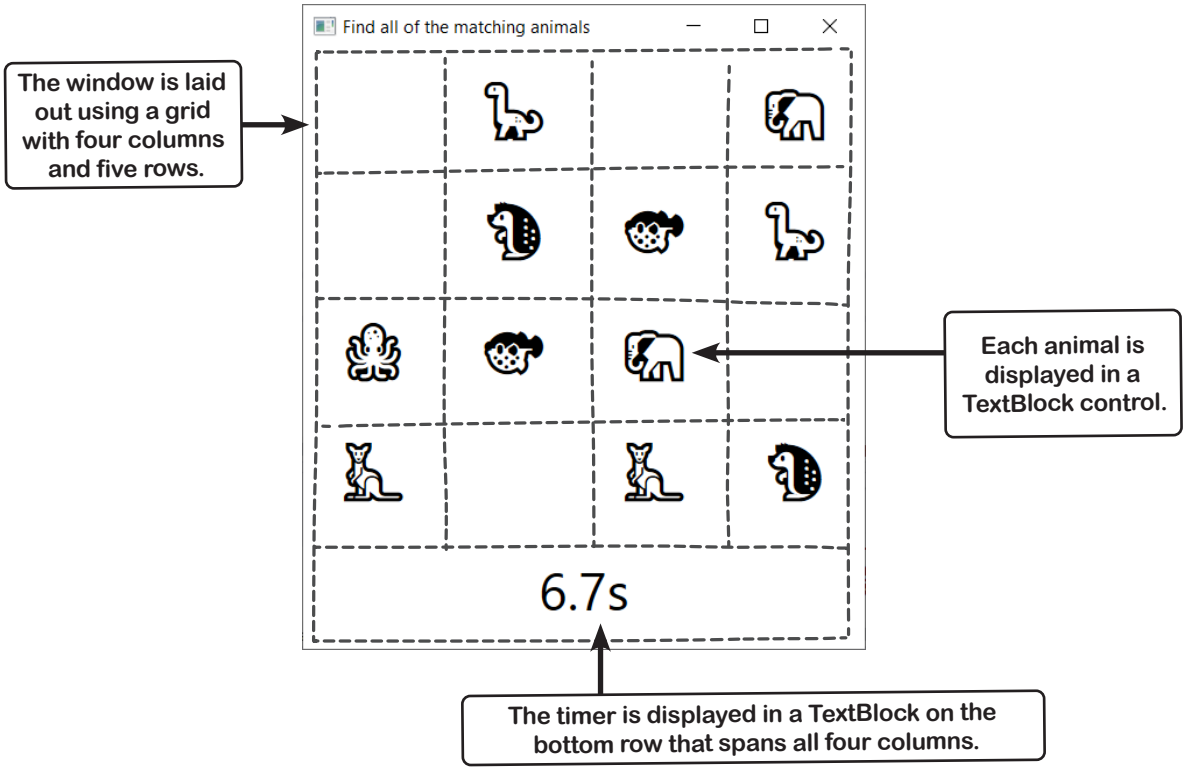
A TextBlock control does exactly what it sounds like it does—it displays a block of text.

These numbers in the screenshot show the text that corresponds to each number we added to the XAML code.

Design the window for your game

You're going to need an application with a graphical user interface, objects to make the game work, and an executable to run. It sounds like a lot of work, but you'll build all of this over the rest of the chapter, and by the end, you'll have a pretty good handle on how to use Visual Studio to design a great-looking WPF app.

Here's the layout of the window for the app we're going to create:



XAML is an important skill for C# developers.

You might be thinking, "Wait a minute! This is *Head First C#*. Why am I spending so much time on XAML? Shouldn't we be concentrating on C#?"

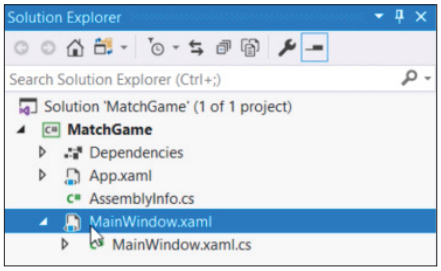
WPF applications use XAML for user interface design—and so do other kinds of C# projects. Not only can you use it for desktop apps, you can also use the same skills to build C# Android and iOS mobile apps with Xamarin Forms, which uses a variant of XAML (with a *slightly* different set of controls). That's why building user interfaces in XAML is an important skill for any C# developer, and why you'll learn a lot more about XAML throughout the book. We'll walk you through building the XAML ***step by step***—you can use the drag-and-drop tools in the Visual Studio 2019 XAML designer to create your user interface without a lot of tedious typing.

Set the window size and title with XAML properties

Let's start building the UI for your animal matching game. The first thing you'll do is make the window narrower and change its title. And in the process, you'll start to get acquainted with Visual Studio's XAML designer, a powerful tool for designing great-looking user interfaces for your apps.

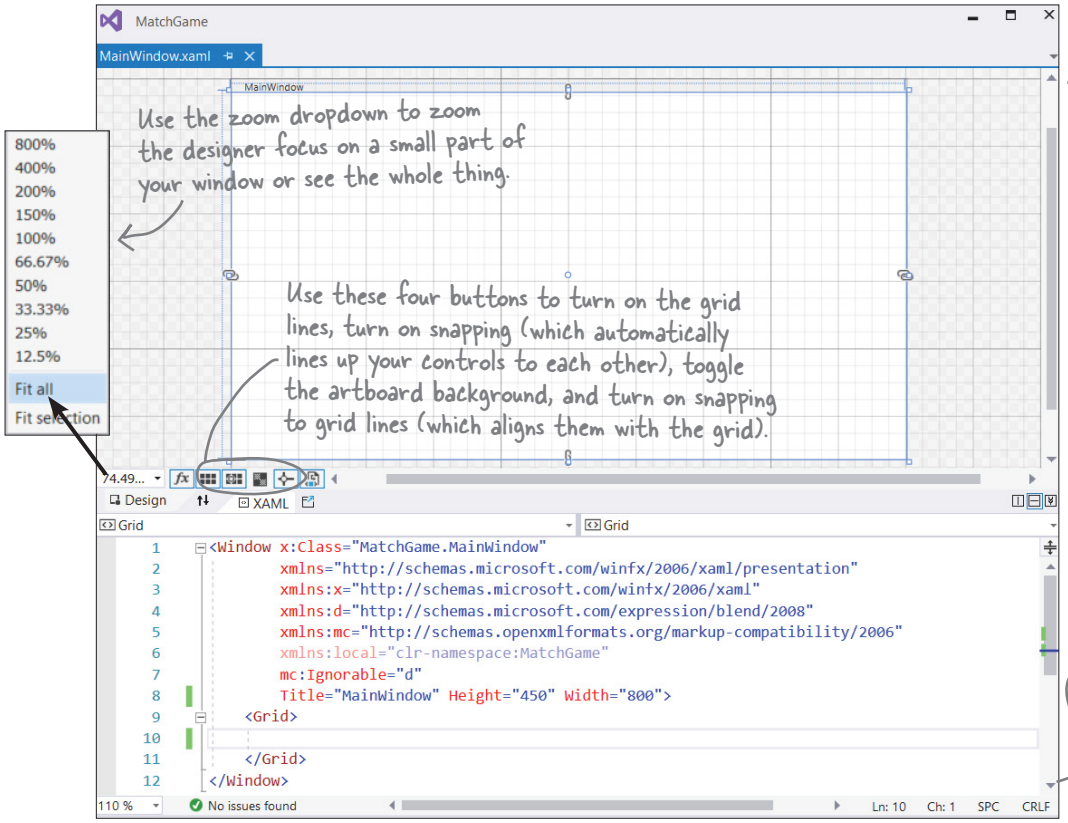
1 **Select the main window.**

Double-click on MainWindow.xaml in the Solution Explorer.



Double-click on a file in the Solution Explorer to open it in the appropriate editor. C# code files that end with .cs will be opened in the code editor. XAML files that end with .xaml will open up in the XAML designer.

As soon as you do, Visual Studio will open it up in the XAML designer.



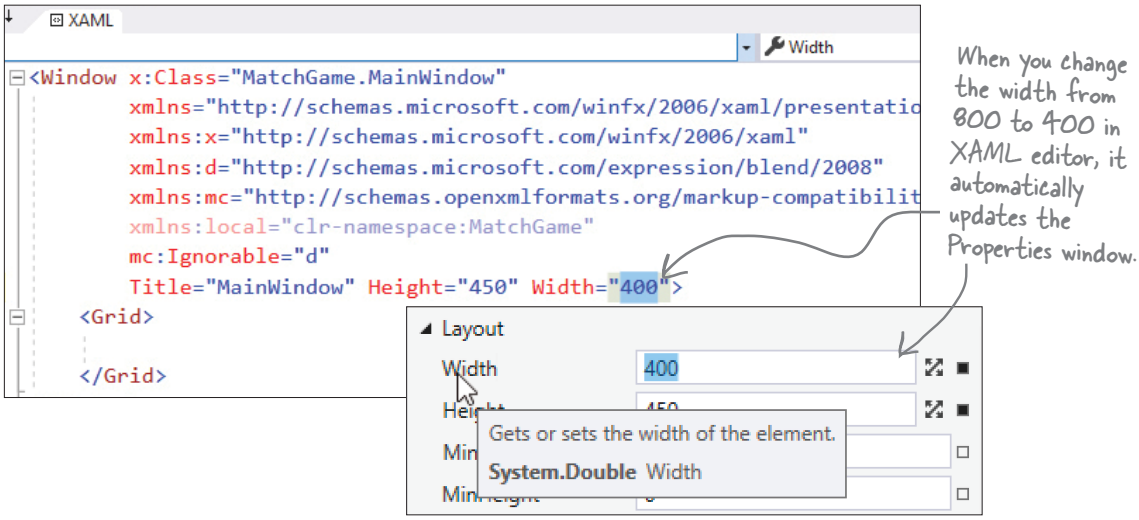
The designer shows you a preview of the window that you're editing. Any change you make here causes the XAML to be updated below.

You can make changes to the XAML here and immediately see the updates displayed in the window above.

2 **Change the size of the window.**

Move your mouse to the XAML editor and click anywhere in the first 8 lines of the XAML code. As soon as you do, you should see the window's properties displayed in the Properties window.

Expand the Layout section and **change the Width to 400**. The window in the Design pane will immediately get narrower. Look closely at the XAML code—the Width property is now 400.



3 **Change the window title.**

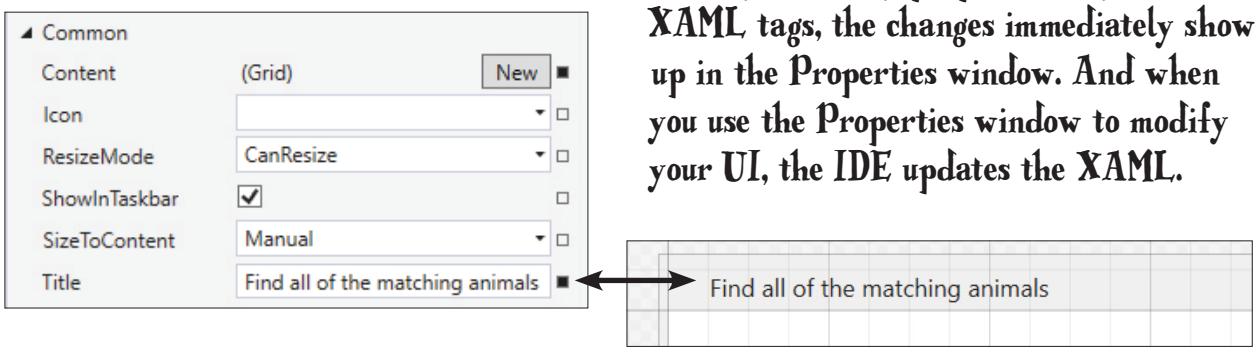
Find this line in the XAML code at the very end of the Window tag:

```
Title="MainWindow" Height="450" Width="400">
```

and change the title to **`Find all of the matching animals`** so it looks like this:

```
Title="Find all of the matching animals" Height="450" Width="400">
```

You'll see the change appear in the Common section in the Properties window—and, more importantly, the title bar of the window now shows the new text.



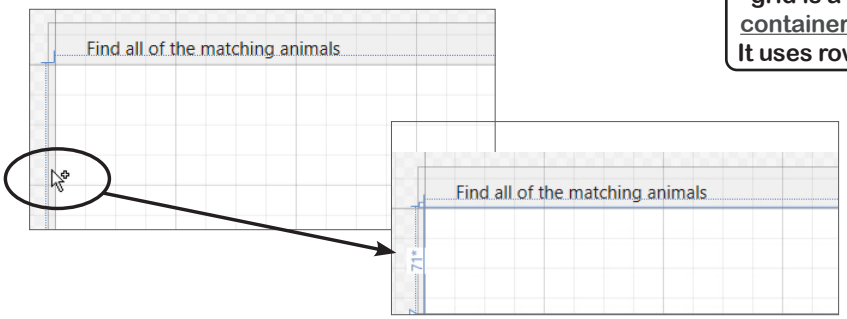
When you modify properties in your XAML tags, the changes immediately show up in the Properties window. And when you use the Properties window to modify your UI, the IDE updates the XAML.

Add rows and columns to the XAML grid

It might look like your main window is empty, but have a closer look at the bottom of the XAML. Notice how there's a line with `<Grid>` followed by one with `</Grid>`? Your window actually has a **grid**—you just don't see anything because it doesn't have any rows or columns. Let's go ahead and add a row.

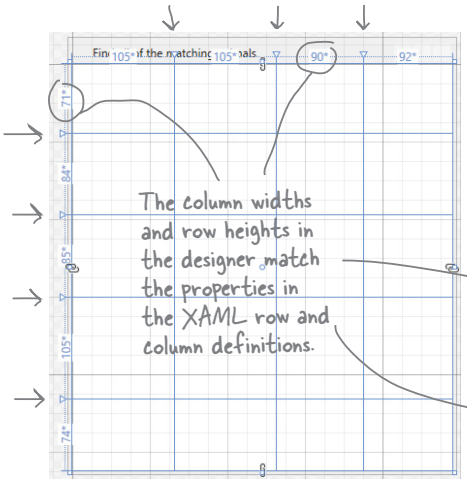
Move your mouse over the left side of the window in the designer. When a plus appears over the cursor, click the mouse to add a row.

Your WPF app's UI is built with controls like buttons, labels, and checkboxes. A grid is a special kind of control—called a container—that can contain other controls. It uses rows and columns to define a layout.



You'll see a number appear followed by an asterisk, and a horizontal line across the window. You just added a row to your grid!

Repeat four more times to add a total of five rows. Then hover over the top of the window and click to add four columns. Your window should look like the screenshot below (but your numbers will be different—that's okay). Now go back to the XAML. It now has a set of `ColumnDefinition` and `RowDefinition` tags that match the rows and columns that you added.




```
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="105*"/>
    <ColumnDefinition Width="105*"/>
    <ColumnDefinition Width="90*"/>
    <ColumnDefinition Width="92*"/>
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
    <RowDefinition Height="71*"/>
    <RowDefinition Height="84*"/>
    <RowDefinition Height="85*"/>
    <RowDefinition Height="105*"/>
    <RowDefinition Height="74*"/>
</Grid.RowDefinitions>
```

These "Watch it!" elements give you a heads-up about important, but often confusing, things that may trip you up or slow you down.

Watch it!

Things may look a bit different in your IDE.

*All of the screenshots in this book were taken with **Visual Studio 2019 Community Edition for Windows**. If you're using the Professional or Enterprise edition, you might see a few minor differences. But don't worry, everything will still work exactly the same.*

Make the rows and columns equal size

When your game displays the animals for the player to match, we want them to be evenly spaced. Each animal will be contained in a cell in the grid, and the grid will automatically adjust to the size of the window, so we need the rows and columns to all be the same size. Luckily, XAML makes it really easy for us to resize the rows and columns. **Click on the first RowDefinition tag in the XAML editor** to display its properties in the Properties window:

When this square is filled in, it means the property does not have the default value. Click the square and choose Reset from the menu to reset it to its default.

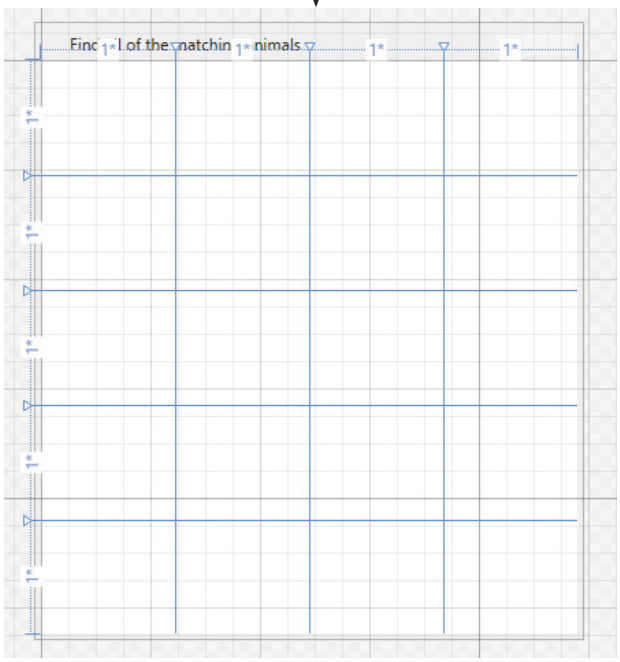
Click on the text here.

```
<Grid.RowDefinitions>
    <RowDefinition Height="71*"/>
    <RowDefinition Height="84*"/>
```

Go to the Properties window and **click the square** at the right of the Height property and **choose Reset from the menu** that pops up. Hey, wait a minute! As soon as you did that, the row disappeared from the designer. Well, actually, it didn't quite disappear—it just became very narrow. Go ahead and **reset the Height property** for all of the rows. Then **reset the Width property** for all of the columns. Your grid should now have four equally sized columns and five equally sized rows.

Try really reading the XAML. If you haven't worked with HTML or XML before, it might look like a jumble of <brackets> and /slashes at first. But the more you look at it, the more it will make sense to you!

Here's what you should see in the designer:



And here's what you should see in the XAML editor between the opening <Window ... > and closing </Window> tags:

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
</Grid>
```

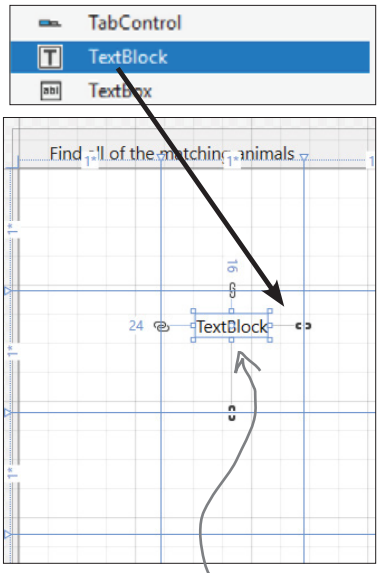
This is the XAML code to create a grid with four equal columns and five equal rows.

Add a TextBlock control to your Grid

WPF apps use **TextBlock controls** to display text, and we'll use them to display the animals to find and match. Let's add one to the window.

Expand the Common WPF Controls section in the Toolbox and **drag a TextBlock into the cell in the second column and second row**. The IDE will add a TextBlock tag between the `<Grid>` start and end tags:

```
<TextBlock x:Name="textBlock" Text="TextBlock"
    Grid.Column="1" Grid.Row="1"
    HorizontalAlignment="Left" VerticalAlignment="Top"
    Margin="24,16,0,0" TextWrapping="Wrap" />
```



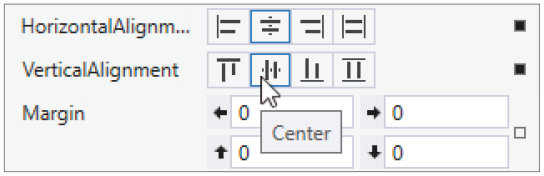
When you drag the control from the toolbox into a cell, the IDE adds a TextBlock to your XAML and sets its row, column, and margin.

Your properties may be in a different order, and the Margin property will have different numbers because it depends on where in the cell you dragged it.

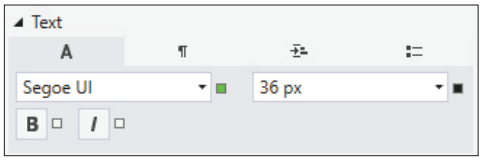
Let's start by **removing the name** from the control, since we won't be using it. Go to the Properties window—you'll see the name `textBlock` at the top:

Select the name and delete it—it **will be replaced with** `<No Name>`. Now check your XAML. The `x:Name="textBlock"` property should be gone.

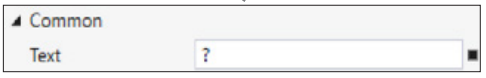
We want each animal to be centered. **Click on the label** in the designer, then go to the Properties window, click Center for the horizontal and vertical alignment properties, and then use the square to reset the margin property.



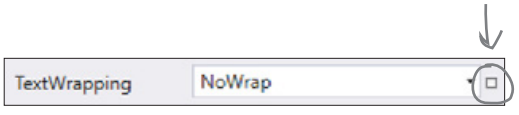
We also want the animals to be bigger, so **expand the Text section** in the Properties window and **change the font size** to **`36 px`**. Then go to the Common section and change the Text property to **`?`** to make it display a question mark.



The Text property (under Common) sets the TextBlock's text.



Click on the search box at the top of the Properties window, then type the word `wrap` to find properties that match. Use the square on the right of the window to reset the TextWrapping property.



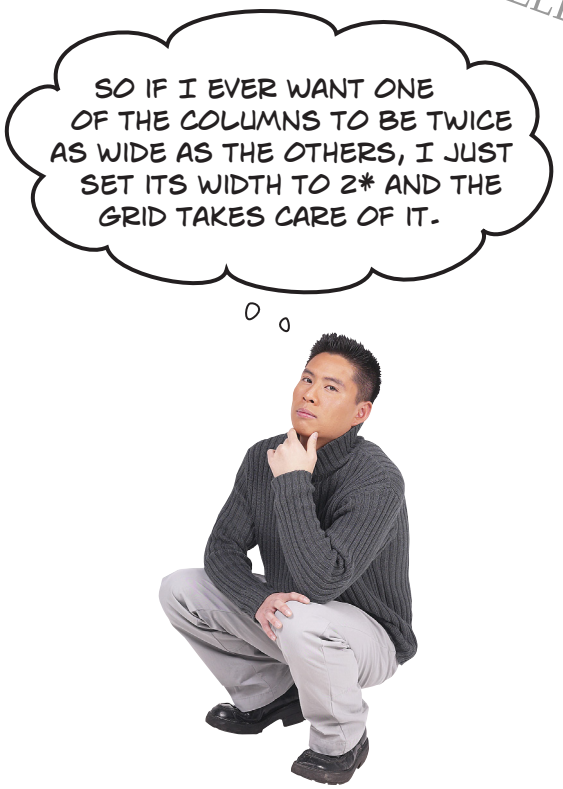
there are no Dumb Questions

Q: When I reset the height of the first four rows they disappeared, but then they all came back when I reset the height of the last one. Why did that happen?

A: The rows looked like they disappeared because by default WPF grids use **proportional sizing** for their rows and columns. If the height of the last row was 74*, then when you changed the first four rows to the default height of 1* that caused the grid to size the rows so that each of the first four rows take up one seventy-eighth (or 1.3%) of the height of the grid, and the last row takes up 74/78ths (or 94.8%) of the height, which made the first four rows look really tiny. As soon as you reset the last row to its default height of 1*, that caused the grid to resize each row to an even 20% of the height of the grid.

Q: When I set the width of the window to 400, what exactly is that measurement? How wide is 400?

A: WPF uses **device-independent pixels** that are always 1/96th of an inch. That means 96 pixels will always equal 1 inch on an *unscaled* display. But if you take out a ruler and measure your window, you might find that it's not exactly 400 pixels (or about 4.16 inches) wide. That's because Windows has really useful features that let you change how your screen is scaled, so your apps don't look teeny tiny if you're using a TV across the room as a computer monitor. Device-independent pixels help WPF make your app look good at any scale.



You'll see many exercises like this throughout the book. They give you a chance to work on your coding skills. And it's always okay to peek at the solution!

Exercise

You have one TextBlock—that's a great start! But we need 16 TextBlocks to show all of the animals to match. Can you figure out how to add more XAML to add an identical TextBlock to all of the cells in the first four rows of the grid?

Start by looking at the XAML tag that you just created. It should look like this—the properties may be in a different order, and we added a line break (which you can do too, if you want your XAML to be easier to read):

```
<TextBlock Text="?" Grid.Column="1" Grid.Row="1" FontSize="36"
       HorizontalAlignment="Center" VerticalAlignment="Center"/>
```

Your job is to **replicate that TextBlock** so each of the top sixteen cells in the grid contains an identical one—to complete this exercise, you'll need to *add fifteen more TextBlocks to your app*. A few of things to keep in mind:

- Rows and columns are numbered starting with 0, which is also the default value. So if you leave out the `Grid.Row` or `Grid.Column` property the TextBlock will appear in the leftmost row or top column.
- You can edit your UI in the designer, or copy and paste the XAML. Try both—see what works for you!

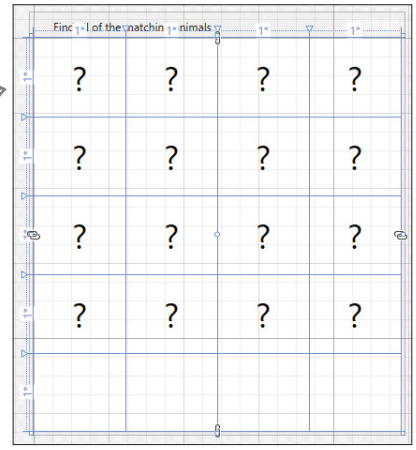
Exercise Solution

Here's the XAML for the 16 TextBlocks for the animals that the player matches—they're all identical except for their Grid.Row and Grid.Column properties, which place one TextBlock in each of the top 16 cells of the 5 row by 4 column grid. *(The <Window> tag stays the same, so we didn't include it here.)*

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>

  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition/>
    <RowDefinition/>
    <RowDefinition/>
    <RowDefinition/>
  </Grid.RowDefinitions>
```

This is what the column and row definitions look like once you make the rows and columns all equal size.

Here's what the window looks like in the Visual Studio designer once all of the TextBlocks are added.




```
  <TextBlock Text="?" FontSize="36" HorizontalAlignment="Center" VerticalAlignment="Center"/>
  <TextBlock Text="?" FontSize="36" HorizontalAlignment="Center" VerticalAlignment="Center" Grid.Column="1"/>
  <TextBlock Text="?" FontSize="36" HorizontalAlignment="Center" VerticalAlignment="Center" Grid.Column="2"/>
  <TextBlock Text="?" FontSize="36" HorizontalAlignment="Center" VerticalAlignment="Center" Grid.Column="3"/>

  <TextBlock Text="?" FontSize="36" HorizontalAlignment="Center" VerticalAlignment="Center" Grid.Row="1"/>
  <TextBlock Text="?" FontSize="36" Grid.Row="1" Grid.Column="1"
        HorizontalAlignment="Center" VerticalAlignment="Center"/>
  <TextBlock Text="?" FontSize="36" Grid.Row="1" Grid.Column="2"
        HorizontalAlignment="Center" VerticalAlignment="Center"/>
  <TextBlock Text="?" FontSize="36" Grid.Row="1" Grid.Column="3"
        HorizontalAlignment="Center" VerticalAlignment="Center"/>
```

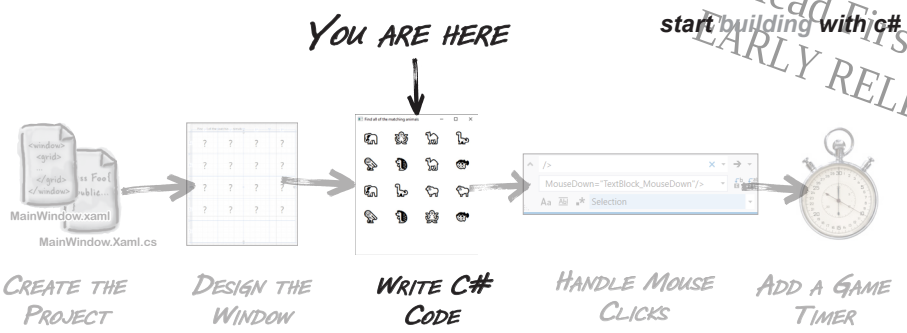
These four TextBlock controls all have their Grid.Row property set to 1, so they're in the second row from the top (because the first row is 0).

```
  <TextBlock Text="?" FontSize="36" Grid.Row="2" HorizontalAlignment="Center" VerticalAlignment="Center"/>
  <TextBlock Text="?" FontSize="36" Grid.Row="2" Grid.Column="1"
        HorizontalAlignment="Center" VerticalAlignment="Center"/>
  <TextBlock Text="?" FontSize="36" Grid.Row="2" Grid.Column="2"
        HorizontalAlignment="Center" VerticalAlignment="Center"/>
  <TextBlock Text="?" FontSize="36" Grid.Row="2" Grid.Column="3"
        HorizontalAlignment="Center" VerticalAlignment="Center"/>
```

It's okay if you included Grid.Row or Grid.Column properties with a value of 0. We left them out because 0 is the default value.

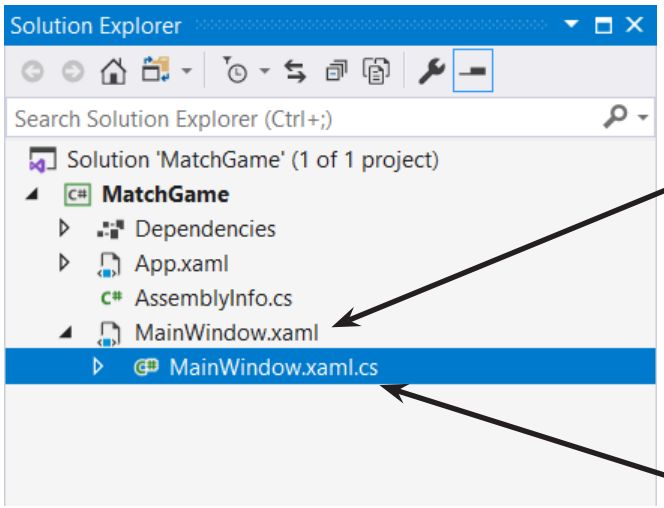
```
  <TextBlock Text="?" FontSize="36" Grid.Row="3" HorizontalAlignment="Center" VerticalAlignment="Center"/>
  <TextBlock Text="?" FontSize="36" Grid.Row="3" Grid.Column="1"
        HorizontalAlignment="Center" VerticalAlignment="Center"/>
  <TextBlock Text="?" FontSize="36" Grid.Row="3" Grid.Column="2"
        HorizontalAlignment="Center" VerticalAlignment="Center"/>
  <TextBlock Text="?" FontSize="36" Grid.Row="3" Grid.Column="3"
        HorizontalAlignment="Center" VerticalAlignment="Center"/>
</Grid>
```

This isn't as complex as it looks! It's really just the same line repeated 16 times with small variations. Every line that starts with <TextBlock has the same four properties (Text, FontSize, HorizontalAlignment, and VerticalAlignment). They just have different Grid.Row and Grid.Column properties. (The properties can be in any order.)



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

You've finished designing the main window—or at least enough of it to get the next part of your game working. Now it's time to add C# code to make your game work.



You've been editing the XAML code in MainWindow.xaml. That's where all of the design elements of the window live—the XAML in this file defines the appearance and layout of the window.

Now you'll start working on the C# code, which will be in MainWindow.xaml.cs. This is called the code-behind for the window because it's joined with the markup in the XAML file. That's why it has the same name, except with the additional ".cs" at the end. You'll add C# code to this file that defines the behavior of the game, including code to add the emoji to the grid, handle mouse clicks, and make the countdown timer work.

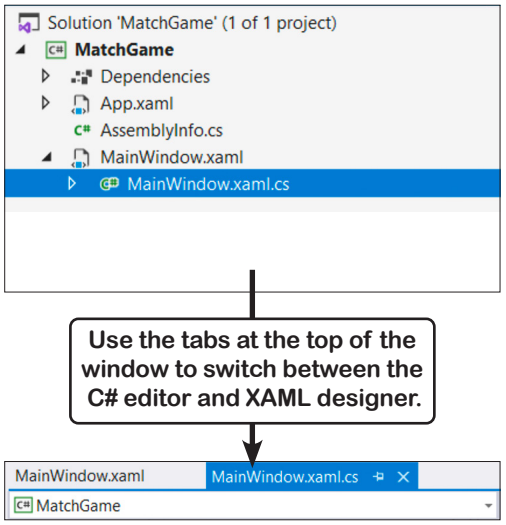
When you enter C# code, it has to be exactly right.

*Some people say that you truly become a developer after the first time you've spent hours tracking down a misplaced period. Case matters: `SetUpGame` is different from `setUpGame`. 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** can help you avoid those problems... but it can't do everything for you.*

Generate a method to set up the game

Generate this!

Now that the user interface is set up, it's time to start writing code for the game. You're going to do that by **generating a method** (just like the Main method you saw earlier), and then adding code to it.



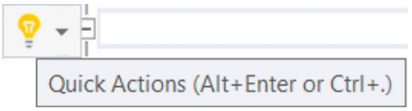
1 **Open MainWindow.xaml.cs in the editor.**
Click the triangle ▷ next to MainWindow.xaml in the Solution Explorer and **double-click on MainWindow.xaml.cs** to open it in the IDE's code editor. You'll notice that there's already code in that file. Visual Studio will help you add a method to it.

2 **Generate a method called SetupGame.**
Find this part of the code that you opened:

```
public MainWindow();
{
    InitializeComponent();
}
```

Click at the end of the `InitializeComponent();` line to put your mouse cursor just to the right of the semicolon. Press enter two times, then type: `SetUpGame();`

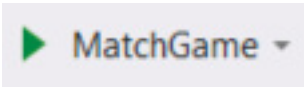
As soon as you type the semicolon, a red squiggly line will appear underneath `SetUpGame`. Click on the word `SetUpGame` – you should see a light bulb icon at the left side of the window. Click on it to **open the Quick Actions menu** and use it to generate a method.



Any time you see the light bulb icon, it's telling you that you've selected code that has a quick action available, which means there's a task that Visual Studio can automate for you. You can either click the light bulb or press Alt+Enter or Ctrl+. (period) to see the available quick actions.

3 Try running your code.

Click the button at the top of the IDE to start your program, just like you did with your console app earlier.



The "Start Debugging" button in the toolbar at the top of the IDE starts your app. You can also use Start Debugging (F5) from the Debug menu to start your app.

Uh-oh—something went wrong. Instead of showing you a window, it **threw an exception**:

```
21  public partial class MainWindow : Window
22  {
        0 references
23      public MainWindow()
24      {
25          InitializeComponent();
26          SetUpGame();
27      }
28
        1 reference
29      private void SetUpGame()
30      {
31          throw new NotImplementedException();
32      }
33  }
34  }
35
```

Exception User-Unhandled

System.NotImplementedException: 'The method or operation is not implemented.'

View Details | Copy Details | Start Live Share session...

Exception Settings

Things may seem like they're broken, but this is actually exactly what we expected to happen! The IDE paused your program, and highlighted the most recent line of code that ran. Take a closer look at it:

```
throw new NotImplementedException();
```

The method that the IDE generated literally told C# to throw an exception. And take a closer look at the message that came with the exception:

System.NotImplementedException: 'The method or operation is not implemented.'

That actually makes sense, because **it's up to you to implement the method** that the IDE generated. If you forget to implement it, the exception is a nice reminder that you still have work to do. And if you generate a lot of methods, it's great to have that as a reminder!

Click the square Stop Debugging button  in the toolbar (or choose Stop Debugging (F5) from the Debug menu) to stop your program so you can finish implementing the SetUpGame method.

When you're using the IDE to run your app, the Stop Debugging button immediately quits it.

Finish your SetUpGame method

This is a special method called a constructor, and you'll learn more about how it works in Chapter 5.

You put your SetUpGame method inside the `public MainWindow()` method because everything inside that method is called as soon as your app starts.

You'll learn a lot more about methods soon.

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

1 **Start adding working code to your SetUpGame method.**

Your SetUpGame method will take eight pairs of animal emoji characters and randomly assign them to the TextBlock controls so the player can match them. So the first thing your method needs is a list of those emoji, and the IDE will help us write code for it. Select the throw statement that the IDE added, and delete it. Then put your cursor where that statement was and type `List`. The IDE will pop up an **IntelliSense window** with a bunch of keywords that start with "List".

```
private void SetUpGame()
{
    List
}
```

JournalEntry**List**Converter
Linked**List**<>
Linked**List**Node<>
List

Choose `List` from the IntelliSense popup. Then type `<str` – another IntelliSense window will pop up with matching keywords:

```
private void SetUpGame()
{
    List<str
}
```

List<**T**>
Represents a strongly typed list of objects that can be accessed by index. Provides methods to search, sort, and manipulate lists.
T: *The type of elements in the list.*

Stretch
StretchDirection
String
string — string Keyword

Choose `string`. Finish typing this line of code, but **don't hit enter yet**:

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

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.

② Add values to your List.

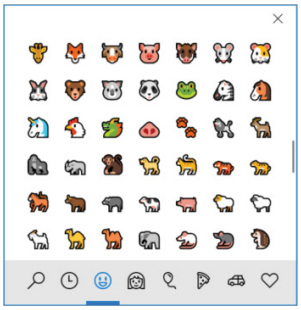
Your C# statement isn't done yet. Make sure your cursor is placed just after the) at the end of the line, then type an opening curly bracket { – the IDE will add the closing one for you, and your cursor will be positioned between the two brackets. **Press enter**—the IDE will add line breaks for you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

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

}
```

Use the Windows emoji panel (press Windows logo key + period) or go to your favorite emoji website (for example, *https://emojipedia.org/nature/*) and copy a single emoji character. Go back to your code, add a " then paste the character, followed by another " and a comma, space, another ", the same character in again, and one last " and comma. Then do the same thing for seven more emoji so you end up with **eight pairs of animal emoji between the brackets**. Add a ; after the closing curly bracket:

```
List<string> animalEmoji = new List<string>()
{
    "🐶","🐶",
    "🐵","🐵",
    "🐘","🐘",
    "🦛","🦛",
    "🐫","🐫",
    "🦒","🦒",
    "🦓","🦓",
    "🦔","🦔",
};
```

Hover over the dots under animalEmoji – the IDE will tell you that the value assigned to it is never used. That warning will go away as soon as you use the list of emoji in the rest of the method.



The emoji panel is built into Windows 10. Just press Windows logo key + period to bring it up.

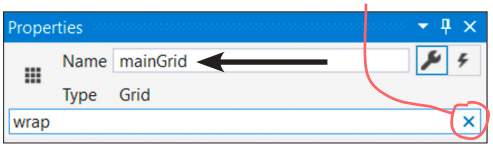
③ Finish your method.

Now **add the rest of the code** for the method—be *careful* with the periods, parentheses, and brackets:

```
Random random = new Random();
foreach (TextBlock textBlock in mainGrid.Children.OfType<TextBlock>())
{
    int index = random.Next(animalEmoji.Count);
    string nextEmoji = animalEmoji[index];
    textBlock.Text = nextEmoji;
    animalEmoji.RemoveAt(index);
}
```

This line goes right after the closing bracket and semicolon.

Not seeing properties? You may still have the word "wrap" typed into the search box.



The red squiggly line under `mainGrid` is the IDE telling you there's an error: your program won't build because there's nothing with that name anywhere in the code. **Go back to the XAML editor** and click on the `<Grid>` tag, then go to the Properties window and enter `mainGrid` in the Name box.

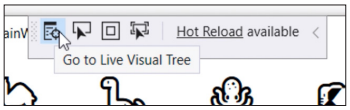
Check the XAML—you'll see `<Grid x:Name="mainGrid">` at the top of the grid. And now there shouldn't be any errors in your code. If there are, ***carefully check every line***—it's easy to miss something.

If you get an exception when you run your game, make sure you have exactly 8 pairs of emoji in your animalEmoji list and 16 <TextBlock ... /> tags in your XAML.

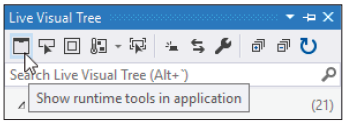
Run your program

Click the Start button in the IDE's toolbar to start your program running. A window will pop up with your eight pairs of animals in random positions:

When your program first runs, you'll see the runtime tools 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 runtime tools.

The IDE goes into debugging mode while your program is running: the Start button is replaced by a grayed-out Continue, and **debug controls** appear in the toolbar with buttons to pause, stop, and restart your program.

Stop your program by clicking X in the upper right corner of the window or the square stop button in the debug controls. Run it a few times—the animals will get shuffled each time.

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. And a really effective way to run a project is to build it in small increments, taking stock along the way to make sure things are going in a good direction. That way you have plenty of opportunities to change course.

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

WHO DOES WHAT?

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

C# statement

```
List<string> animalEmoji = new List<string>()
{
    "🐙","🐙",
    "🐵","🐵",
    "🐘","🐘",
    "🐏","🐏",
    "🐂","🐂",
    "🦒","🦒",
    "🐿","🐿",
    "🐊","🐊",
};
```

```
Random random = new Random();
```

```
foreach (TextBlock textBlock in
    mainGrid.Children.OfType<TextBlock>())
```

```
int index = random.Next(animalEmoji.Count);
```

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

```
textBlock.Text = nextEmoji;
```

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

What it does

Update the TextBlock with the random emoji from the list

Find every TextBlock in the main grid and repeat the following statements for each of them

Remove the random emoji from the list

Create a list of eight pairs of emoji

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

Create a new random number generator

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

WHO DOES WHAT? solution

C# statement | **What it does**

```
List<string> animalEmoji = new List<string>()
{
    "🐶","🐶",
    "🐷","🐷",
    "🐮","🐮",
    "🐵","🐵",
    "🐯","🐯",
    "🦒","🦒",
    "🐰","🐰",
    "🐘","🐘",
};

Random random = new Random();

foreach (TextBlock textBlock in
    mainGrid.Children.OfType<TextBlock>())

int index = random.Next(animalEmoji.Count);

string nextEmoji = animalEmoji[index];

textBlock.Text = nextEmoji;

animalEmoji.RemoveAt(index);
```

Update the TextBlock with the random emoji from the list

Find every TextBlock in the main grid and repeat the following statements for each of them

Remove the random emoji from the list

Create a list of eight pairs of emoji

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

Create a new random number generator

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

MINI Sharpen your pencil

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

1. Take a piece of paper and turn it on its side so it's in landscape orientation, and draw a vertical line down the middle.
2. Write out 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 we'll learn about *refactoring*, or programming techniques that are all about improving your code after you've written it.

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

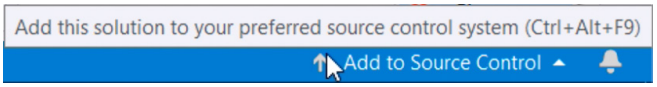
BULLET POINTS

- Visual Studio is **Microsoft's IDE**—or **integrated development environment**—that simplifies and assists in editing and managing your C# code files.
- **.NET Core Console Apps** are cross-platform apps that use text for input and output.
- The IDE's **AI-assisted IntelliSense** helps you enter code more quickly.
- **WPF** (or Windows Presentation Foundation) is a technology that you can use to build visual apps in C#.
- WPF 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.
- The **Grid XAML control** provides a grid layout that holds other controls.
- The **TextBlock XAML tag** adds a control for holding text.
- The IDE's **Properties window** makes it easy to edit the properties of your controls—like changing their layout, text, or what row or column of the grid they're in.

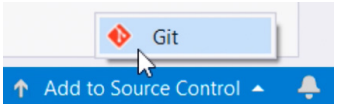
Add your new project to source control

You're going to be building a lot of different projects in this book. Wouldn't it be great if there was an easy way to back them up and access them from anywhere? What if you make a mistake—wouldn't it be great to roll back to a previous version of your code? Well, you're in luck! Because that's exactly what **source control** does: it gives you an easy way to back up all of your code, and keeps track of every change that you make. And Visual Studio makes it really easy for you to add your projects to source control.

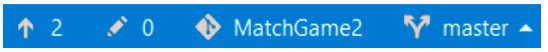
Find **Add to Source Control** in the status bar at the bottom of the IDE.



Click on it—Visual Studio will prompt you to add your code to Git.

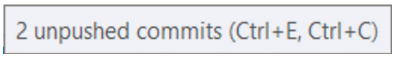


Click Git. Visual Studio may prompt you for your name and email address. Then it should now show you this in the status bar:



Your code is now under source control. Now hover your mouse over 

:



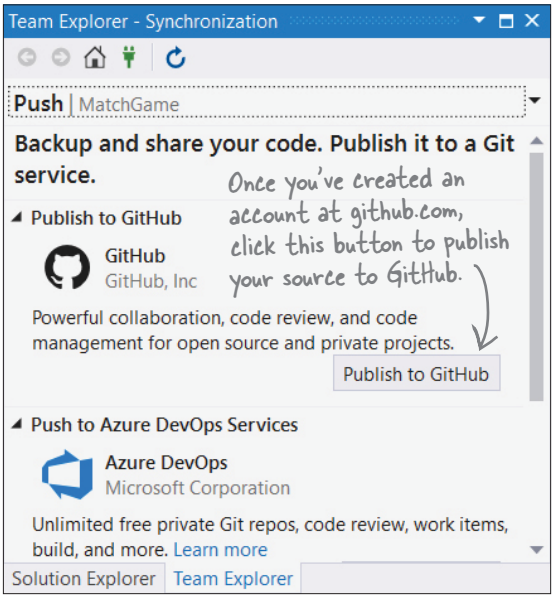
The IDE is telling you that you have two **commits**—or saved versions of your code—that haven't been ***pushed*** to a location that's outside of your computer.

When you added your project to source control, the IDE opened the **Team Explorer window** in the same panel as the Solution Explorer. (If you don't see it, select it from the View menu.)

The Team Explorer helps you manage your source control. You'll use it to publish your project to a **remote repository**—sometimes called a "**repo**"— and when you have local changes you'll use the Team Explorer to push them to the remote repo.

Visual Studio will publish your source to any Git repository, but the easiest one to use is **GitHub**, a Git provider that's owned by Microsoft.

If you don't have a GitHub account, go to https://github.com and create one. Then **click the Publish to GitHub button** in the Team Explorer window.

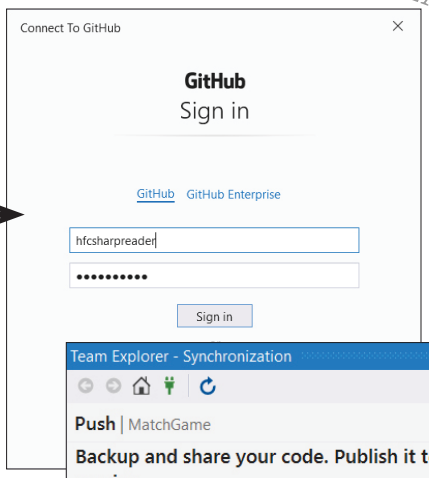


Adding your project to source control is optional.

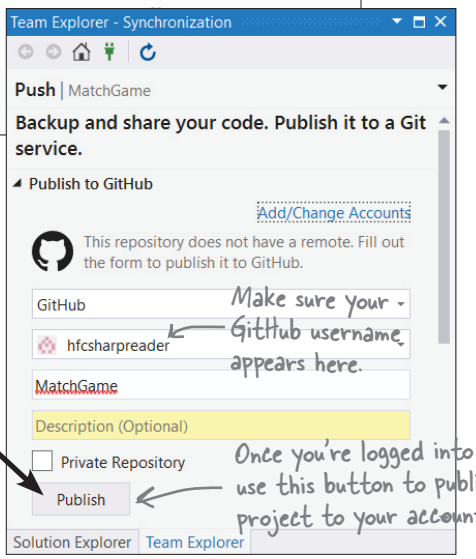
Maybe you're working on a computer on an office network that doesn't allow access to GitHub. Maybe you just don't feel like doing it. Whatever the reason, you can skip this step—or, alternately, you can also publish it to a private repository if you want to keep a backup but don't want other people to find it.

As soon as you add your code to Git, the status bar changes to show you that the code in the project is now under source control. Git is the most popular system for source control, and Visual Studio includes a full-featured Git client. Your project folder now has a hidden folder called `.git` that Git uses to keep track of every revision that you make to your code.

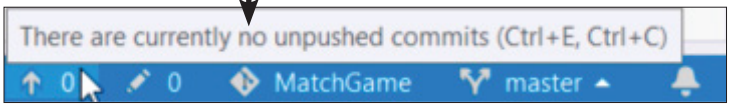
When you press the Publish to GitHub button, Visual Studio will display a **GitHub sign in form**. Enter your GitHub username and password. (If you've set up two-factor authentication, you'll also be asked to use it.)



After the IDE logs into GitHub it will display a "Publish to GitHub" form, where you specify the Git provider, user, project name, and indicate whether it should be a private repository. You can keep all of the default options. Press the **Publish button** to publish to GitHub.

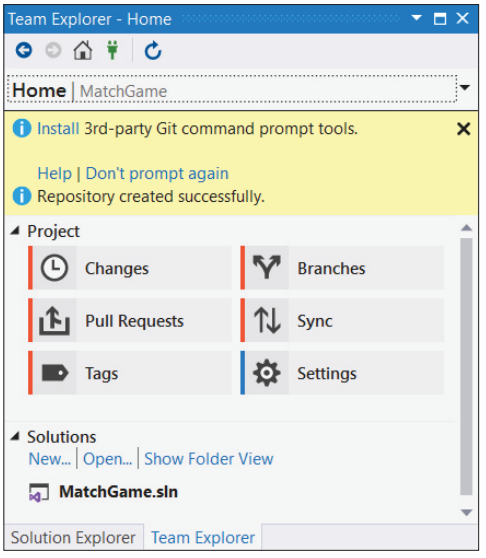


Once you publish to GitHub, the Git status in the status bar will update to show you that there are no more unpushed commits. That means your project is now in sync with a repository in your GitHub account.



Git has powerful command prompt tools, and Visual Studio can install them for you automatically. Visual Studio makes it very easy to work with Git, so feel free to install these tools, but they're not required.

Once you've published your code to GitHub, you can use the Team Explorer to work with your Git repo.



Go to https://github.com/your-github-username/MatchGame to see the code that you just pushed. When you sync your project to the remote, you'll see updates in the "Commits" section.

there are no Dumb Questions

Q: Is XAML really code?

A: .Yes, absolutely. Remember how the red squiggly lines appeared underneath `mainGrid` in your C# code, and only disappeared when you added the name to the `<Grid>` tag in the XAML? That's because you were actually modifying the code—once you added the name in the XAML, your C# code was able to use it.

Q: I assumed XAML was like HTML, which is interpreted by a browser. XAML isn't like that?

A: No, XAML is actually code that's built alongside your C# code. Back at the beginning of the chapter we talked about how you could use the partial keyword to split up a class into multiple files. That's exactly how XAML and C# are joined up: the XAML defines a user interface, the C# defines the behavior, and they're joined up using partial classes.

That's why it's important to think of your XAML as code, and why learning XAML is an important skill for any C# developer.

Q: I noticed a LOT of `using` lines at the top of my C# file. Why so many?

A: WPF apps tend to use code from a lot of different namespaces, so when Visual Studio creates a WPF project for you, it automatically adds **`using` directives** for the most common ones at the top of the MainWindow.xaml.cs file. In fact, you're using some of them already: the IDE uses a lighter text color to show you namespaces that aren't in use in the code. Your C# code is using classes from five different namespaces.

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

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

IDE Tip: The Error List

Look at the bottom of the code editor—notice how it says ✓ No issues found. That means your code **builds**, which is what the IDE does to turn your code into a **binary** that the operating system can run. Let's break your code.

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

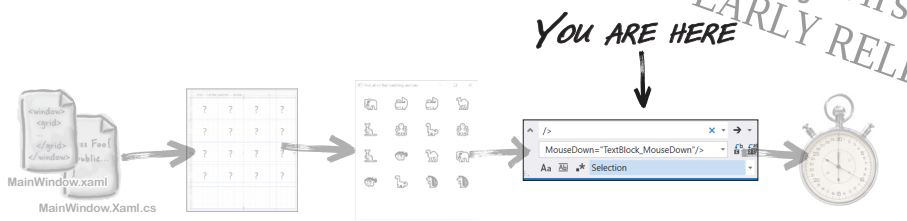
Check the bottom of the code editor again—now it says ✗ 3. If you don't have the Error List window open, choose Error List from the View menu to open it. You'll see three errors in the Error List:

Error List

Entire Solution | ✗ 3 Errors | ⚠ 0 Warnings | ⓘ 0 Messages | Build + IntelliSense | Search Error List

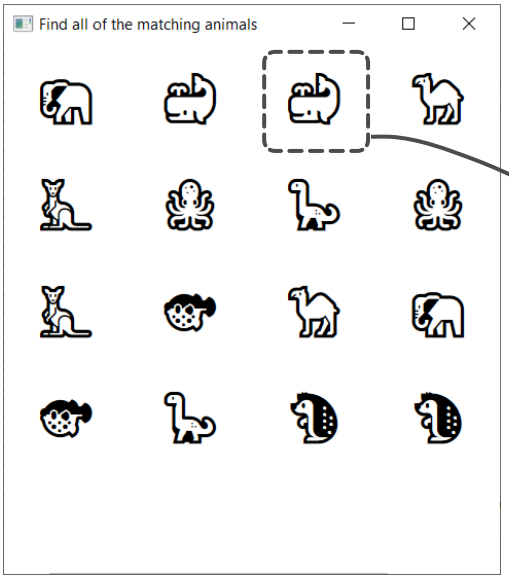
	Code	Description	Project	File	Line	Suppression State
✗	CS1001	Identifier expected	MatchGame	MainWindow.xaml.cs	32	Active
✗	CS1002	; expected	MatchGame	MainWindow.xaml.cs	32	Active
✗	CS0246	The type or namespace name 'Xyz' could not be found (are you missing a using directive or an assembly reference?)	MatchGame	MainWindow.xaml.cs	32	Active

The IDE displayed these errors because `Xyz` is not valid C# code, and these prevent the IDE from building your code. As long as there are errors in your code it won't run, so go ahead and delete the `Xyz` line that you added.



The next step to build the game is handling mouse clicks

Now that the game is displaying the animals for the player to click on, we need to add code that makes the gameplay work. The player will click on animals in pairs. When they click on the first animal, it disappears. Then they click on a second animal—if it matches, that one disappears too, but if it doesn't the first animal reappears. We'll make this work by adding an **event handler**, which is just a name for a method that gets called when certain actions (like mouse clicks, double-clicks, windows getting resized, etc.) happen in the app.



When the player clicks on one of the animals, the app will call a method called TextBlock_MouseDown that handles mouse clicks. Here's what that method will do.

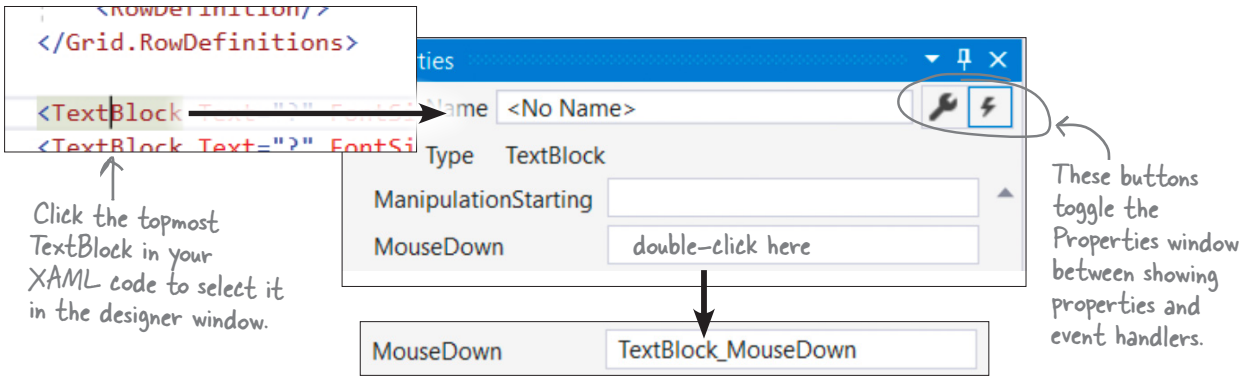
```
TextBlock_MouseDown() {

  /* If it's the first in the
   * pair being clicked, keep
   * track of which TextBlock
   * was clicked and make the
   * animal disappear. If
   * it's the second one,
   * either make it disappear
   * (if it's a match) or
   * bring back the first one
   * (if it's not).
   */
}
```

This is a comment. Everything between /* and */ is ignored by C#. We added this comment to tell you what your TextBlock_MouseDown method will do – and also to show you what a comment looks like.

Make your TextBlocks respond to mouse clicks

Your SetUpGame method changes the TextBlocks to show animal emoji, so you've seen how your code can modify controls in your application. Now we need to write code that goes in the other direction: your controls need to call your code—and the IDE can help.

Go back to the XAML editor window and **click first `TextBlock` tag**—this will cause the IDE to select it in the designer so you can edit its properties. Then go to the Properties window and click the event handlers button ( ). An **event handler** is a method that your application calls when a specific event happens. These events include keyboard presses, drag and drop, window resizing, and of course, mouse movement and clicks. Scroll down the Properties window and look through the names of the different events your TextBlock can add event handlers for. **Double-click inside the box to the right of the event called MouseDown.**



The IDE filled in the MouseDown box with a method name, TextBlock_MouseDown, and the XAML for the TextBlock now has a MouseDown property:

```
<TextBlock Text="?" FontSize="36" HorizontalAlignment="Center"
      VerticalAlignment="Center" MouseDown="TextBlock_MouseDown"/>
```

But you probably didn't notice that, because the IDE also **added a new method** to the code-behind—the code that's joined with the XAML—and switched to the C# editor to display it. You can always jump right back to it from the XAML editor by right-clicking on TextBlock_MouseDown in the XAML editor and choosing View Code. Here's the method it added:

```
private void TextBlock_MouseDown(object sender, MouseButtonEventArgs e)
{

}
```

Whenever the player clicks on the TextBlock, the app will automatically call the TextBlock_MouseDown method. So now we just need to add code to it. And then we'll need to hook up all of the other TextBlocks so they call it, too.

An event handler is a method that your app calls in response to an event like a mouse click, key press, or window resize.

Sharpen your pencil

Here's the code for the TextBlock_MouseDown method. Before you add this code to your program, read through it and try to figure out what it does. 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.

```
TextBlock lastTextBlockClikcked;
bool findingMatch = false;

private void TextBlock_MouseDown(object sender, MouseButtonEventArgs e)
{
    TextBlock textBlock = sender as TextBlock;
    if (findingMatch == false)
    {
        textBlock.Visibility = Visibility.Hidden;
        lastTextBlockClikcked = textBlock;
        findingMatch = true;
    }
    else if (textBlock.Text == lastTextBlockClikcked.Text)
    {
        textBlock.Visibility = Visibility.Hidden;
        findingMatch = false;
    }
    else
    {
        lastTextBlockClikcked.Visibility = Visibility.Visible;
        findingMatch = false;
    }
}
```

1. What does *findingMatch* do?

2. What does the block of code starting with *if (findingMatch == false)* do?

3. What does the block of code starting with *else if (textBlock.Text == lastTextBlockClikcked.Text)* do?

4. What does the block of code starting with *else* do?

Sharpen your pencil Solution

Here's the code for the TextBlock_MouseDown method. Before you add this code to your program, read through it and try to figure out what it does. 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.

```
TextBlock lastTextBlockClikcked;
bool findingMatch = false;

private void TextBlock_MouseDown(object sender, MouseButtonEventArgs e)
{
    TextBlock textBlock = sender as TextBlock;
    if (findingMatch == false)
    {
        textBlock.Visibility = Visibility.Hidden;
        lastTextBlockClikcked = textBlock;
        findingMatch = true;
    }
    else if (textBlock.Text == lastTextBlockClikcked.Text)
    {
        textBlock.Visibility = Visibility.Hidden;
        findingMatch = false;
    }
    else
    {
        lastTextBlockClikcked.Visibility = Visibility.Visible;
        findingMatch = false;
    }
}
```

Here's what all of the code in the TextBlock_MouseDown method does. Reading code in a new programming language is a lot like reading sheet music—it's a skill that takes practice, and the more you do it the better you get at it.

1. What does *findingMatch* do?

It keeps track of whether or not the player just clicked on the first animal in a pair and is now trying to find its match.

2. What does the block of code starting with *if (findingMatch == false)* do?

The player just clicked the first animal in a pair, so it makes that animal invisible and keeps track of its TextBlock in case it needs to make it visible again.

3. What does the block of code starting with *else if (textBlock.Text == lastTextBlockClikcked.Text)* do?

The player found a match! So it makes the second animal in the pair invisible (and unclickable) too, and resets findingMatch so the next animal clicked on is the first one in a pair again.

4. What the does block of code starting with *else* do?

The player clicked on an animal that doesn't match, so it makes the first animal that was clicked visible again and resets findingMatch.

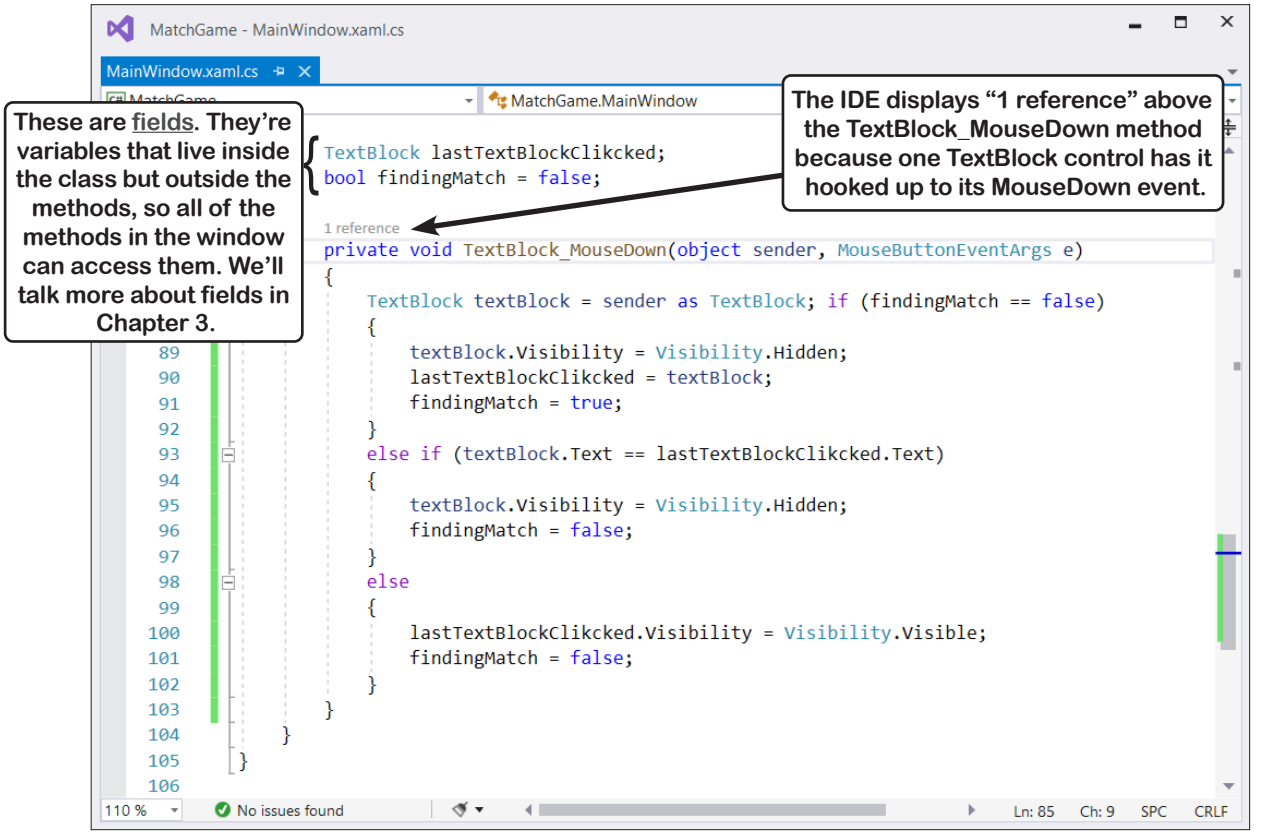
Add the TextBlock_MouseDown code

Now that you've read through the code for TextBlock_MouseDown, it's time to add it to your program. Here's what you'll do next:

1. Add the first two lines with `lastTextBlockClicked` and `findingMatch` **above the first line** of the TextBlock_MouseDown method that the IDE added for you. Make sure you put them between the closing curly bracket at the end of SetUpGame and the new code the IDE just added.

2. **Fill in the code** for TextBlock_MouseDown. Be really careful about equals signs – there's a big difference between = and == (which you'll learn about in the next chapter).

Here's what it looks like in the IDE:

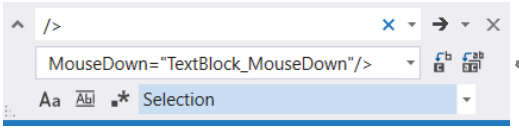


Make the rest of the TextBlocks call the same MouseDown event handler

Right now only the first TextBlock has an event handler hooked up to its MouseDown event. Let's hook up the other 15 TextBlocks to it, too. You *could* do it by selecting each one in the designer and entering TextBlock_MouseDown into the box next to MouseDown. But we already know that just adds a property to the XAML code, so let's take a shortcut.

1. **Select the last 15 TextBlocks in the XAML editor.**
Go to the XAML editor, click to the left of the second `TextBlock` tag, and drag down to the end of the TextBlocks, just above the closing `</Grid>` tag. You should now have the last 15 TextBlocks selected (but not the first one).

2. **Use Quick Replace to add MouseDown event handlers.**
Choose **Find and Replace >> Quick Replace** from the Edit menu. Search for `/>` and replace it with `MouseDown="TextBlock_MouseDown"/>` – make sure that there's a space before `MouseDown` and that the search range is ***Selection*** so it only adds the property to the selected TextBlocks.



There's a space in front of MouseDown so it doesn't run into the previous property.

3. **Run the replace over all 15 selected TextBlocks.**
Click the Replace All button ( ) to add the MouseDown property to the TextBlocks—it should tell you that 15 occurrences were replaced. Carefully examine the XAML code to make sure they each have a MouseDown property that exactly matches the one in the first TextBlock.

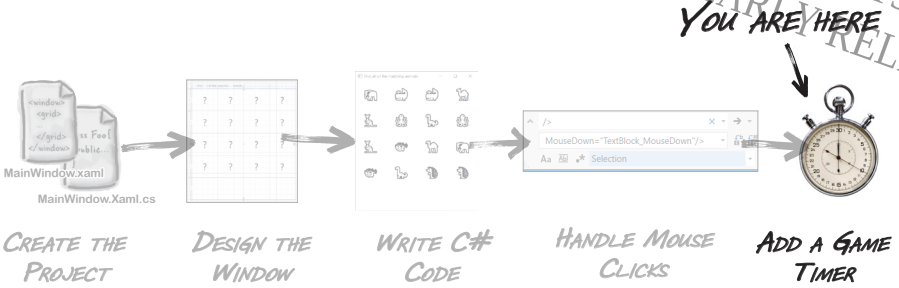
   Make sure that the method now shows **16 References** in the C# editor (choose Build Solution from the Build menu to update it). If you see 17 references, you accidentally attached the event handler to the Grid. You definitely don't want that—if you do, you'll get an exception when you click an animal.

Run your program. Now you can click on pairs of animals to make them disappear. The first animal you click will disappear. If you click on its match, that one disappears, too. If you click on an animal that doesn't match, the first one will appear again. When all the animals are gone, restart or close the program.

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

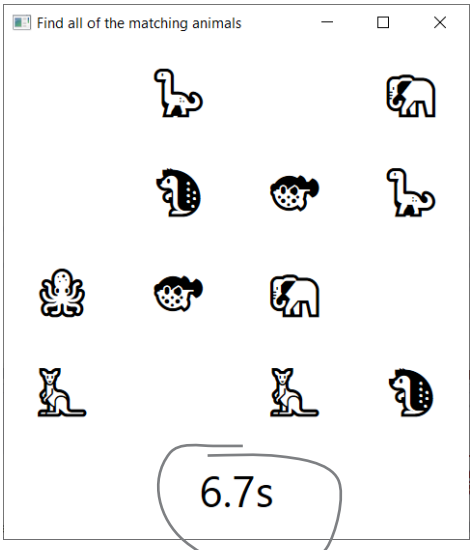
BRAIN POWER

You've reached another checkpoint in your project! Your game might be pretty simple so far, but it works and it's playable, so this is a great time to step back and think about how you could make it better. What could you change to make it more interesting?



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

Add this!

Let's dive right in and add that timer.

1. Start by finding the namespace keyword near the top of MainWindow.xaml.cs and add the `using System.Windows.Threading;` directly underneath it:

```
namespace MatchGame
{
    using System.Windows.Threading;
```

2. Find *`public partial class MainWindow`* and **add this code** just after the opening curly bracket **`{`**:

```
public partial class MainWindow : Window
{
    DispatcherTimer timer = new DispatcherTimer();
    int tenthsOfSecondsElapsed;
    int matchesFound;
```

You'll add these three lines of code to create a new timer and add two fields to keep track of the time elapsed and number of matches the player has found.

3. We need to tell our timer how frequently to "tick" and what method to call. Put your mouse cursor at the beginning of the line where you call the SetUpGame method. Press enter, then type the two lines of code in the screenshot below that start with **`timer.`** – as soon as you type **`+=`** the IDE will display a message:

```
0 references
public MainWindow()
{
    InitializeComponent();

    timer.Interval = TimeSpan.FromSeconds(.1);
    timer.Tick +=
    SetUpGame();
}
```

Timer_Tick; (Press TAB to insert)

Next, add these two lines of code. Start typing the second line: "timer.Tick +="

As soon as you add the equals sign, the IDE will display this "Press TAB to insert" message.

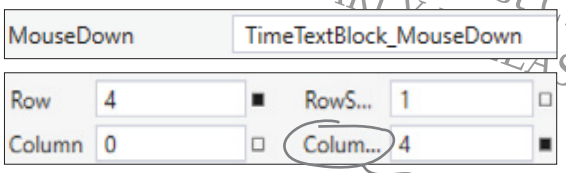
4. Press the tab key. The IDE will finish the line of code and add a Timer_Tick method:

```
0 references
public MainWindow()
{
    InitializeComponent();

    timer.Interval = TimeSpan.FromSeconds(.1);
    timer.Tick += Timer_Tick;
    SetUpGame();
}

1 reference
private void Timer_Tick(object sender, EventArgs e)
{
    throw new NotImplementedException();
}
```

When you press the TAB key, the IDE automatically inserts a method for the timer to call.

5 The Timer_Tick method will update a TextBlock that spans the entire bottom row of the grid. Here's how to set it up:

- Drag a TextBlock into the lower left square and give it the name `timeTextBlock`
- Reset its margins, center it in the cell, and set the FontSize to 36px and Text to "Elapsed time"
- Find the ColumnSpan property and set it to 4
- Add a MouseDown event handler called TimeTextBlock_MouseDown



ColumnSpan is in the Layout section in the Properties window. Use the buttons at the top of the window to switch between properties and events.

Here's what the XAML will look like:

```
<TextBlock x:Name="timeTextBlock" Text="Elapsed time" FontSize="36"
    HorizontalAlignment="Center" VerticalAlignment="Center"
    Grid.Row="4" Grid.ColumnSpan="4" MouseDown="TimeTextBlock_MouseDown"/>
```

6 When you added the MouseDown event handler, Visual Studio created a method in the code-behind called TimeTextBlock_MouseDown, just like with the other TextBlocks. Add this code to it:

```
private void TimeTextBlock_MouseDown(object sender, MouseButtonEventArgs e)
{
    if (matchesFound == 8)
    {
        SetUpGame();
    }
}
```

This resets the game if all 8 matched pairs have been found (otherwise it does nothing because the game is still running).

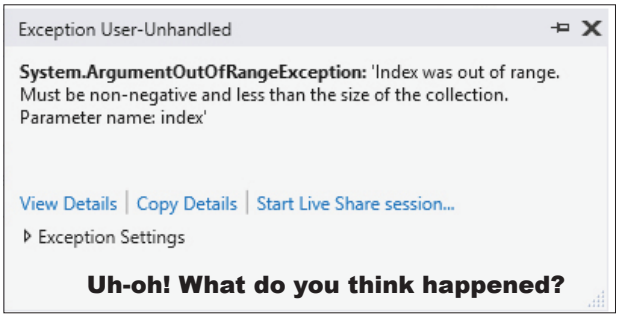
7 And now you have everything you need to finish the Timer_Tick method, which updates the new TextBlock with the elapsed time and stops the timer once the player has found all of the matches:

```
private void Timer_Tick(object sender, EventArgs e)
{
    tenthsOfSecondsElapsed++;
    timeTextBlock.Text = (tenthsOfSecondsElapsed / 10F).ToString("0.0s");
    if (matchesFound == 8)
    {
        timer.Stop();
        timeTextBlock.Text = timeTextBlock.Text + " - Play again?";
    }
}
```

But something's not quite right here. Run your code... oops! You get an **exception**.

We're about to fix this problem. But before we do, take a close look at the error message and line that the IDE highlighted.

Can you guess what caused the error?



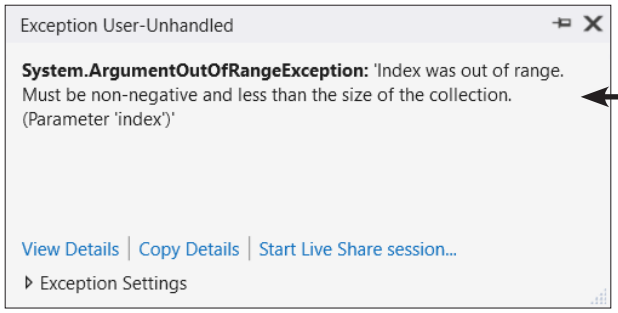
Use the debugger to troubleshoot the exception

You've heard the word "bug" before. You've probably said something like this to your friends at some point in the past: "That game is really buggy, it has so many glitches." But have you ever really stopped to think about what causes bugs? Every bug has an explanation—everything in your program happens for a reason—but not every bug is easy to track down.

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

1 **Restart your game a few times.**

The first thing to notice is that your program always throws the same type of exception with the same message:



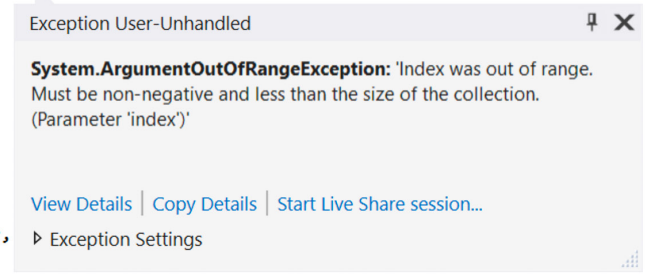
An exception is C#'s way of telling you that something went wrong when your code was running. This is an ArgumentOutOfRange exception with the message that starts with the words "Index was out of range."

When you get an exception, you can often think of it as good news—you found a bug, and now you can fix it.

And if you move the exception window out of the way, you'll see that it always stops on the same line:

Here's the line that's throwing the exception.

```
    foreach (TextBlock textBlock in mainGrid.Children.OfType<TextBlock>())
    {
        int index = random.Next(animalEmoji.Count);
        string nextEmoji = animalEmoji[index];
        textBlock.Text = nextEmoji;
        animalEmoji.RemoveAt(index);
    }
}

TextBlock lastTextBlockClikcked;
bool findingMatch = false;

16 references
private void TextBlock_MouseDown(object sender,
{
```



That means the exception is **reproducible**: you can reliably get your program to throw the exact same exception, and you have a really good idea of where the problem is.

Anatomy of the debugger

When your app is paused in the debugger—that's called "breaking" the app—the Debug controls show up in the toolbar. You'll get plenty of practice using them throughout the book, so you don't need to memorize what they do. For now, just read the descriptions we've written, and hover your mouse over them so you can see their names and shortcut keys.

You can use the Break All button to pause your app. It's grayed out when your app is already paused.

The Restart button restarts your app. It's like stopping it and running it again.

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

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

This button starts your app running again. If you press it now, it will just throw the same exception again.

You've already used the Stop Debugging button to halt your app.

The Show Next Statement button jumps your cursor to the next statement that's about to be executed.

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

2 **Add a breakpoint to the line that's throwing the exception.**

Run your program again so it halts on the exception. Before you stop it, choose **Toggle Breakpoint (F9)** from the Debug menu. As soon as you do, the line will be highlighted in red, and a red dot will appear in the left margin next to the line. Now **stop your app again**—the highlight and dot will still be there:

```
67    int index = random.Next(animalEmoji.Count);
68    string nextEmoji = animalEmoji[index];
69    textBlock.Text = nextEmoji;
```

You've just placed a breakpoint on the line. Your program will now break every time it executes that line of code. Try that out now. Run your app again. The program will halt on that line, but this time ***it won't throw the exception***. Press continue. It halts on the line again. Press continue again. It halts again. Keep going until you see the exception. Now stop your app.

Sharpen your pencil

Run your app again, but this time pay close attention and answer these questions.

1. How many times does your app halt on the breakpoint before the exception? _____

2. A Locals window appears when you're debugging your app. What do you think it does? (If you don't see the Locals window, choose ***Debug >> Windows >> Locals (Ctrl D, L)*** from the menu.)

_____

_____

Sharpen your pencil Solution

Your app halted 17 times.. After the 17th time it threw the exception.

The Locals window shows you the current values of the variables and fields. You can use it to watch them change as your program runs.

3 Gather evidence so you can figure out what's causing the problem.

Did you notice anything interesting in the Locals window when you ran your app? Restart it and keep a really close eye on the animalEmoji variable. The first time your app breaks, you should see this in the Locals window:

Press Continue. It looks like the Count went down by 1, from 16 to 15:

The app is adding random emoji from the animalEmoji list to the TextBlocks and then removing them from the list, so its count should go down by 1 each time. Things go just fine until the animalEmoji list is empty (so Count is 0), then you get the exception. So that's one piece of evidence! Another piece of evidence is that this is happening in a **foreach loop**. And the last piece of evidence is that ***this all started after we added a new TextBlock to the window***.

Time to put on your Sherlock Holmes cap. Can you sleuth out what's causing the exception?

foreach is a kind of loop that runs on every element in a collection.

Behind the Scenes

A loop is a way to run a block of code over and over again. Your code uses a **foreach loop**, or a special kind of loop that runs the same code for each element in a collection (like your animalEmoji List) Here's an example of a foreach loop that uses a List of numbers:

```
List<int> numbers = new List<int>() { 2, 5, 9, 11 };
foreach (int aNumber in numbers)
{
    Console.WriteLine("The number is " + aNumber);
}
```

This foreach loop runs a Console.WriteLine statement for every int in a List of ints

This foreach loop creates a new variable called `aNumber`, then it goes through the `numbers` List in order and executes the `Console.WriteLine` for each of them, setting `aNumber` to each value in the List in order:

```
The number is 2
The number is 5
The number is 9
The number is 11
```

The foreach loop runs the same code over and over again for each element in the collection, setting the variable to the next element each time. So in this case, it sets oneNumber to the next number in the List. and uses it to print a line of text.

This may look a little complex right now—and that's okay! We'll talk about loops in Chapter 2. Then in Chapter 3 we'll come back to foreach loops, and you'll write one yourself that looks a lot like the loop above. So even if this seems a little fast right now, when you come back to this example when you're working on Chapter 3, see if it makes more sense than it did when you first saw it. We bet it will!

4 Figure out what's actually causing the bug.

The reason your program is crashing is because it's trying to get the next emoji from the animalEmoji list but the list is empty, and that causes it to throw an ArgumentOutOfRange exception. But what caused it to run out of emoji to add?

Sleuth it out

Your program worked before you made the most recent change. Then you added a TextBlock... and then it stopped working. Right inside of a loop that iterates through all of the TextBlocks. A clue... how very, very interesting.

So when you run your app, ***it breaks on this line for every TextBlock in the window***. So for the first 16 TextBlocks, everything goes fine because there are enough emoji in the collection:

```
foreach (TextBlock textBlock in mainGrid.Children.OfType<TextBlock>())
{
    int index = random.Next(animalEmoji.Count);
    string nextEmoji = animalEmoji[index];
    textBlock.Text = nextEmoji;
    animalEmoji.RemoveAt(index);
}
```

The debugger highlights the statement that it's about to run. Here's what it looks like just before it throws the exception.

But now that there's a new TextBlock at the bottom of the window, it breaks a 17th time—and since the animalEmoji collection only had 16 emoji in it, it's now empty:

So before you made the change, you had 16 TextBlocks and a list of 16 emoji, so there were just enough emoji to add one to each TextBlock. Now you have 17 TextBlocks but still only 16 emoji, so your program runs out of emoji to add... and then it throws the exception.

5 Fix the bug.

Since the exception is being thrown because we're running out of emoji in the loop that iterates through the TextBlocks, we can fix it by skipping the TextBlock we just added. We can do that by checking the TextBlock's name and skipping the one that we added to show the time. And remove the breakpoint by toggling it again or choosing **Delete All Breakpoints** (**Ctrl-Shift-F9**) from the Debug menu.

```
foreach (TextBlock textBlock in mainGrid.Children.OfType<TextBlock>())
{
    if (textBlock.Name != "timeTextBlock")
    {
        textBlock.Visibility = Visibility.Visible;
        int index = random.Next(animalEmoji.Count);
        string nextEmoji = animalEmoji[index];
        textBlock.Text = nextEmoji;
        animalEmoji.RemoveAt(index);
    }
}
```

Add this if statement inside the foreach loop so that it skips the TextBlock with the name timeTextBlock.

This isn't the only way to fix the bug. One thing you'll learn as you write more code is that there are many, many, MANY ways to solve any problem... and this bug is no exception (no pun intended).

Add the rest of the code and finish the game

There's one more thing you need to do. Your TimeTextBlock_MouseDown method checks the matchesFound field, but that field is never set anywhere. So add these three lines to the SetUpGame method immediately after the closing bracket of the foreach loop:

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

        timer.Start();
        tenthsOfSecondsElapsed = 0;
        matchesFound = 0;
    }
```

Add these three lines of code to the very end of the SetUpGame method to start the timer and reset the fields.

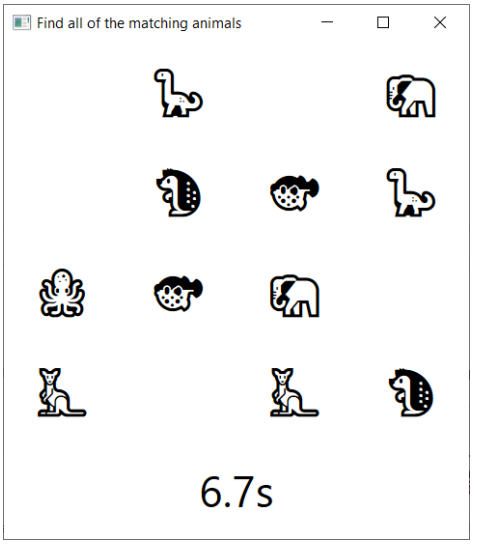
And add this statement to the if/else in TextBlock_MouseDown:

```
else if (textBlock.Text == lastTextBlockClikcked.Text)
{
    matchesFound++;
    textBlock.Visibility = Visibility.Hidden;
    findingMatch = false;
}
```

Add this line of code to increase matchesFound by one every time the player successfully finds a match.

Now your game has a timer that stops when the player finishes matching animals, and when the game is over you can click it to play again. ***You've built your first game in C#. Congratulations!***

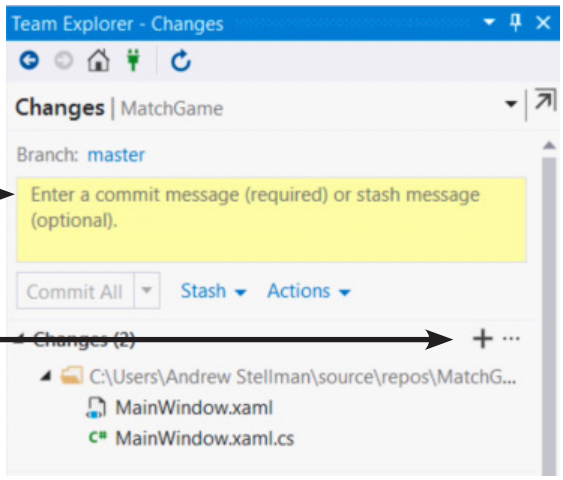
Now your game has a timer that keeps track of how long it takes the player to find all of the matches. Can you beat your lowest time?



Go to https://github.com/head-first-csharp/fourth-edition to view and download the complete code for this project and all of the other projects in this book.

Update your code in source control

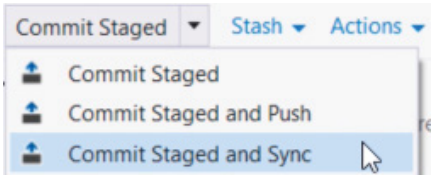
Now that your game is up and running, it's a great time to **push your changes to Git**, and Visual Studio makes it easy to do that. All you need to do is stage your commits, enter a commit message, and then sync to the remote repo.

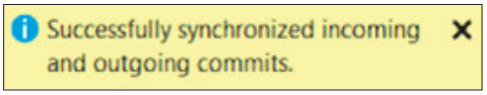


1. Enter a **commit message** that describes what changed.

2. Press the **+** button to **stage** your files, which tells Git that they're ready to be pushed. If you make changes to a file after it's staged, only the staged changes will get pushed back to the remote repo.

3. Choose **Commit Staged and Sync** from the commit dropdown (it's right under the commit message box). It may take a few seconds to sync, and then you'll see a success message in Team Explorer.





Pushing your code to a Git repo is optional—but a really good idea!

IT WAS REALLY USEFUL TO BREAK THE GAME UP INTO SMALLER PIECES THAT I COULD TACKLE ONE AT A TIME.

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

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

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

Even better if's...

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

- Add different kinds of animals so the same ones don't show up each time.
- Keep track of the player's best time so he or she 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 match game.

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

BULLET POINTS

- The IDE tracks the number of times a method is **referenced** elsewhere in the C# or XAML code.
- An **event handler** is a method that your application calls when a specific event like a mouse click, keypress, or window resize happens.
- The IDE 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.
- **Timers** execute Tick event handler methods over and over again on a specified interval.
- **`foreach`** is a kind of loop that iterates through a collection of items.
- When your program throws an **exception**, gather evidence and try to figure out what's causing it.
- Exceptions are easier to fix when they're **reproducible**.
- Visual Studio makes it really easy to use **source control** to easily 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.

Unity Lab

Start Exploring Unity

Welcome to your first **Head First C# Unity lab**. Writing code is a skill, and like any other skill, getting better at it takes **practice and experimentation**, and Unity will be a really valuable tool for that.

Unity is a cross-platform game development tool that you can use to make professional-quality games and simulations. It's also a fun and satisfying way to get **practice with the C# tools and ideas** you'll learn throughout this book. We designed these short, targeted labs after every chapter in this book to help **reinforce** the concepts and techniques you just learned to help you hone your C# skills.

These labs are optional, but valuable practice—**even if you aren't planning on using C# to build games**.

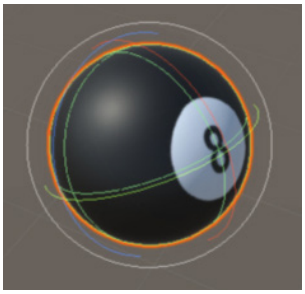
In this first lab, you'll get up and running with Unity. You'll get oriented with the Unity editor, and you'll start creating and manipulating 3D shapes.

Unity is a powerful tool for game design

Welcome to the world of Unity, a complete system for designing professional-quality 2D and 3D games, simulations, tools, and projects. But what is Unity? Unity includes many powerful things, including...

A cross-platform game engine

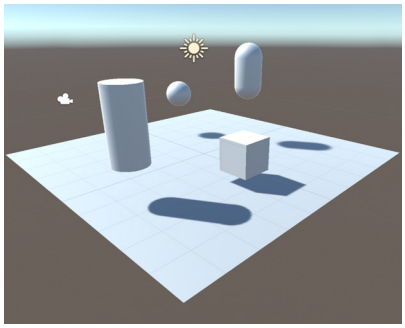
A **game engine** displays the graphics, keeps track of the 2D or 3D characters, detects when they hit each other, makes them act like real-world physical objects, and much, much more. Unity will do all of these things for the 3D games you build throughout this book.



A powerful 2D and 3D scene editor

You'll be spending a lot of time in the Unity Editor, which is like an IDE for your games. It lets you edit levels full of 2D or 3D objects, with tools that you can use to design complete worlds for your games. And since Unity games use C# to define their behavior, the Unity Editor integrates with Visual Studio to give you a seamless game development environment.

While these Unity Labs will concentrate on C# development in Unity, if you're a visual artist or designer, the Unity Editor has many artist-friendly tools designed just for you. Check them out here: https://unity3d.com/unity/features/editor/art-and-design



An ecosystem for game creation

Unity is more than "just" an enormously powerful tool for creating games. It also features an ecosystem to help you build and learn. The Learn Unity page (https://unity.com/learn) has valuable resources to help you learn, and the Unity forums (https://forum.unity.com/) help you connect with other game designers and ask questions. The Unity Asset Store (https://assetstore.unity.com/) lets you download free and paid assets like characters, shapes, and effects, that you can use in your Unity projects.

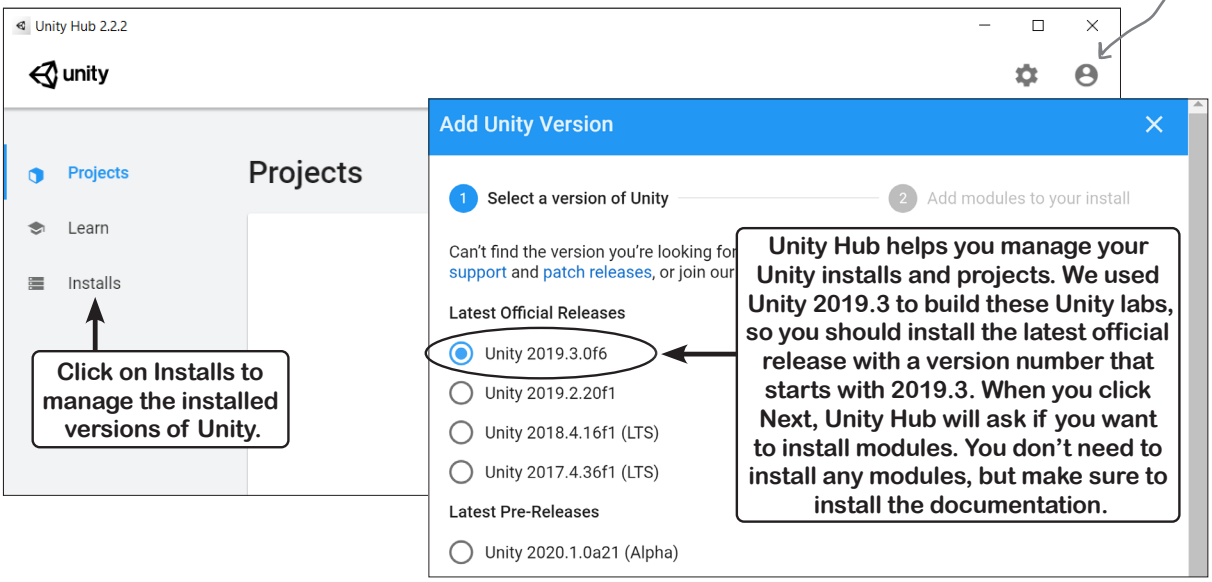
These Unity Labs will focus on using Unity as a tool to explore—and get lots of practice with!—the C# tools and ideas that you've learned throughout the book.

The *Head First C#* Unity Labs are laser-focused on a **developer-centric learning path**. The goal of these Labs is to help you ramp up on Unity quickly, with the same focus on brain-friendly just-in-time learning you'll see throughout *Head First C#* to ***give you lots of targeted, effective practice with C# ideas and techniques***.

Download Unity Hub

All of the screenshots in this book were taken with the free Personal edition of Unity. You'll need to enter your unity.com username and password into Unity Hub to activate your license.

Unity Hub is an application that helps you manage your Unity projects and your Unity installations, and it's the starting point for creating your new Unity project. Start by downloading and installing Unity Hub from https://store.unity.com/download – and once Unity Hub is installed, run it.



Unity Hub lets you install multiple versions of Unity on the same computer, so you should install the same version that we used to build these labs. **Click Official Releases** and install the latest version that starts with ***Unity 2019.3*** – that's the same version we used to take the screenshots in these labs. Once it's installed, make sure that it's set as the preferred version.

The Unity installer may prompt you to install a different version of Visual Studio. You can always have multiple installations of Visual Studio on the same computer. But if you already have one version of Visual Studio installed, there's no need to make the Unity installer add another one.

You can learn more about installing Unity Hub on Windows, MacOS, and Linux here: https://docs.unity3d.com/2019.3/Documentation/Manual/GettingStartedInstallingHub.html

Unity Hub lets you have many Unity installs on the same computer. So even if there's a newer version of Unity available, you can use Unity Hub to install the version we used in the Unity Labs.

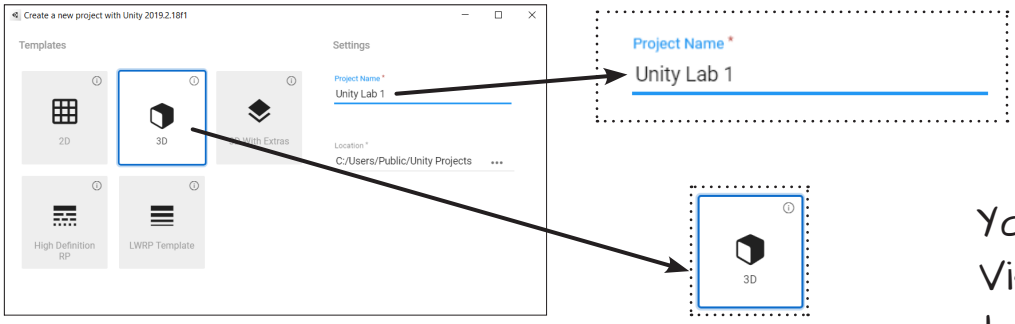
Watch it!

Unity Hub may look a little different.

These screenshots in this book were taken with Unity 2019.3 (Personal Edition) and Unity Hub 2.2.2. You can use Unity Hub to install many different versions of Unity on the same computer, but you can only install the latest version of Unity Hub. The Unity development team is constantly improving Unity Hub and the Unity editor, so it's possible that Unity Hub won't quite match the screenshot on this page. We update these Unity labs for newer printings of Head First C#. We'll add PDFs of updated labs to our GitHub page: https://github.com/head-first-csharp/fourth-edition

Use Unity Hub to create a new project

Click the NEW button on the Project page in Unity Hub to create a new Unity project. Name it ***Unity Lab 1***, make sure the 3D template is selected, and check that you're creating it in a sensible location (usually the "Unity Projects" folder underneath your home directory).

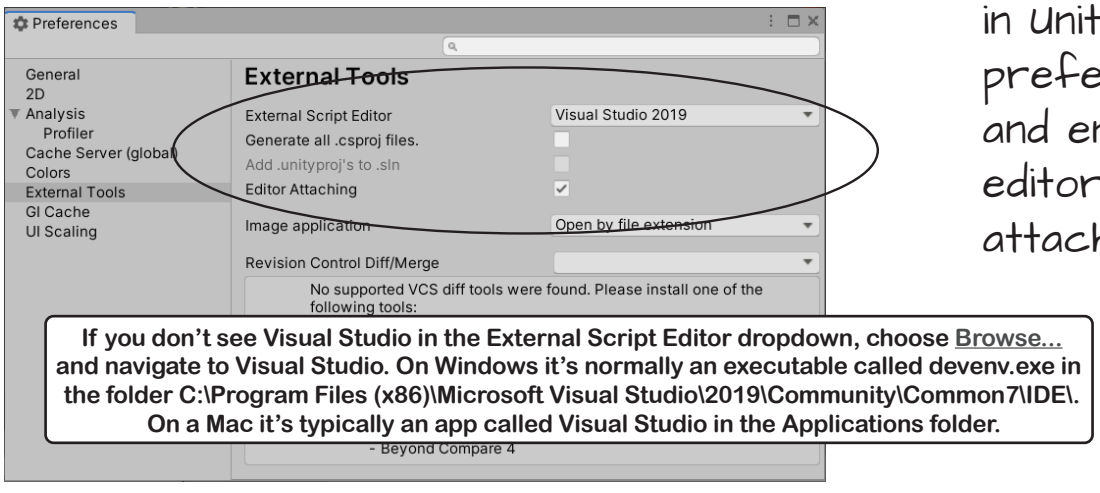


Click Create Project to create the new folder with the Unity project. When you create a new project, Unity generates a lot of files (just like Visual Studio does when it creates new projects for you). It could take Unity a minute or two to create all of the files for your new project.

Make Visual Studio your Unity script editor

The Unity editor works hand-in-hand with the Visual Studio IDE to make it really easy to edit and debug the code for your games. So the first thing we'll do is make sure that Unity is hooked up to Visual Studio. **Choose Preferences from the Edit menu** (or from the Unity menu on a Mac) to open the Unity Preferences window. Click on External Tools on the left, and **choose Visual Studio** from the External Script Editor window. Make sure the **Editor Attaching box is checked**, too—this will let you debug your Unity code in the IDE.

You can use Visual Studio to debug the code in your Unity games. Just choose Visual Studio 2019 as the external script editor in Unity's preferences and enable editor attaching.



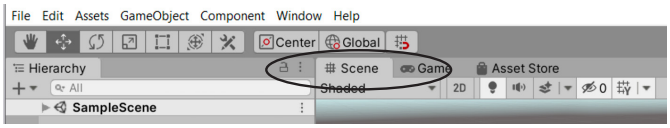
If you don't see Visual Studio in the External Script Editor dropdown, choose Browse... and navigate to Visual Studio. On Windows it's normally an executable called devenv.exe in the folder C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\Common7\IDE\. On a Mac it's typically an app called Visual Studio in the Applications folder.

Okay! You're all ready to get started building your first Unity project.

Take control of the Unity layout

The Unity editor is like an IDE for all of the parts of your Unity project that aren't C#. You'll use it to work with scenes, edit 3D shapes, create materials, and so much more. And like Visual Studio, the windows and panels in the Unity editor can be rearranged in many different layouts.

Find the Scene tab near the top of the window. Click on the tab and drag it to detach the window.

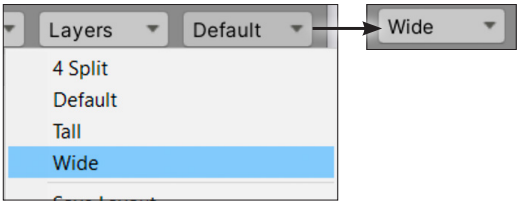


Try docking it inside or next to other panels, then drag it to the middle of the editor to make it a floating window.

The Scene view is your main interactive view of the world that you're creating. You use it to position 3D shapes, cameras, lights, and all of the other objects in your game.

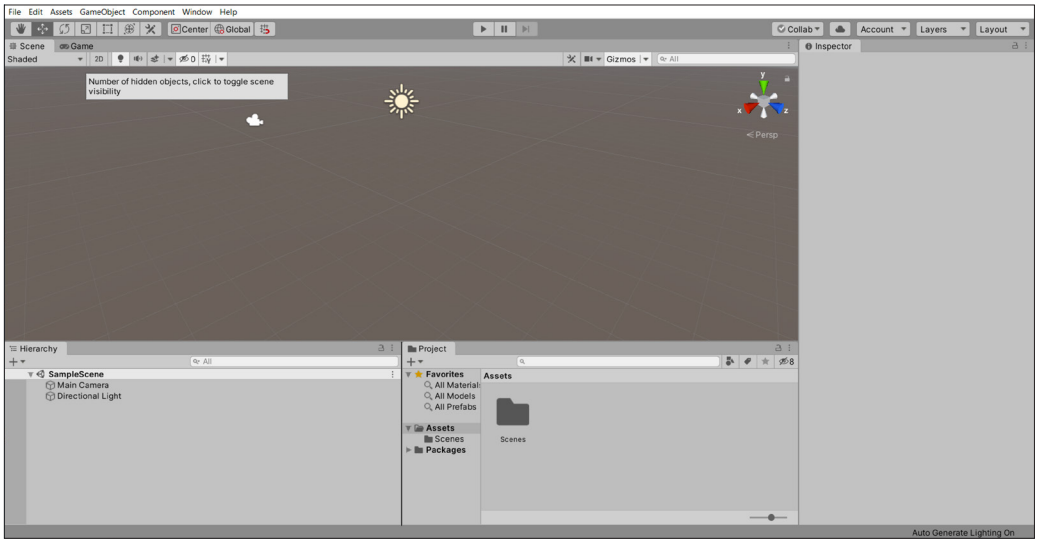
Choose the Wide layout to match our screenshot

We've chosen the Wide layout because it works well for the screenshots in these labs. Find the Layout dropdown and choose Wide so your Unity editor looks like ours.



Once you change the layout with the Layout dropdown on the right side of the toolbar, the dropdown may change its label to match the layout that you selected.

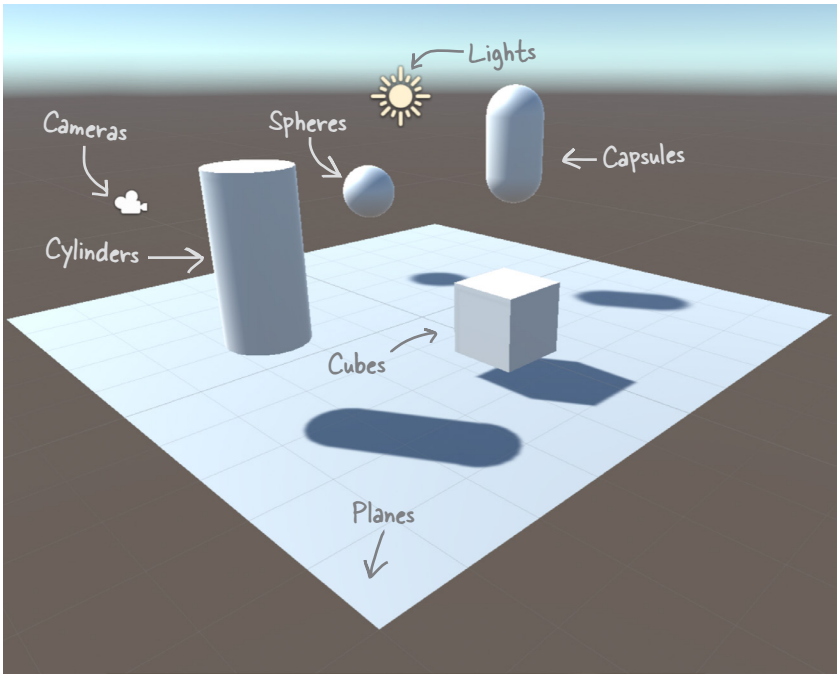
Here's what your Unity editor should look like in the Wide layout.



Unity games are made with GameObjects

When you added a sphere to your scene, you created a new **GameObject**. The GameObject is a fundamental concept in Unity. Every item, shape, character, light, camera, and special effect in your Unity game is a GameObject. Any scenery, characters, and props that you use in a game are represented by GameObjects.

In these Unity Labs, you'll build games out different kinds of GameObjects, including:



Each GameObject contains a number of **components** that provide its shape, set its position, and give it all of its behavior. For example:

- Transform components determine the position and rotation of the GameObject.
- Material components change the way the GameObject is **rendered**—or how it's drawn by Unity—by changing the color, reflection, smoothness, and more.
- Script components use C# scripts to determine the GameObject's behavior.

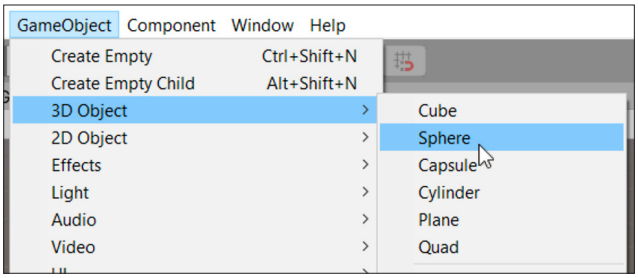
ren-der, verb.
to represent or depict artistically.
*Michelangelo **rendered** his favorite model with more detail than he used in any of his other drawings.*

GameObjects are the fundamental objects in Unity, and components are the basic building blocks of their behavior. The Inspector window shows you details about each GameObject in your scene and its components.

Your scene is a 3D environment

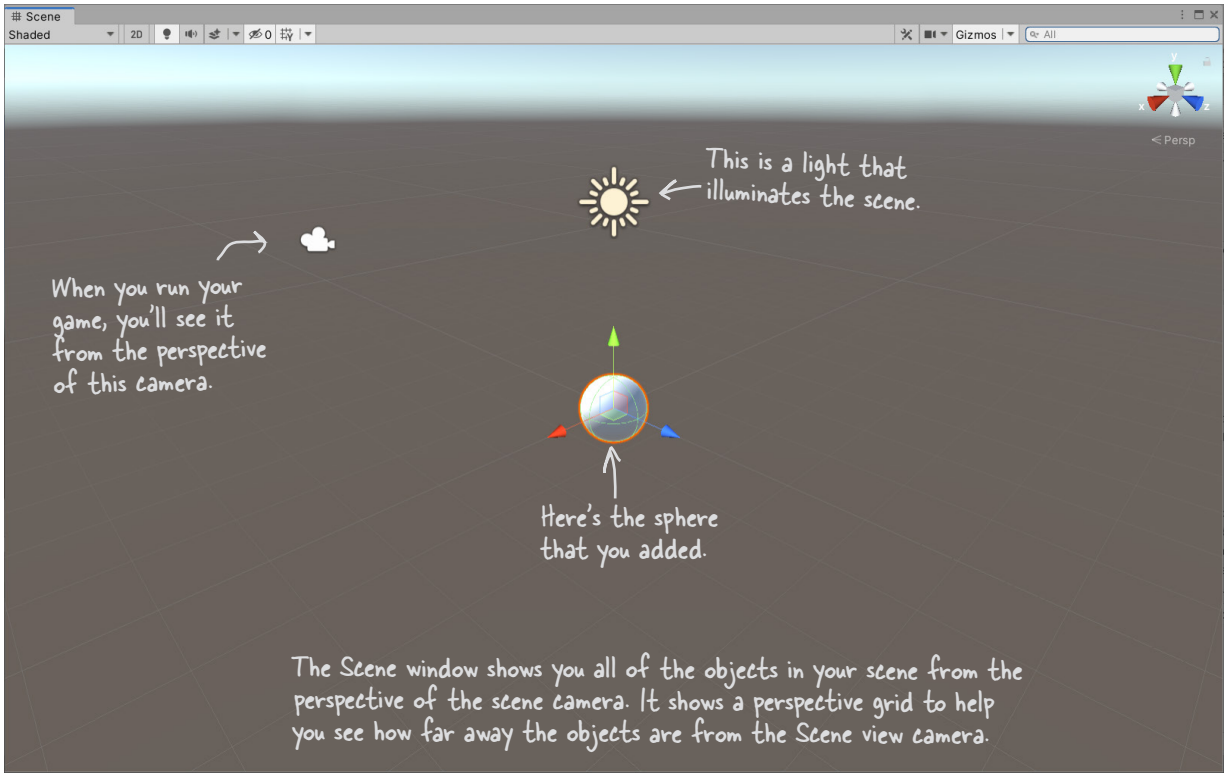
As soon as you start the editor, you're editing a **scene**. You can think of scenes as levels in your Unity games. Every game in Unity is made up of one or more scenes. Each scene contains a separate 3D environment, with its own set of lights, shapes, and other 3D objects. When you created your project, Unity added a scene called SampleScene, and stored it in a file called `SampleScene.unity`.

Add a sphere to your scene by choosing **GameObject >> 3D Object >> Sphere** from the menu:



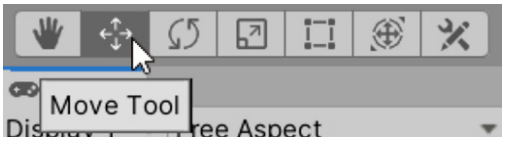
These are called Unity's primitive objects. We'll be using them a lot throughout these Untiy labs.

A sphere will appear in your Scene window. Everything you see in the Scene window is shown from the perspective of the **Scene view camera**, which "looks" at the scene and captures what it sees.



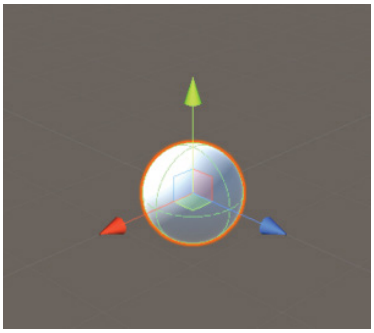
Use the Move Gizmo to move your GameObjects

The toolbar at the top of the Unity Editor lets you choose Transform Tools. If the Move Tool isn't selected, press its button to select it.



The buttons on the left side of the toolbar let you choose Transform Tools like the Move Tool, which displays the Move Gizmo as arrows and a cube on top of the GameObject that's currently selected.

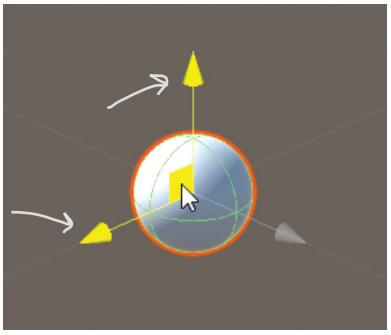
The Move Tool lets you use the **Move Gizmo** to move GameObjects around the 3D space. You should see red, green, and blue arrows and a cube appear in the middle of the window. This is the Move Gizmo, which you can use to move the selected object around the scene.



Did you notice the grid in your 3D space? As you're dragging the sphere around, hold down the control key. That causes the GameObject that you're moving to snap to that grid. You'll see the numbers in the Transform component move by whole numbers instead of small decimal increments.

Move your mouse cursor over the cube at the center of the Move Gizmo—notice how each of the faces of the cube lights up as you move your mouse cursor over it? Click on the upper left face and drag the sphere around. You're moving the sphere in the X-Y plane.

When you click on the upper left face of the cube in the middle of the Move Gizmo, its X and Y arrows light up and you can drag your sphere around the X–Y plane in your scene.



Move your sphere around the scene to get a feel for how the Move Gizmo works. Click and drag each of the three arrows to drag it along each plane individually. Then try clicking on each of the faces of the cube in the Scene Gizmo to drag it around all three planes. Notice how the sphere gets smaller as it moves farther away from you—or really, the scene camera—and larger as it gets closer.

The Move Gizmo lets you move GameObjects along any axis or plane of the 3D space in your scene.

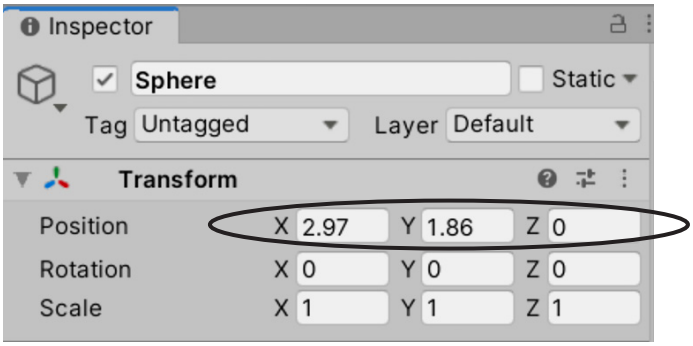
The Inspector shows your GameObject's components

As you move your sphere around the 3D space, watch the **Inspector window**, which is on the right side of the Unity Editor if you're using the Wide layout. Look through the Inspector window—you'll see that your sphere has four components labeled Transform, Sphere (Mesh Filter), Mesh Renderer, and Sphere Collider.

Every GameObject has a set of components that provide the basic building blocks of its behavior, and every GameObject has a **Transform component** that drives its location, rotation, and scale.

You can see the Transform component in action as you use the Move Gizmo to drag the sphere around the X-Y plane. Watch the X and Y numbers in the Position row of the Transform component change as the sphere moves.

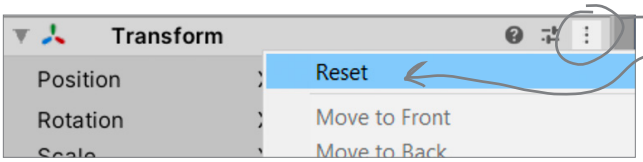
If you accidentally deselect a GameObject, just click on it again. If it's not visible in the scene, you can select it in the Hierarchy window, which shows all of the GameObjects in the scene.



Try clicking on each of the other two faces of the Move Gizmo cube and drag to move the sphere in the X-Z and Y-Z planes. Then click on the red, green, and blue arrows and drag the sphere along just the X, Y, or Z axis. You'll see the X, Y, and Z values in the Transform component change as you move the sphere.

Now **hold down shift** to turn the cube in the middle of the Gizmo into a square. Click and drag on that square to move the sphere in the plane that's parallel to the Scene view camera.

Once you're done experimenting with the Move Gizmo, use the sphere's Transform component context menu to reset its Transform component. Click the **context menu button** (⋮) at the top of the transform component and choose Reset from the menu.



Use the context menu to reset a component. You can either click the three dots or right-click anywhere in the top line of the Transform panel in the Inspector window to bring up the context menu.

The position will reset back to [0, 0, 0].

You can learn more about the tools and how to use them to position GameObjects in the Unity Manual. Click Help >> Unity Manual and search for the "Positioning GameObjects" page.

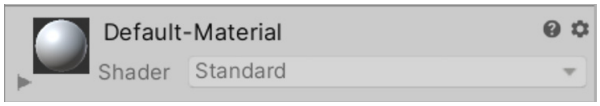
Save your scene often! Use File >> Save or Ctrl-S to save the scene right now.

Add a material to your sphere GameObject

Unity uses **materials** to provide color, patterns, textures, and other visual effects. Your sphere looks pretty boring right now because it just has the default material, which causes the 3D object to be rendered in a plain, off-white color. Let's make it look like a billiard ball.

① Select the sphere.

When the sphere is selected, you can see its material as a component in the Inspector window:



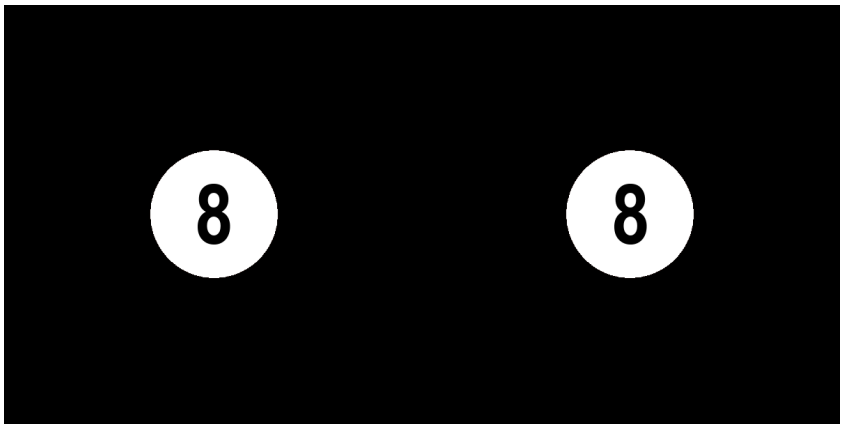
We'll make your sphere more interesting by adding a **texture**—that's just a simple image file that's wrapped around a 3D shape, almost like you printed the picture on a rubber sheet and stretched it around your object.

② Go to our Billiard Ball Textures page on GitHub.

Go to https://github.com/head-first-csharp/fourth-edition and click on the *Billiard Ball Textures* link to browse a folder of texture files for a complete set of billiard balls.

③ Download the texture for the 8 ball.

Click on the file **`8 Ball Texture.png`** to view the texture for an 8 ball. This is a normal 1200x600 image file that you can open in your favorite image viewer.



We designed this image file so that it looks like an 8 ball when Unity "wraps" it around a sphere.

Download the file into a folder on your computer.

(You might need to right-click on the Download button to save the file, or click Download to open it and then save it, depending your browser.)

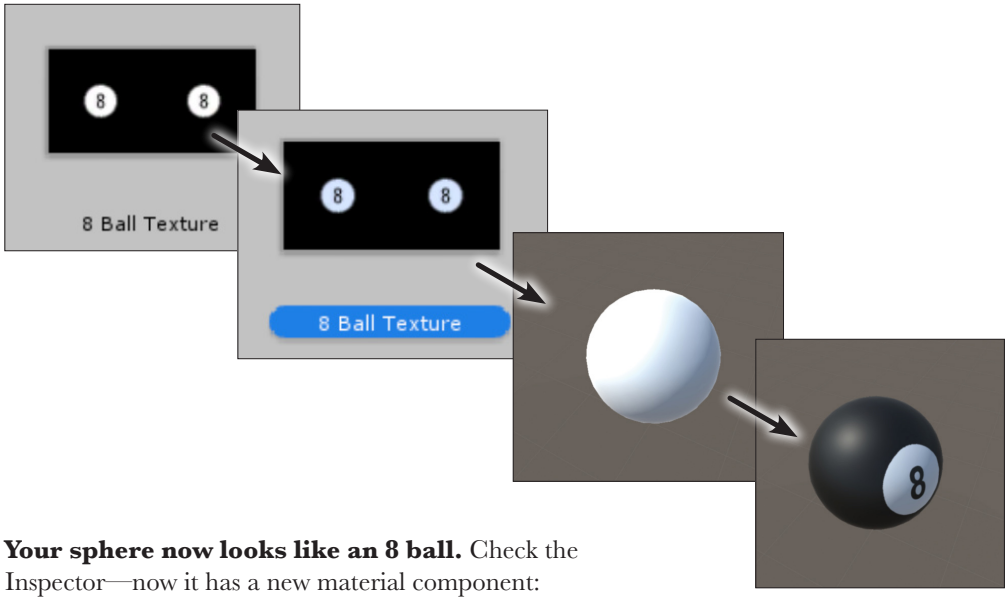
④ **Import the 8 Ball Texture image into your Unity project.**

Right-click on the Assets folder in the Project window, choose **Import New Asset...** and import the texture file. You should now see it when you click on the Assets folder in the Project window.

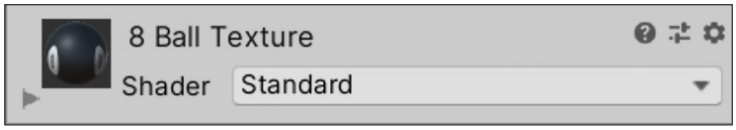
You right-clicked inside the Assets folder in the Project window to import the new asset, so Unity imported the texture into that folder.

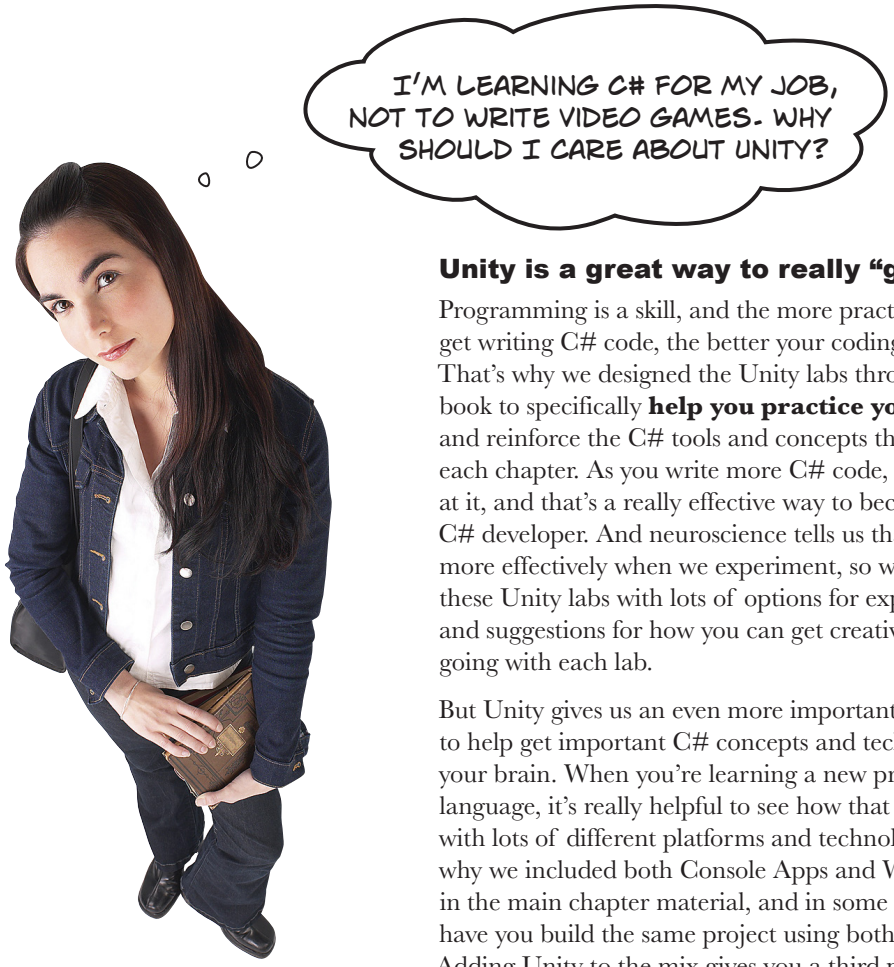
⑤ **Add the texture to your sphere.**

Now we just need to take that texture and "wrap" it around your sphere. Click on 8 Ball Texture in the Project window to select it. Once it's selected, **drag it onto your sphere**.



Your sphere now looks like an 8 ball. Check the Inspector—now it has a new material component:





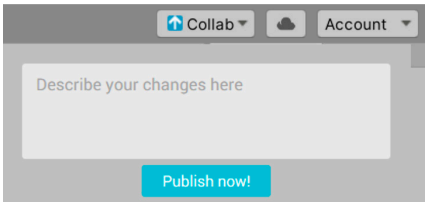
Unity is a great way to really "get" C#.

Programming is a skill, and the more practice that you get writing C# code, the better your coding skills will get. That's why we designed the Unity labs throughout this book to specifically **help you practice your C# skills** and reinforce the C# tools and concepts that you learn in each chapter. As you write more C# code, you'll get better at it, and that's a really effective way to become a great C# developer. And neuroscience tells us that we learn more effectively when we experiment, so we designed these Unity labs with lots of options for experimentations, and suggestions for how you can get creative and keep going with each lab.

But Unity gives us an even more important opportunity to help get important C# concepts and techniques into your brain. When you're learning a new programming language, it's really helpful to see how that language works with lots of different platforms and technologies. That's why we included both Console Apps and WPF Apps in the main chapter material, and in some cases even have you build the same project using both technologies. Adding Unity to the mix gives you a third perspective, which can really accelerate your understanding of C#.

You probably noticed that we haven't talked about adding your Unity projects to source control. That's because we've been using Visual Studio to take care of source control for us, but it doesn't have a way of knowing about exactly which files are part of your Unity project. Luckily, there's an easy way to back up and share your Unity projects: **Unity Collaborate**, which lets you publish your projects to their cloud storage. Your Unity personal account comes with 1GB of cloud storage for free, which is enough for all of the Unity Lab projects in this book. Unity will even keep track of your project history (which doesn't go against your storage limit).

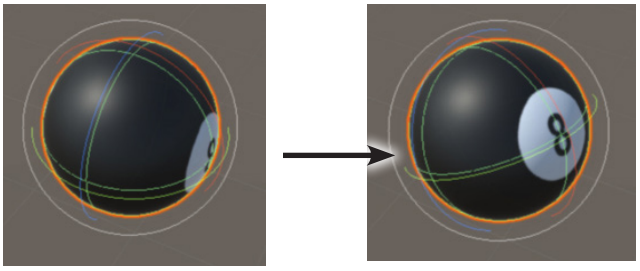
To publish your project, click the **Collab (** Collab **) button** on the toolbar, then click Publish. Use the same button to publish any updates. To see your published projects, log into https://unity3d.com and use the account icon to view your account, then click the Projects link from your account overview page to see your projects.



Rotate your sphere

Click the **Rotate Tool** in the toolbar. You can use the Q, W, E, R, T, and Y keys to quickly switch between the Transform Tools—press E and W to toggle between the Rotate Tool and Move Tool.

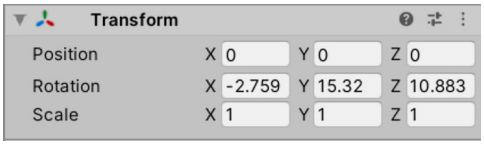
1 **Click on the sphere.** Unity will display a wireframe sphere Rotate Gizmo with red, blue, and green circles. Click the red circle and drag it to rotate the sphere around the X axis.



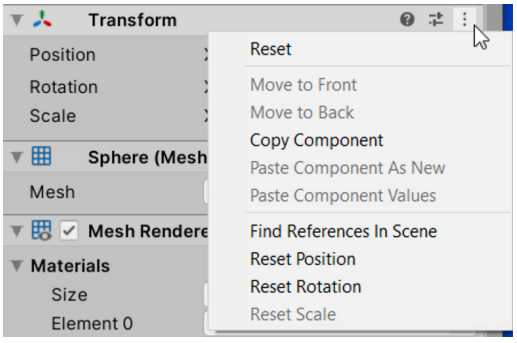
It's easy to reset your windows and scene camera.

If you pan or rotate your Scene view camera so you can't see your sphere, or if you drag your windows out of position, just use the layout dropdown in the upper right corner to **reset the Unity editor to the Wide layout**. It will reset the window layout and move the Scene view camera back to its default position. Go ahead... try it now!

2 **Click and drag the green and blue circles to rotate on the Y and Z axes.** The outer white circle rotates the sphere along the axis coming out of the Scene view camera. Watch the Rotation numbers change in the Inspector window.



3 **Open the context menu of the Transform panel in the Inspector window.** Click Reset, just like you did before. It will reset everything in the Transform component back to default values—in this case, it will change your sphere's rotation back to [0, 0, 0].



Click the three dots (or anywhere in the header of the Transport window) to bring up the component context menu. The Reset option at the top of the menu resets the component to its default values.

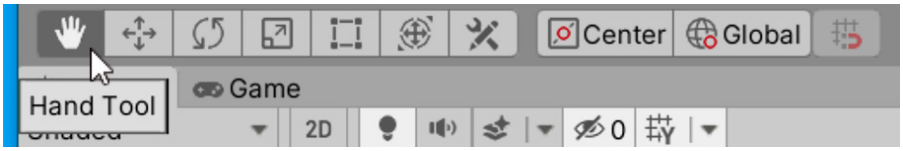
Use these options from the context menu to reset the position and rotation of a GameObject.

Use File >> Save or Ctrl-S to save the scene right now. Save early, save often!

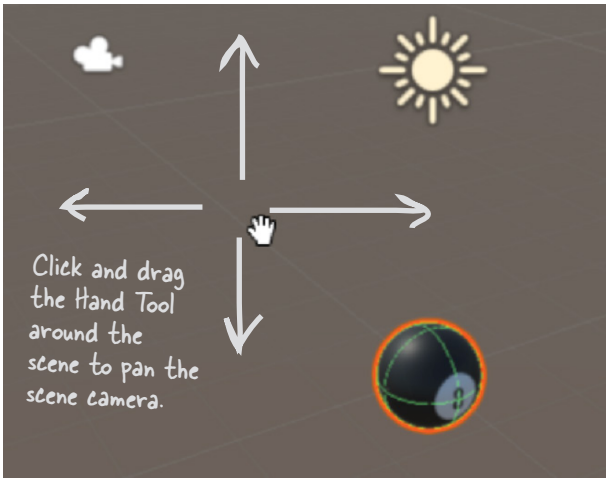
Move the Scene view camera with the Hand Tool and Scene Gizmo

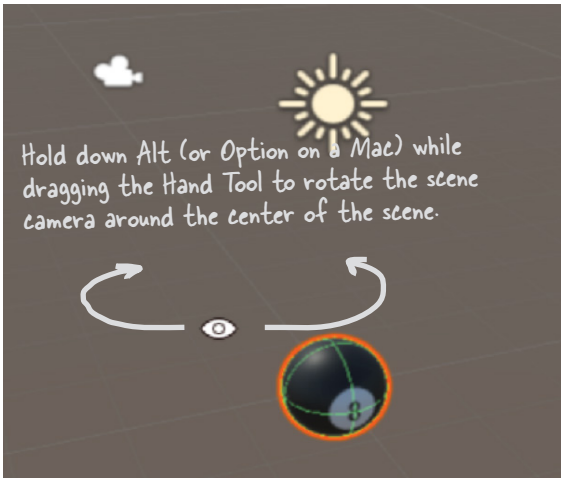
Use the mouse scroll wheel or scroll feature on your trackpad to zoom in and out, and toggle between the Move and Rotate Gizmos. Notice that the sphere changes size, but the Gizmos don't. The Scene window in the editor shows you the view from a virtual **camera**, and the scroll feature zooms that camera in and out.

Press Q to select the **Hand Tool**, or choose it from the toolbar. Your cursor will change to a hand.

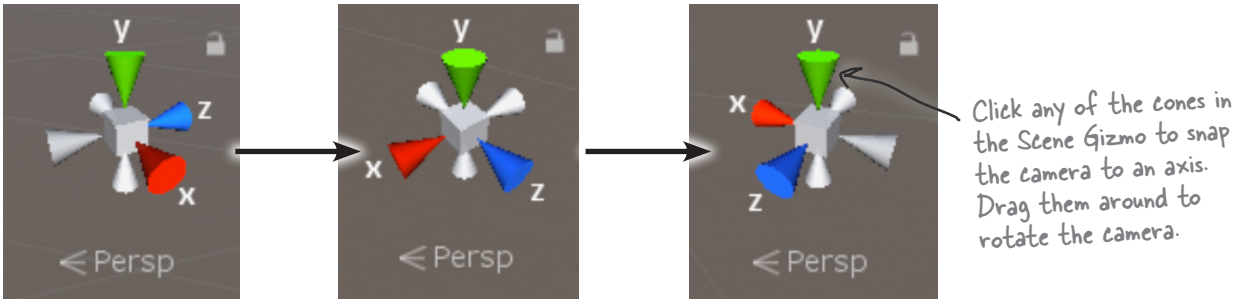


The Hand Tool pans around the scene by changing the position and rotation of the scene camera. When the Hand Tool is selected, you can click anywhere in the scene to pan. Hold down Alt (or Option on a Mac) while dragging and the Hand Tool turns into an eye and rotates the view around the center of the window.





When you Alt-click (or Option-click) and drag the Hand Tool to rotate the Scene view around its center, keep an eye on the **Scene Gizmo** in the upper right corner of the Scene window. The Scene Gizmo always displays the camera's orientation— keep your eye on it as you use the Hand Tool to move the Scene view camera. Click on the X, Y, and Z cones to snap the camera to an axis.



there are no Dumb Questions

Q: I'm still not clear on exactly what a component is. What does it do, and how is it different than a GameObject?

A: A GameObject doesn't actually do much on its own. All a GameObject really does is serve as a *container* for components. When you used the GameObject menu to add a Sphere to your scene, Unity created a new GameObject and added all of the components that make up a sphere, including a Transform component to give it position, rotation, and scale, a default Material to give it its plain white color, and a few other components to give it its shape, help your game figure out when it bumps into other objects. These components are what make it a sphere.

Q: So does that mean I can just add any component to a GameObject and it gets that behavior?

A: Yes, exactly. When Unity created your scene, it added two GameObjects, one called Main Camera and another called Directional Light. If you click on Main Camera in the Hierarchy window, you'll see that it has three components: a Transform, a Camera, and an Audio Listener. If you think about it, that's all a camera actually needs to do: be somewhere, and pick up visuals and audio. And the Directional Light GameObject just has two components: a Transform and a Light, which casts light on other GameObjects in the scene.

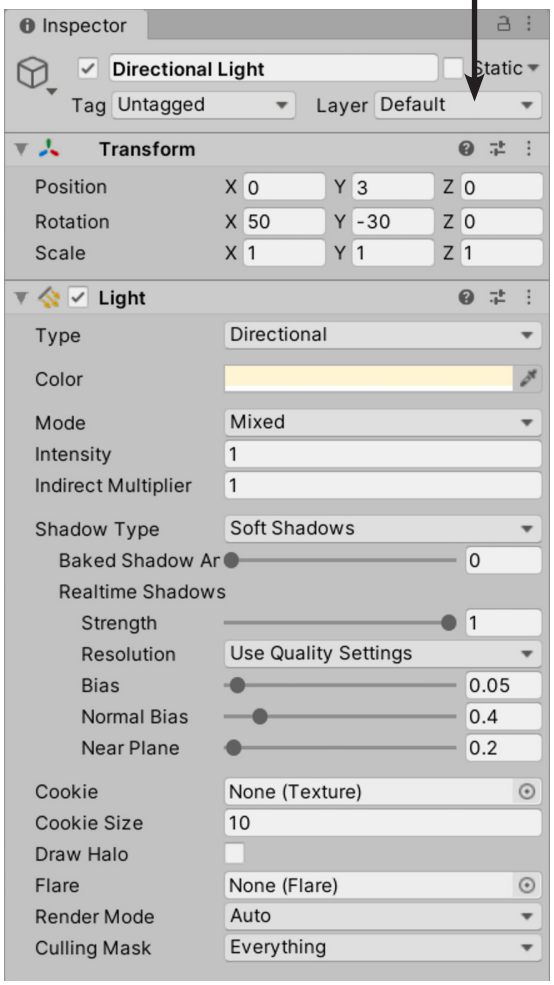
Q: If I add a Light component to any GameObject, does it become a light?

A: Yes! A light is just a GameObject with a Light component. If you click on the Add Component button at the bottom of the Inspector and add a Light component to your ball, it will start emitting light. If you add another GameObject to the scene, it will reflect that light.

Q: It sounds like you're being careful with the way you talk about light. Is there a reason you talk about emitting and reflecting light? Why don't you just say that it glows?

A: Because there's a difference between a GameObject that emits light and one that glows. If you add a Light component to your ball, it will start emitting light—but it won't look any different, because the Light only affects other GameObjects in the scene that reflect its light. If you want your GameObject to glow, you'll need to change its material or use another component that affects how it's rendered.

You can click on the folder icon in any of the components to bring up the Unity manual page for it.



When you click on the Directional Light GameObject in the Hierarchy window, the Inspector shows you its components. It just has two components: a Transform component that provides its position and rotation and a Light component that actually casts the light.

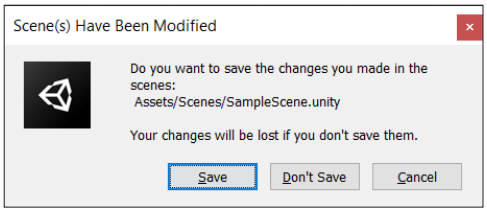
Get creative!

We built these Unity Labs to give you a **platform to experiment on your own with C#** because that's the single most effective way for you to become a great C# developer. At the end of every Unity Lab, we'll include a few suggestions for things that you can try on your own. Take some time to experiment with everything you just learned before moving on to the next chapter.

- Add a few more spheres to your scene. Try using some of the other billiard ball maps. You can download them all from the same location that you downloaded `8 Ball Texture.png`.
- Try adding other shapes by choosing different Cube, Cylinder, or Capsule from the GameObject >> 3D Object menu.
- Experiment with using different images as textures. See what happens to photos of people or scenery when you use them to create textures and add them to different shapes.
- Can you create an interesting 3D scene out of shapes, textures, and lights?

The more C# code that you write, the better you'll get at it. That's the most effective way for you to become a great C# developer. We designed these Unity Labs to give you a platform for practice and experimentation.

When you're ready to move on to the next Chapter, make sure you save your project, because you'll come back to it in the next Lab.. Unity will prompt you to save when you quit.

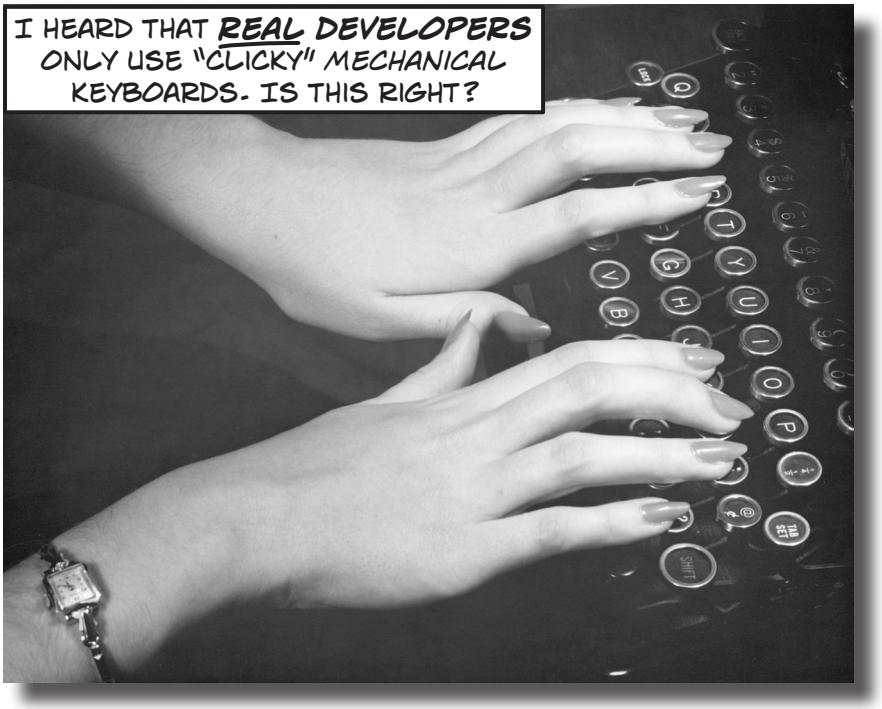


BULLET POINTS

- The **Scene view** is your main interactive view of the world that you're creating.
- The **Move Gizmo** lets you move objects around your scene. The **Scale Gizmo** lets you modify your GameObjects' scale.
- The **Scene Gizmo** always displays the camera's orientation.
- Unity uses **materials** to provide color, patterns, textures, and other visual effects.
- Some materials use **textures**, or image files wrapped around shapes.
- Your game's scenery, characters, props, cameras, and lights are all built from **GameObjects**.
- GameObjects are the fundamental objects in Unity, and **components** are the basic building blocks of their behavior.
- Every GameObject has a **Transform component** that provides its position, rotation, and scale.
- The **Project window** gives you a folder-based view of your project's assets, including media files, C# scripts, and textures.
- The **Hierarchy window** shows all of the GameObjects in the scene.
- **Unity Collaborate** makes it easy to back up your projects to cloud storage. A free Unity Personal account comes with 1GB of storage, enough for all of these Unity Labs.

2 dive into C#

Statements, classes, and code



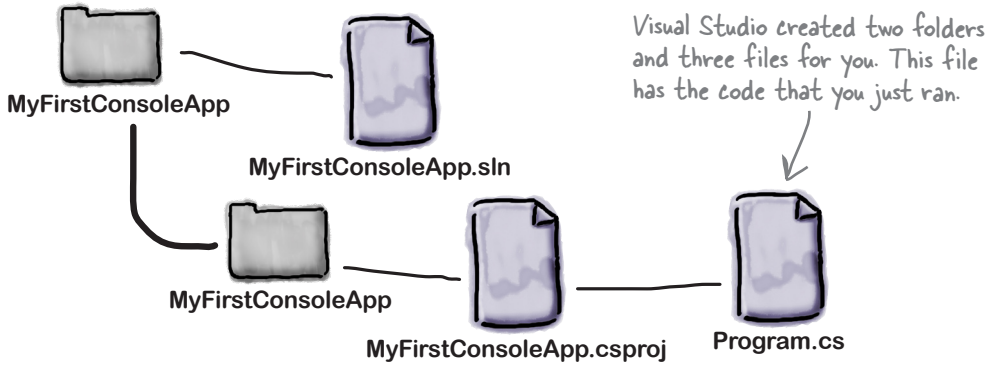
You're not just an IDE user. You're a developer.

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

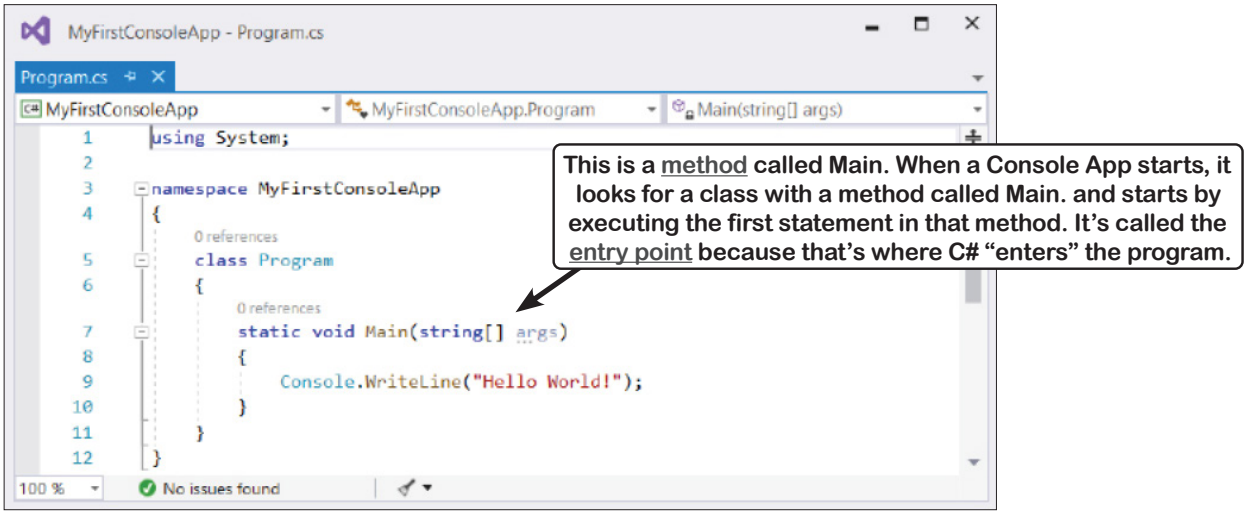
(And for the record, you can be a **real developer** no matter what kind of keyboard you prefer. The only thing need to do is **write code**!)

Let's take a closer look at the files for a console app

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



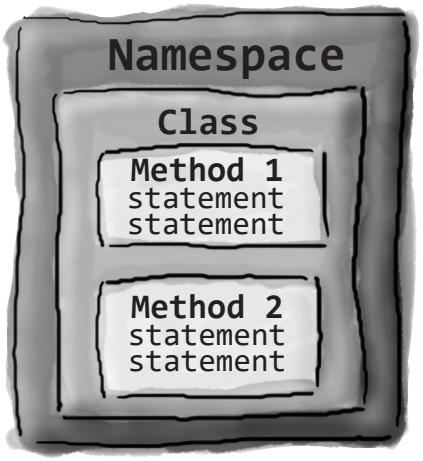
Let's take a closer look at the Program.cs file that it created. Open it up in Visual Studio:



- At the top of the file is a **`using` directive**. You'll see `using` lines like this in all of your C# code files.
- Right after the using directives comes the **namespace keyword**. Your code is in a namespace called MyFirstConsoleApp. Right after it is an opening curly bracket **`{`**, and at the end of the file is the closing bracket **`}`**. Everything between those brackets is in the namespace.
- Inside the namespace is a **class**. Your program has one class called Program. Right after the class declaration is an opening curly bracket, with its pair in the second-to-last line of the file.
- Inside your class is a **method** called Main—again, followed by a pair of brackets with its contents.
- Your method has one **statement**: `Console.WriteLine("Hello World!");`

Anatomy of a C# program

Every C# program's code is structured in exactly the same way. All programs use namespaces, classes, and methods to make your code easier to manage.



When you create classes, you define namespaces for them so that your classes are separate from the ones that come with .NET.

A class contains a piece of your program (although some very small programs can have just one class).

A class has one or more methods. Your methods always have to live inside a class. And methods are made up of statements—like the Console.WriteLine statement your app used to print a line to the console.

The order of the methods in the class file doesn't matter. Method 2 can just as easily come before method 1.

A statement performs one single action

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

there are no Dumb Questions

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

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

Two classes can be in the same namespace (and file!)

Take a look at these two C# code files from a program called PetFiler2. They contain three classes: a Dog class, a Cat class, and a Fish class. Since they're all in the same PetFiler2 namespace, statements in the Dog.Bark method can call Cat.Meow and Fish.Swim ***without adding a using directive***.

SomeClasses.cs

```
namespace PetFiler2 {

  public class Dog {
     public void Bark() {
       // statements go here
     }
  }

  public partial class Cat {
     public void Meow() {
        // more statements
     }
  }
}
```

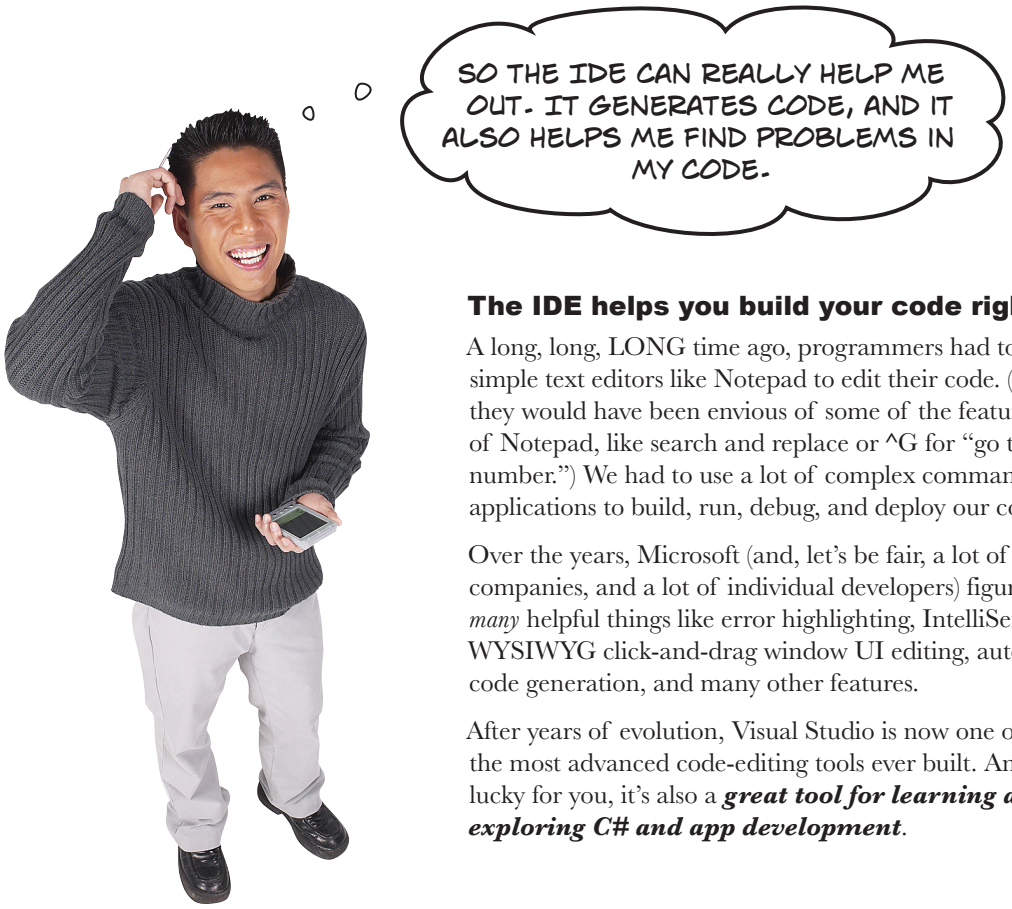
When a method is marked public that means it can be used by other classes.

MoreClasses.cs

```
namespace PetFiler2 {

 public class Fish {
     public void Swim() {
       // statements
     }
 }

 public partial class Cat {
     public void Purr() {
        // statements
     }
  }
}
```

A class can span multiple files too, but you need to use the `partial` keyword when you declare it. It doesn't matter how the various namespaces and classes are divided up between files. They still act the same when they're run.

You can only split a class up into different files if you use the `partial` keyword. You probably won't do that in much of the code you write in this book, but you'll see it later in this chapter, and we want to make sure there are no surprises.



The IDE helps you build your code right.

A long, long, LONG time ago, programmers had to use simple text editors like Notepad to edit their code. (In fact, they would have been envious of some of the features of Notepad, like search and replace or ^G for "go to line number.") We had to use a lot of complex command-line applications to build, run, debug, and deploy our code.

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

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

there are no Dumb Questions

Q: I've seen the phrase "Hello World" before. Does it mean something special?

A: A "Hello World!" program is a program that does one thing: it outputs the phrase "Hello World!" In most programming languages this is simple to do. It shows that you can actually get something working, so it's often the first program you write in a new language.

Q: That's a lot of curly brackets—it's hard to keep track of them all. Do I really need so many of them?

A: C# uses curly brackets (some people say "braces" or "curly braces," and we may use "braces" instead of "brackets" sometimes, too) to group statements together into blocks. Brackets always come in pairs. You'll only see a closing curly bracket after you see an opening one. The IDE helps you match up curly brackets—click on one, and you'll see it and its match change color. You can also use the — button on the left of the editor to collapse and expand them.

Q: So what exactly *is* a namespace, and why do I need it?

A: Namespaces help keep all of the tools that your programs use organized. When your app printed a line of output, it used a class called Console that's part of **.NET Core**. That's an open-source, cross-platform framework that a lot of classes that you can use to build your apps. And we mean a LOT—literally thousands and thousands of classes—so .NET uses namespaces to keep them organized. The Console class is in a namespace called System, so your code needs `using System;` at the top to use it.

Q: I don't quite get what the entry point is. Can you explain it one more time?

A: Your program has a whole lot of statements in it, but they can't all run at the same time. The program starts with the first statement in the program, executes it, and then goes on to the next one, and the next one, etc. Those statements are usually organized into a bunch of classes.

So when you run your program, how does it know which statement to start with? That's where the entry point comes in. Your code won't build unless there is **exactly one method called `Main`**. It's called the entry point because the program starts running—can we say that it *enters* the code—with the first statement in the `Main` method

Q: So my console app really runs on Linux and MacOS?

A: Yes! .NET Core is the cross-platform implementation of .NET (including classes like List and Random), so you can run your app on your Mac or Linux box.

If you have a computer running Mac or Linux, you can try this out right now. First, you'll need to install .NET Core on it, which you can download here: *https://dotnet.microsoft.com/download*

Once it's installed, find your project folder by clicking on MyFirstConsoleApp in the Solution Explorer. Open the project folder, go to the subdirectory under bin/Debug/, and copy all of the files to your Mac or Linux computer. Then you can run it—and this will work on ***any* Windows, Mac, or Linux box** with .NET Core installed:

```
Andrews-MacBook-Pro — -bash — 46×5
$ dotnet MyFirstConsoleApp.dll
Hello World!
$
```

Q: I can usually run programs in Windows by double-clicking on them, but I can't double-click on that .dll file. Can I create an executable that I can run in Windows?

A: Yes. You can use dotnet.exe to publish binaries for different platforms. Open a Command Prompt, go to the folder with either your .sln or .csproj file and run this command (this will work on any operating system with dotnet installed, not just Windows):

```
dotnet publish -c Release -r win10-x64
```

The last line of the output should be MyFirstConsoleApp -> followed by a folder. That folder will contain MyFirstConsoleApp.exe (and a bunch of DLL files that it needs to run). You can also build executable programs for other platforms. Replace win10-x64 with osx-x64 to publish a MacOS app, or linux-x64 to publish a Linux app. Those are called **runtime identifiers** (or RIDs), and there's a list of RIDs here: https://docs.microsoft.com/en-us/dotnet/core/rid-catalog

Statements are the building blocks for your apps

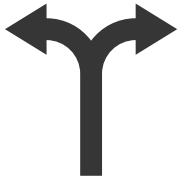
Your app is made up of classes, and those classes contain methods, and those methods contain statements. So if we want to build apps that do a lot of things, we'll need a few **different kinds of statements** to make them work. You've already seen one kind of statement:

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

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

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

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



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

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

Your programs use variables to work with data

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

Declare your variables

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

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

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

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

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

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

Variables vary

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

```
int myHeight = 63;
```

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

```
myHeight = 12;
```

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

Whenever your program needs to work with numbers, text, true/false values, or any other kind of data, you'll use variables to keep track of them.

You need to assign values to variables before you use them

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

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

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

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

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

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

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

A few useful types

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

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

Generate a new method to work with variables

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

Do this!

1 Add a method to your new MyFirstConstoleApp project.

Open the Console App project that you created in the last chapter. The IDE created your app with a Main method that has exactly one statement:

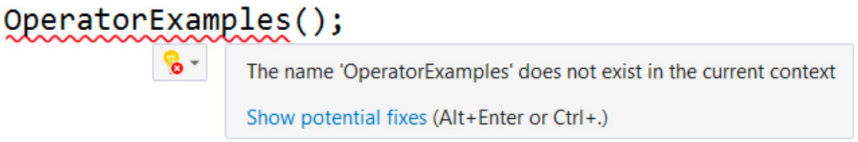
```
Console.WriteLine("Hello World!");
```

Replace this with a statement that calls a method:

```
OperatorExamples();
```

2 Let Visual Studio tell you what's wrong.

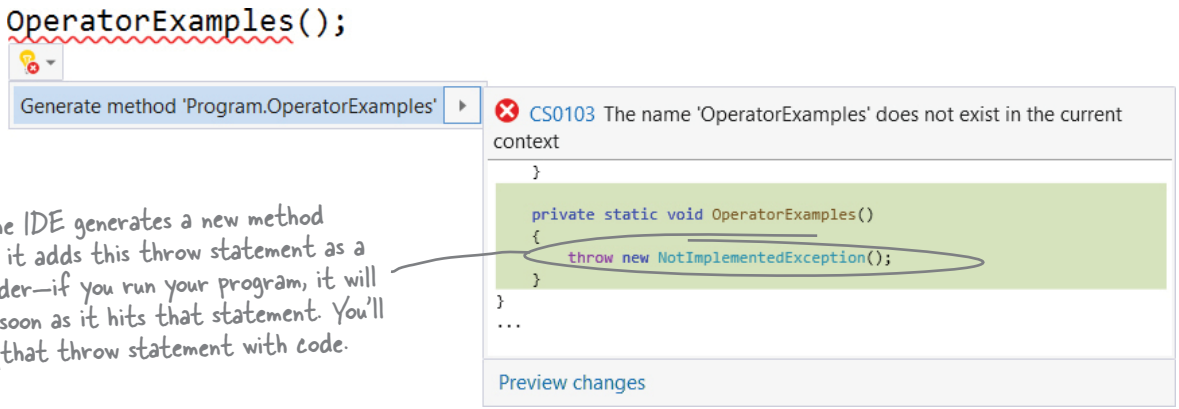
As soon as you finish replacing the statement, Visual Studio will draw a red squiggly underline beneath your method call. Hover your mouse cursor over it. The IDE will display a pop-up window:



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

3 Generate the OperatorExamples method.

The last line of the pop-up window tells you to press Alt+Enter or Ctrl+. to see the potential fixes. So go ahead and press either of those key combinations (or click on the dropdown to the left of the pop-up).



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

The IDE has a solution: it will generate a method called OperatorExamples in your Program class. **Click Preview changes** to display a window that has the IDE's potential fix—adding a new method. Then **click Apply** to add the method to your code.

Add code that uses operators to your method

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

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

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

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

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

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

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

MINI Sharpen your pencil

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

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

Line 1: _____

Line 2: _____

Line 3: _____

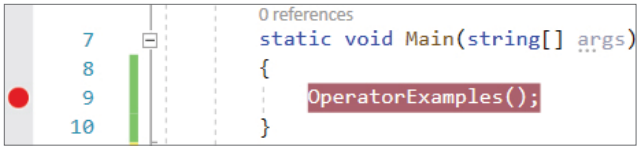
Use the debugger to watch your variables change

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

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

1 Add a breakpoint and run your program.

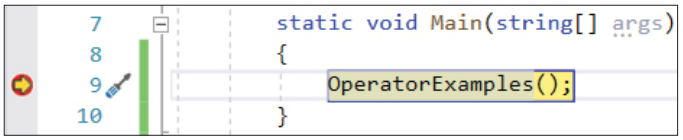
Place your mouse cursor on the method call that you added to your program's Main method and **choose Toggle Breakpoint (F9) from the Debug menu**. The line should now look like this:



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

2 Step into the method.

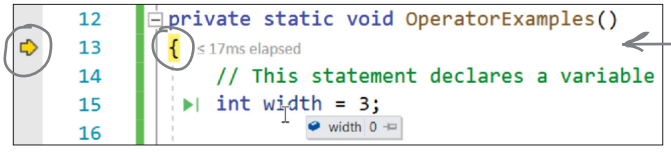
Your debugger is stopped at the breakpoint on the statement that calls the OperatorExamples method.



Press *Step Into (F11)* – the debugger will jump into the method, then stop before it runs the first statement.

3 Examine the value of the width variable.

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

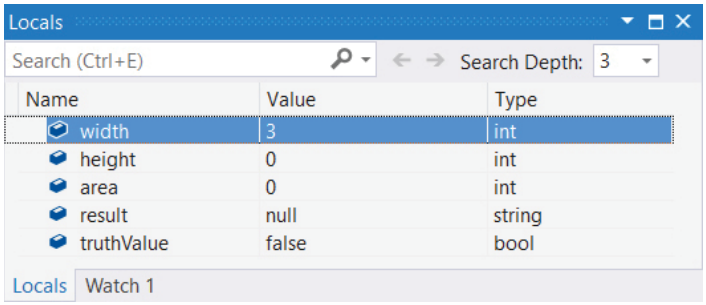


The highlighted bracket and arrow in the left margin mean the code is paused just before the first statement of the method.

The IDE displays a pop-up that shows the current value of the variable—it's currently 0. Now **press Step Over (F10)**—the execution jumps over the comment to the first statement, which is now highlighted. We want to execute it, so **press Step Over (F10) again**. Hover over `width` again. It now has a value of 3.

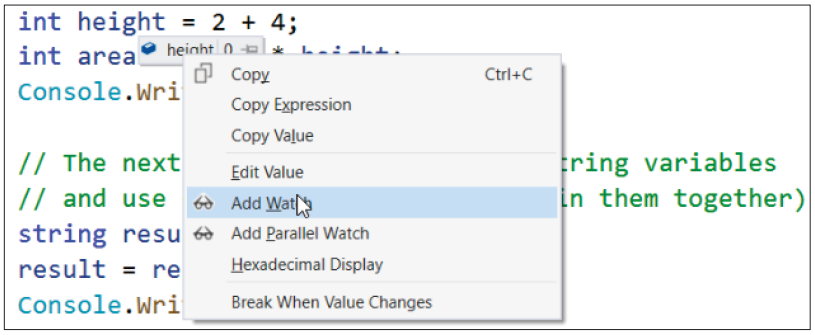
4 The locals window shows values of your variables.

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

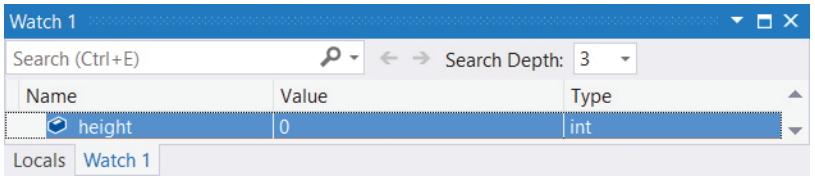


5 Add a watch for the height variable.

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



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



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

5 Step through the rest of the method.

Step over each statement in OperatorExamples. As you step through the method, keep an eye on the Locals or Watch window and watch the values as they change. **Press Alt-Tab** before and after the Console.WriteLine statements to switch back and forth to the Debug Console to see the output.

Use operators to work with variables

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

Equality Operators

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

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

Relational Operators

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

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

Logical Operators

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

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

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

Watch it!

Don't confuse the two equals sign operators!

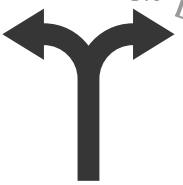
You use one equals sign (=) to set a variable's value, but two equals signs (==) to compare two variables. You won't believe how many bugs in programs—even ones made by experienced programmers!—are caused by using = instead of ==. If you see the IDE complain that you "cannot implicitly convert type 'int'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

Use operators to compare two int variables

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

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

These are the ones you'll use most often.



if statements make decisions

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

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

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

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

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

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

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

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

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

Loops perform an action over and over

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

while loops keep looping statements while a condition is true

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

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

do/while loops run the statements then check the condition

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

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

for loops run a statement after each loop

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

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

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

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

For Loops Up Close

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

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

}
```

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

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

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

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

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

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

Sharpen your pencil

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

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

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

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

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

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

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

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

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

Sharpen your pencil Solution

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

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

Loop #1 executes once

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

Loop #2 executes 7 times.

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

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

Loop #3 executes 8 times.

The statement p = p * 2 executes 3 times.

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

Loop #4 runs forever.

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

Loop #5 is also an infinite loop.

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

Use code snippets help to write loops

Do this!

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

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

```
while (true)
{

}
```

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

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

IDE Tip: Brackets

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

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

```
do
{

} while (true);
```

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

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

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

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

Sharpen your pencil

Let's get some practice working with conditionals and loops. Update the Main method in your console app so it matches the new Man method below, then add the TryAnIf, TryAnIfElse, and TrySomeLoops methods. Before you run your code, try to answer the questions. Then run your code and see if you got them right.

```
static void Main(string[] args)
{
    OperatorExamples();
    TryAnIf();
    TrySomeLoops();
    TryAnIfElse();
}

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

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

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

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

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

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

What does the TryAnIf method write to the console?

.....

.....

What does the TryAnIfElse method write to the console?

.....

What does the TrySomeLoops method write to the console?

.....

We didn't include answers for this exercise in the book. Just run the code and see if you got the console output right.

Some useful things to keep in mind about C# code

- **Don't forget that all your statements need to end in a semicolon.**
  name = "Joe";
- **Add comments to your code by starting a line with two slashes.**
  `// this text is ignored`
- **Use /* and */ to start and end comments that can include line breaks.**
  ```
  /* this comment
   * spans multiple lines */
  ```
- **Variables are always declared with a *name* and a *type*.**
  ```
  int weight;
  // the name is weight and the type is int
  ```
- **Most of the time, extra whitespace is fine.**
  So this: `int j = 1234 ;`
  Is exactly the same as this: `int j = 1234;`
- **If/else, while, do, and for are all about testing conditions.**
  Every loop we've seen so far keeps running as long as a condition is true.

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

Then your loop runs forever.

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

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

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

Game design... and beyond

Mechanics

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

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

I BET THE CONCEPT OF MECHANICS CAN HELP ME WITH ANY KIND OF PROJECT, NOT JUST GAMES.

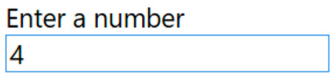
Definitely! Every program has its own kind of mechanics.

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

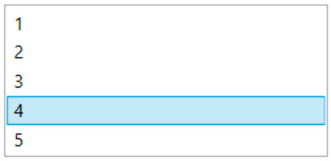
Here's an example. The mechanics of a game determine how hard or easy it is to play. Make Pac Man faster or the ghosts slower and the game gets easier. That doesn't necessarily make it better or worse—just different. And guess what? The same exact idea applies to how you design your classes! You can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

Controls drive the mechanics of your user interfaces

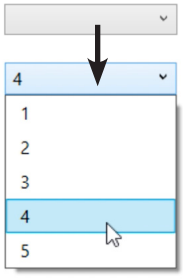
In the last chapter, you built a game using TextBlock and Grid **controls**. But there are a lot of different ways that you can use controls, and the choices you make about what controls to use can really change your app. Does that sound weird? It's actually really similar to the way we make choices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



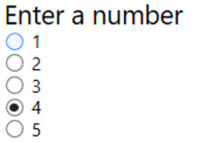
★ A **text box** lets a user enter any text they want. But we need a way to make sure they're only entering numbers and not just any text.



★ A **list box** gives users a way to choose from a list of items. If the list is long, it will show a scroll bar to make it easier for the user to find an item.



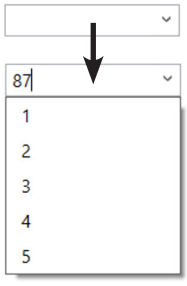
★ A **combo box** combines the behavior of a list box and a text box. It looks like a normal text box, but when the user clicks it a list box pops up underneath it.



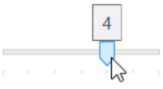
★ **Radio buttons** let you restrict the user's choice. You can use them for numbers if you want, and you can choose how you want to lay them out.

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

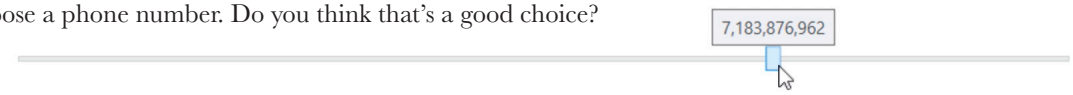
We can borrow the idea of mechanics from video games to understand our options, so we can make great choices for any of our own apps—not just games.



Editable check boxes let the user either choose from a list of items or type in their own value.

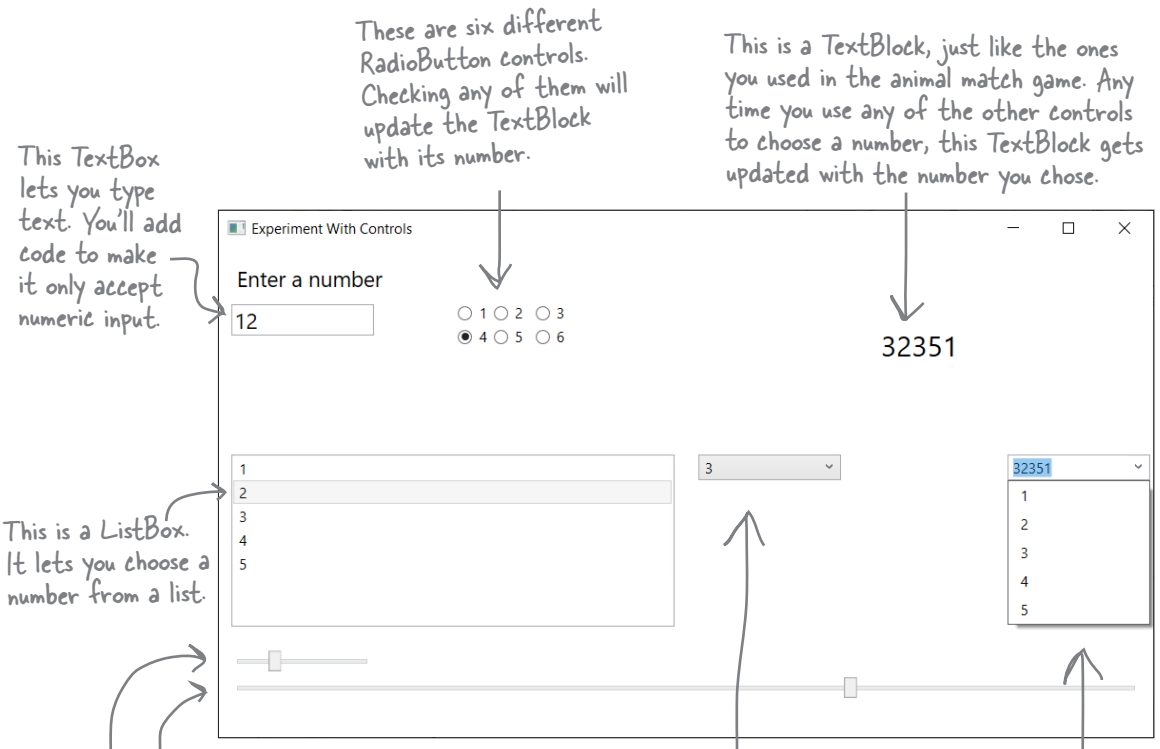


★ The other controls on this page can be used for other types of data, but **sliders** are used exclusively to choose a number. Phone numbers are just numbers, too. So *technically* you could use a slider to choose a phone number. Do you think that's a good choice?



Create a WPF app to experiment with controls

If you've filled out a form on a web page, you've seen the controls we just showed you (even if you didn't know all of their official names). Now let's **create a WPF app** to some practice using those controls. The app will be really simple—the only thing it will do is let the user pick a number, and display the number that was picked.



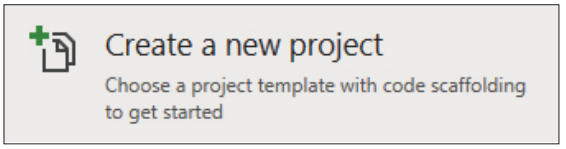
Don't worry about committing the XAML for controls to memory.

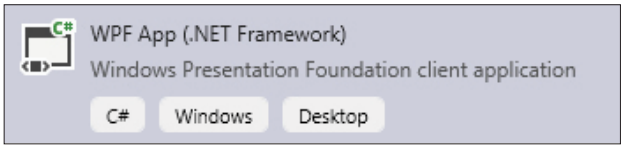
This Do this! and these exercises are all about getting some practice using XAML to build a UI with controls. You can always refer back to it when we use these controls in projects later in the book.

In Chapter 1 you added row and column definitions to the Grid in your WPF app—specifically, you created a grid with five equal-sized rows and four equal-sized columns. You'll do the same for this app. In this exercise, you'll use what you learned about XAML in Chapter 1 to start your WPF app.

Create a new WPF project

Start up Visual Studio 2019 and **create a new WPF project**, just like you did in with your animal matching game in Chapter 1. Choose Create a new project and select WPF App (.NET Core).





Name your project **ExperimentWithControls**.

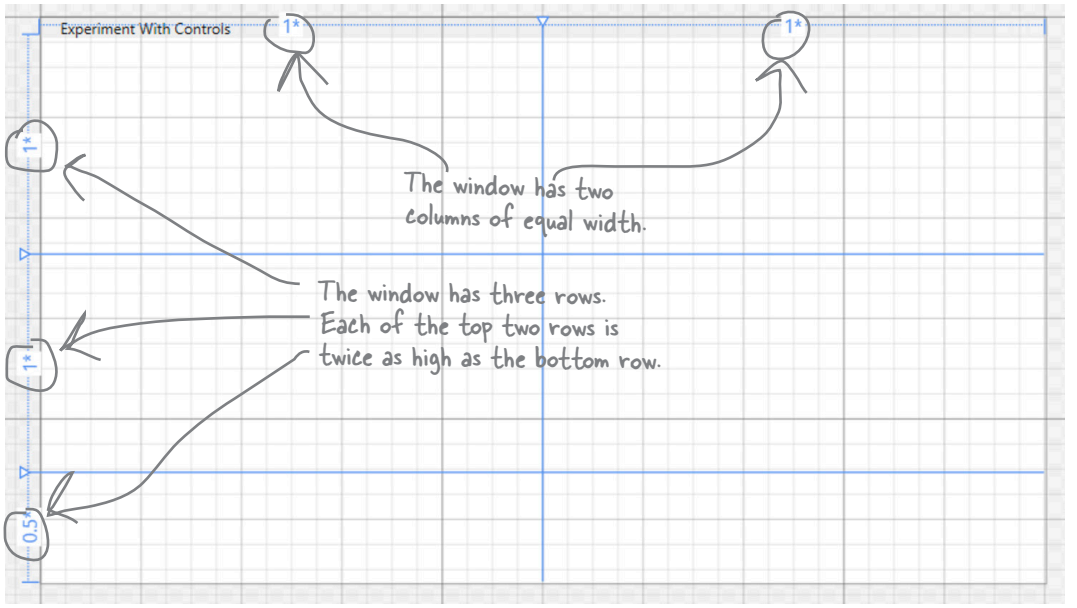
Set the window title

Modify the `Title` property of the <Window> tag to set the title of the window to `Experiment With Controls`.

Add the rows and columns

Add three rows of equal height, and three columns of equal width. You can use the XAML designer to do it just like you did in Chapter 1, or you can type in the XAML by hand.

This is what your window should look like in the designer:



Here's the XAML for your main window. We used a lighter color for the XAML code that Visual Studio created for you and you didn't have to change. You had to change the Title property in the <Window> tag, then add the <Grid.RowDefinitions> and <Grid.ColumnsDefinitions> sections.

```
<Window x:Class="ExperimentWithControls.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:ExperimentWithControls"
        mc:Ignorable="d"
        Title="Experiment With Controls" Height="450" Width="800">
    <Grid>

        <Grid.RowDefinitions>
            <RowDefinition/>
            <RowDefinition/>
            <RowDefinition Height=".5*"/>
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>

    </Grid>
</Window>
```

Change the Title property of the window to set the window title.

Setting the height of the bottom row to .5* causes it to be half as tall as each of the other rows. You could also set the other two row heights to 2* (or you could set the top two to 4* and the bottom to 2*, or the top two to 1000* and the bottom to 500*, etc.)

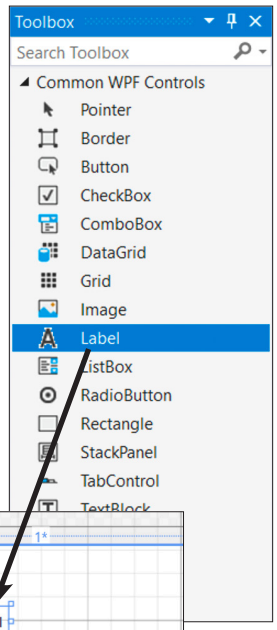
I BET THIS WOULD BE A GREAT TIME TO ADD THE PROJECT TO SOURCE CONTROL...

"Save early, save often."

That's an old saying from a time before video games had an autosave feature, and when you had to stick one of these in your computer to back up your projects... but it's still great advice! Visual Studio makes it easy to add your project to source control and keep it up to date—so you'll always be able to go back and see all the progress you've made.

Add a TextBox control to your app

A **TextBox control** gives your user a box to type text into, so let's add one to your app. But we don't just want a TextBox sitting there without a label, so we'll use a **Label control** (which is a lot like a TextBlock, except it's specifically used to add labels to other controls).

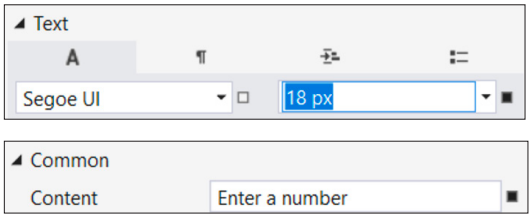


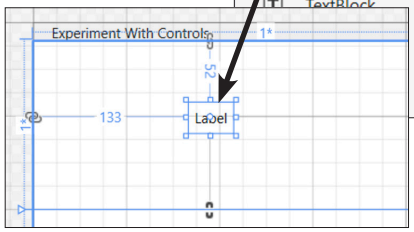
1 Drag a Label out of the Toolbox into the top left cell of the grid

This is exactly how you added TextBlock controls onto your animal match game in Chapter 1, except this time you're doing it with a Label control. It doesn't matter where in the cell you drag it, as long as it's in the upper left cell.

2 Set the text size and content of the Label.

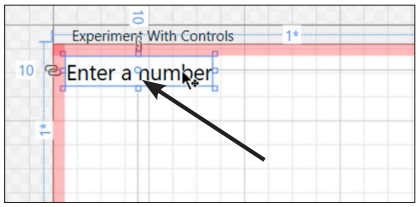
While the Label control is selected, go to the Properties, expand the Text section, and set the font size to **18px**. Then expand the Common section and set the Content to the text `Enter a number`.





3 Drag the Label to the upper left corner of the cell.

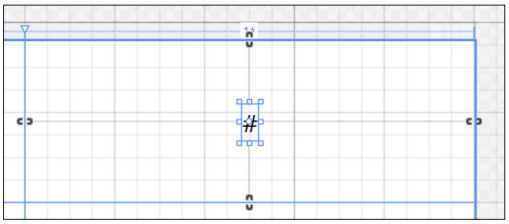
Click on the Label in the designer and drag it to the upper left corner. When it's 10 pixels away from the left or top cell wall, you'll see gray bars appear and it will snap to a 10px margin.



The XAML for your window should now contain Label control:

```
<Label Content="Enter a number" FontSize="18"
       Margin="10,10,0,0" HorizontalAlignment="Left"
       VerticalAlignment="Top"/>
```

In Chapter 1 you added TextBlock controls to many cells in your grid and put a **?** inside each of them. You also gave a name to the Grid control and one of the TextBlock controls. For this project, **add one TextBlock control**, give it the name ***number***, set the text to **#** and font size to **24px**, make **centered** in the **upper right cell** of the grid.

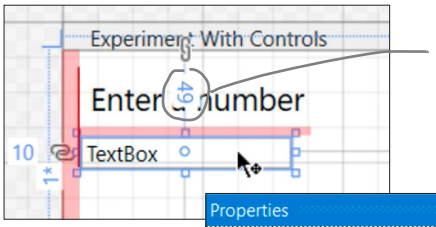


Here's the XAML for the TextBlock that goes in the upper right cell of the grid. You can use the visual designer or type in the XAML by hand. Just make sure your TextBlock has exactly the same properties as this solution—but like earlier, it's okay if your properties are in a different order.

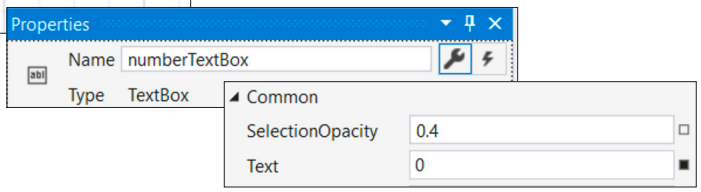
```
<TextBlock x:Name="number" Grid.Column="1" Text="#"  FontSize="24"
         HorizontalAlignment="Center" VerticalAlignment="Center" TextWrapping="Wrap"/>
```

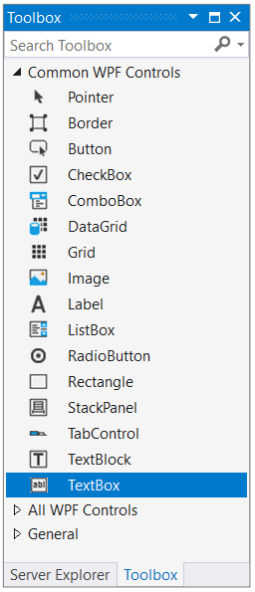
4 Drag a TextBox into the top left cell of the grid.

Your app will have a TextBox positioned just underneath the Label so the user can type in numbers. Drag it so it's on the left side and just under the TextBox—the same gray bars will appear to position it just under the Label with a 10px left margin. Then set its name to **numberTextBox**, font size to **18px**, and text to **0**.

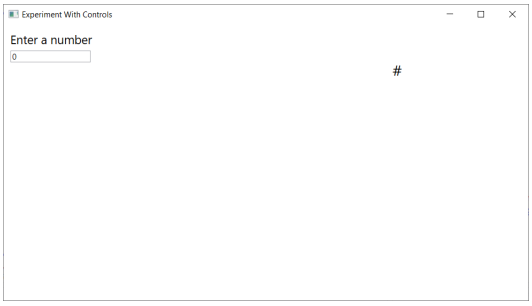


When you use the gray bars to position a control, it snaps into position with a 10px margin below the control above it. You can see the top and left margins change as you drag the control.





Your window should now look like this →



And the XAML code that appears inside the `<Grid>` after the row and column definitions and before the `</Grid>` should look like this ↓

Remember, it's okay if your properties are in a different order or if there are line breaks.

```
<Label Content="Enter a number" FontSize="18" Margin="10,10,0,0"
       HorizontalAlignment="Left" VerticalAlignment="Top"/>

<TextBox x:Name="numberTextBox" FontSize="18" Margin="10,49,0,0" Text="0" Width="120"
         HorizontalAlignment="Left" TextWrapping="Wrap" VerticalAlignment="Top" />

<TextBlock x:Name="number" Grid.Column="1" Text="#" FontSize="24"
          HorizontalAlignment="Center" VerticalAlignment="Center" TextWrapping="Wrap" />
```

Add C# code to update the TextBlock

In Chapter 1 you added **event handlers**—methods that are called when a certain event is **raised** (sometimes we say the event is **triggered** or **fired**)—to handle mouse clicks in your animal match game. Now we'll add an event handler to the code-behind that's called any time the user enters text into the TextBox and copies that text to the TextBlock from the mini-exercise that you added to the upper right cell.

When you double-click on a TextBox control, the IDE adds an event handler for the TextChanged event that's called any time the user changes its text. Double-clicking on other types of controls add other event handlers—and some (like TextBlock) don't add any event handlers at all.

5 Double-click on the TextBox control to add the method.

As soon as you double-click on the TextBox, the IDE will **automatically add a C# event handler method** hooked up to its TextChanged event. It generates an empty method and gives it a name that consists of the name of the control (`numberTextBox`) followed by an underscore and the name of the event being handled: numberTextBox_TextChanged.

```
private void numberTextBox_TextChanged(object sender, TextChangedEventArgs e)
{

}
```

6 Add code to the new TextChanged event handler.

Any time the user enters text into the TextBox, we want the app to copy it into the TextBlock that you added to the upper right cell of the grid. Since you gave the TextBlock a name (`number`) and you also gave the TextBox a name (`numberTextBox`), you just need one line of code to copy its contents:

```
private void numberTextBox_TextChanged(object sender, TextChangedEventArgs e)
{
    number.Text = numberTextBox.Text;
}
```

This line of code sets the text in the TextBlock so it's the same as the text in the TextBox, and it gets called any time the user changes the text in the TextBox.

7 Run your app and try out the TextBox.

Use the Start Debugging button (or choose Start Debugging (F5) from the Debug menu) to start your app, just like you did with the animal match game in Chapter 1. (If the runtime tools appear, you can disable them just like you did in Chapter 1.) Type any number into the TextBox and it will get copied.

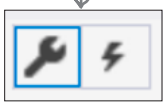
When you type a number into the TextBox, the TextChange event handler copies it to the TextBlock.

But something's wrong—you can enter any text into the TextBox, not just numbers!

There has to be a way to only allow the user to enter only numbers! How do you think we'll do that?

Add an event handler that only allows number input

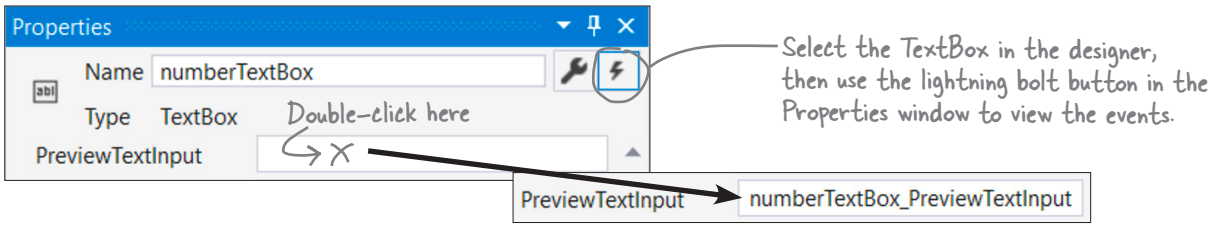
The wrench button in the upper right corner of the Properties window shows you the properties for the selected control. The lightning bolt button switches to show its event handlers.



When you added the MouseDown event to your TextBlock in Chapter 1, you used the buttons in the upper right corner of the Properties window to switch between properties and events. Now you'll do the same thing, except this time you'll use the **PreviewTextInput event** to only accept input that's made up of numbers, and reject any input that isn't a number.

If your app is currently running, stop it. Then go to the designer, click on the TextBox to select it, and switch the Properties window to show you its events. Then scroll down and **double-click inside the box next to PreviewTextInput** to make the IDE generate an event handler method.

Do this!



Your new event handler method will have one statement in it:

```
private void numberTextBox_PreviewTextInput(object sender, TextCompositionEventArgs e)
{
    e.Handled = !int.TryParse(e.Text, out int result);
}
```

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

Here's how this event handler works:

1. The event handler is called when the user enters text into the TextBox, but ***before*** the TextBox is updated.
2. It uses a special method called `int.TryParse` to check if the text that the user entered is a number.
3. If the user entered a number, it sets `e.Handled` to true, which tells WPF to ignore the input.

Before you run your code, go back and look at the XAML tag for the TextBox:

```
<TextBox x:Name="numberTextBox" FontSize="18" Margin="10,49,0,0" Text="0" Width="120"
         HorizontalAlignment="Left" TextWrapping="Wrap" VerticalAlignment="Top"
         TextChanged="numberTextBox_TextChanged"
         PreviewTextInput="numberTextBox_PreviewTextInput" />
```

Now it's hooked up to two event handlers: the TextChange event is hooked up to an event handler method called numberTextBox_TextChanged, and right below it the PreviewTextInput event is hooked up to a method called numberTextBox_PreviewTextInput.

Now run your app again. Oops! Something went wrong—it threw an exception.

```
1 reference
private void numberTextBox_TextChanged(object sender, TextChangedEventArgs e)
{
    number.Text = numberTextBox.Text;
}

1 reference
private void numberTextBox_PreviewText
{
    e.Handled = !int.TryParse(e.Text,
}
```

Exception User-Unhandled

System.NullReferenceException: 'Object reference not set to an instance of an object.'

number was null.

View Details | Copy Details | Start Live Share session...

Exception Settings

Being a great developer is about more than just writing lines of code! Here's another exception to sleuth out, just like you did in Chapter 1 – tracking down and fixing problems like this is a really important programming skill.

Now take a look at the bottom of the IDE. It has an Autos window that shows you any defined variables.

The number TextBox says "null" – and we see that same word in the NullReferenceException.

Autos

Search (Ctrl+E) Search Depth: 3

Name	Value	Type
e	{System.Windows.Controls.TextChangedEventArgs}	Syst...
number	null	Syst...
numberTextBox	{System.Windows.Controls.TextBox: 0}	Syst...

So what's going on—***and, more importantly, how do we fix it?***

Sleuth it out

The Autos window is showing you the variables used by the statement that threw the exception: `number` and `numberTextBox`. The value of numberTextBox is *{System.Windows.Controls.TextBox: 0}*, and that's what a healthy TextBox looks like in the debugger. But the value of number—the TextBlock that you're trying to copy the text to—is **null**. You'll learn more about what null means later in the book.

But here's the all-important clue: the IDE is telling you is that the **number TextBlock is not initialized**.

The problem is that the XAML for the TextBox includes `Text="0"`, so when the app starts running it initializes the TextBox and tries to set the text. That fires the TextChanged event handler, which tries to copy the text to the TextBlock. But the TextBlock is still null, so the app throws an exception.

So all we need to do to fix the bug is to make sure the TextBlock is initialized before the TextBox. When a WPF app starts up, the controls are **initialized in the order they appear in the XAML**. So you can fix the bug by ***changing the order of the controls in the XAML***.

Moving the TextBlock tag in the XAML so it's above the TextBox causes the TextBlock to get initialized first.

Swap the order of the TextBlock and TextBox controls so the TextBlock appears above the TextBox:

```
<Label Content="Enter a number" ... />
<TextBlock x:Name="number" Grid.Column="1" ... />
<TextBox x:Name="numberTextBox" ... />
```

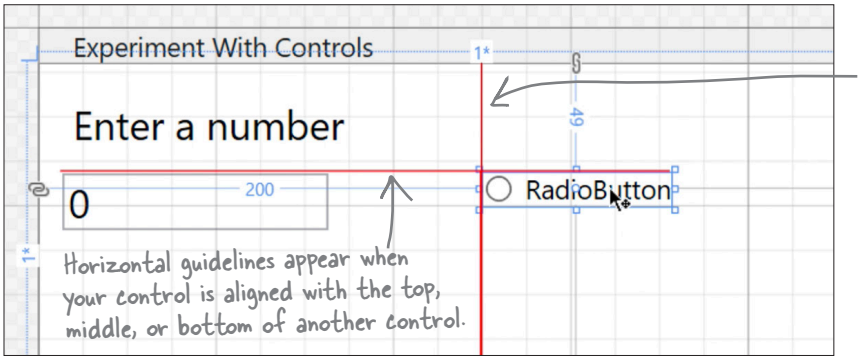
Select the TextBlock tag in the XAML editor move it above the TextBox so it gets initialized first.

The app should still look exactly the same in the designer—which makes sense, because it still has the same controls. Now run your app again. This time it starts up, and the TextBox now only accepts numeric input.

Add the rest of the XAML controls for the ExperimentWithControls app: radio buttons, a list box, two different kinds of combo boxes, and two sliders. Each of the controls will update the TextBlock in the upper right cell of the grid.

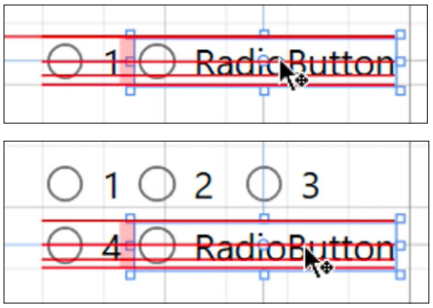
Add radio buttons to the upper left cell next to the TextBox

Drag a RadioButton out of the Toolbox and into the top left cell of the grid. Then drag it until its left side is aligned with the center of the cell and the top is aligned with the top of the TextBox. As you drag controls around the designer, **guidelines** appear to help you line everything up neatly, and the control will snap to those guidelines.



Expand the Common section of the Properties window and set the Content of the RadioButton control to 1.

Next, add five more RadioButton controls out of the Toolbox, align them, and set their Content properties. But this time, don't drag them out of the Toolbox. Instead, **click on RadioButton in the Toolbox, then click inside the cell**. *(The reason you're doing that is if you have a RadioButton selected and then drag another control out of the Toolbox, the IDE will newt the new control inside of the RadioButton. We'll learn about nesting controls later in the book.)*

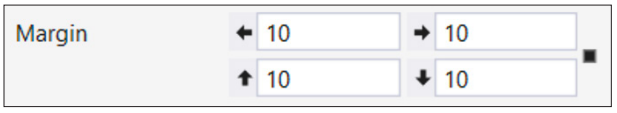


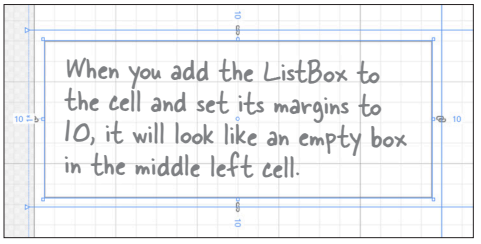
When you add each radio button, you can use the bars and guidelines to align it with the others.

Still seeing event handlers and not properties in the Properties window? Use the wrench button to display properties again—and if you used the search box, make sure you clear it.

Add a list box to the middle left cell of the grid

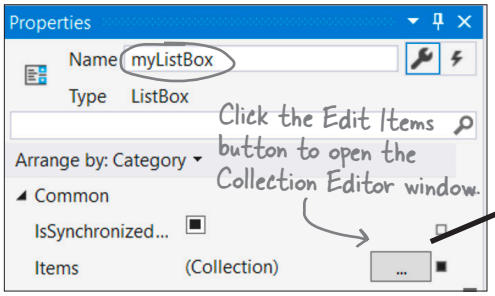
Click on ListBox in the Toolbox, then click inside the middle left cell to add the control. Use the Layout section set all of its margins to 10.

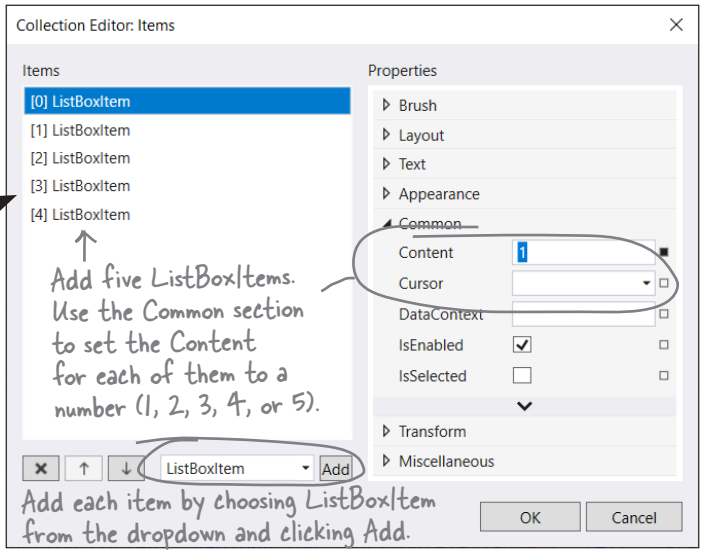




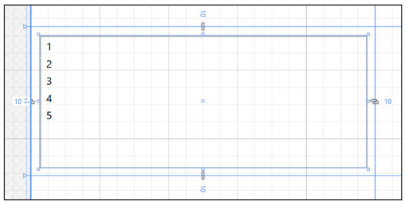
Name your ListBox myListBox and add ListBoxItems to it

The purpose of the ListBox is to let the user choose a number. We'll do that by adding **items** to the list. Select the ListBox, expand Common in the properties window, and **click the Edit Items button** next to Items (...). **Add five ListBoxItem items** and set their Content values to numbers 1 to 5.



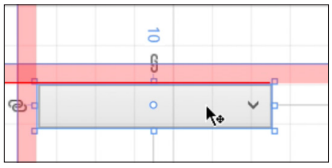


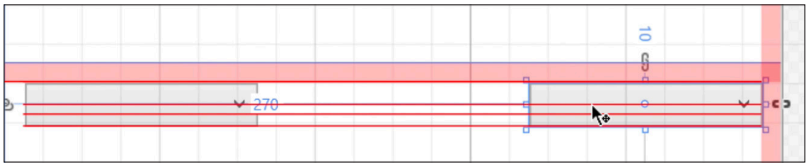
Your ListBox should now look like this:



Add two different ComboBoxes to the middle right cell in the grid

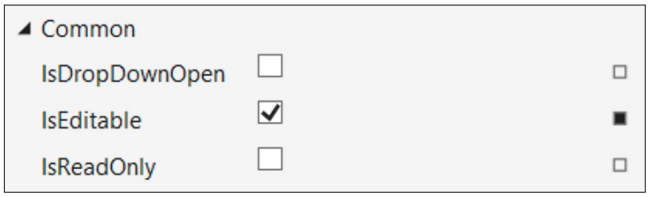
Click on ComboBox in the Toolbox, then click inside the middle right cell to **add a ComboBox and name it readOnlyComboBox**. Drag it to the upper right corner and use the gray bars to give it left and top margins of 10. Then **add another ComboBox named editableComboBox** to the same cell and align it with the upper right corner.





Use the Collection Editor window to **add the same ListBoxItems** with numbers 1, 2, 3, 4, and 5 to ***both*** ComboBoxes—so you'll need to do it for the first ComboBox, then the second ComboBox.

Finally, **make the ComboBox on the right editable** by expanding the Common section in the Properties window and checking IsEditable. Now the user can type in their own number into that ComboBox.



The editable ComboBox looks different to let users know they can either type in their own value or choose one from the list.

Here's the XAML for the RadioButton, ListBox, and two ComboBox controls that you added in the exercise. This XAML should be at the very bottom of the grid contents—you should find them just above the closing `</Grid>` tag. And just like with any other XAML you've seen so far, it's okay if the properties for a tag are in a different order in your code, or if you have different line breaks.

```
<RadioButton Content="1" Margin="200,49,0,0"
             HorizontalAlignment="Left" VerticalAlignment="Top"/>
<RadioButton Content="2" Margin="230,49,0,0"
             HorizontalAlignment="Left" VerticalAlignment="Top"/>
<RadioButton Content="3" Margin="265,49,0,0"
             HorizontalAlignment="Left" VerticalAlignment="Top"/>
<RadioButton Content="4" Margin="200,69,0,0"
             HorizontalAlignment="Left" VerticalAlignment="Top"/>
<RadioButton Content="5" Margin="230,69,0,0"
             HorizontalAlignment="Left" VerticalAlignment="Top"/>
<RadioButton Content="6" Margin="265,69,0,0"
             HorizontalAlignment="Left" VerticalAlignment="Top"/>

<ListBox x:Name="myListBox" Grid.Row="1" Margin="10,10,10,10">
    <ListBoxItem Content="1"/>
    <ListBoxItem Content="2"/>
    <ListBoxItem Content="3"/>
    <ListBoxItem Content="4"/>
    <ListBoxItem Content="5"/>
</ListBox>

<ComboBox x:Name="readOnlyComboBox" Grid.Column="1" Margin="10,10,0,0" Grid.Row="1"
          HorizontalAlignment="Left" VerticalAlignment="Top" Width="120">
    <ListBoxItem Content="1"/>
    <ListBoxItem Content="2"/>
    <ListBoxItem Content="3"/>
    <ListBoxItem Content="4"/>
    <ListBoxItem Content="5"/>
</ComboBox>

<ComboBox x:Name="editableComboBox" Grid.Column="1" Grid.Row="1" IsEditable="True"
   HorizontalAlignment="Left" VerticalAlignment="Top" Width="120" Margin="270,10,0,0">
    <ListBoxItem Content="1"/>
    <ListBoxItem Content="2"/>
    <ListBoxItem Content="3"/>
    <ListBoxItem Content="4"/>
    <ListBoxItem Content="5"/>
</ComboBox>
```

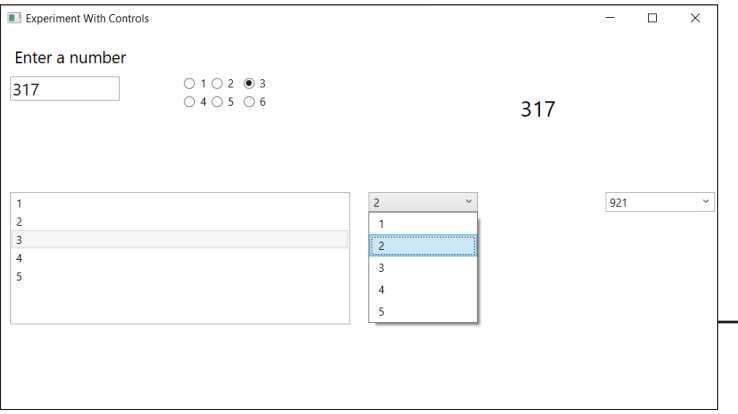
The IDE added the margin and alignment properties to each RadioButton control when you dragged it into place.

When you use the Collection Editor window to add ListBoxItem items to a ListBox or ComboBox, it creates a closing `</ListBox>` or `</ComboBox>` tag and adds `<ListBoxItem>` tags between the opening and closing tags.

Make sure you gave your ListBox and two ComboBoxes the right names. You'll use them in the C# code.

The only difference between the two ComboBox controls is the IsEditable property.

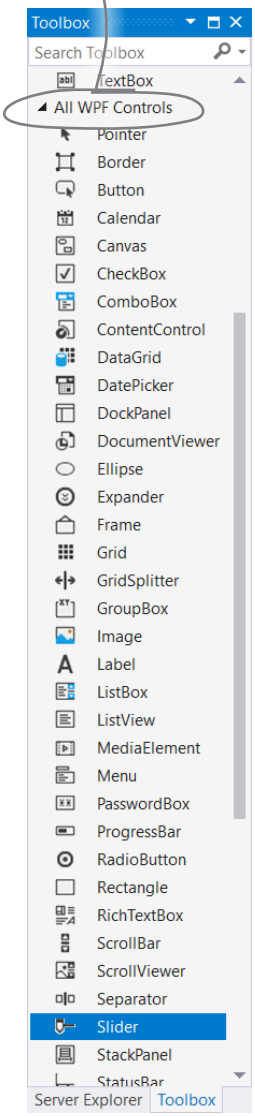
When you run your program, it should look like this. You can use all of the controls, but only the TextBox actually updates the value. in the upper right.



Add sliders to the bottom row of the grid

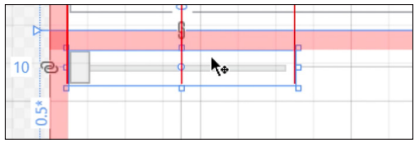
Let's add two sliders to the bottom row and then hook up their event handlers so they update the TextBlock in the upper right.

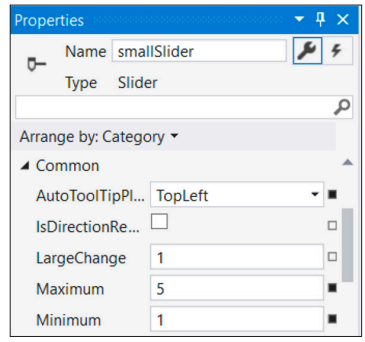
To find the Slider control in the Toolbox, you'll need to expand the "All WPF Controls" section and scroll almost all the way to the bottom.



1 Add a slider to your app.

Drag a Slider out of the Toolbox and into the lower right cell. Drag it to the upper right corner of the cell and use the gray bars to give it left and top margins of 10.





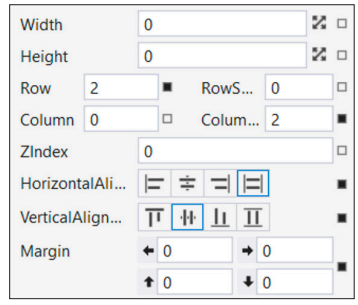
Use the Common section of the Properties window to set AutoToolTipPlacement to **`TopLeft`**, Maximum to **`5`**, Minimum to **`1`**, and give it the name **`smallSlider`**. Then double-click on the slider to add this event handler:

```
private void smallSlider_ValueChanged(
          object sender, RoutedPropertyChangedEventArgs<double> e)
{
    number.Text = smallSlider.Value.ToString("0");
}
```

The value of the Slider control is fractional number with a decimal point. This "0" converts it to a whole number.

2 Add a ridiculous slider to choose phone numbers.

There's an old saying: *"Just because an idea is terrible and also maybe stupid, that doesn't mean you shouldn't do it."* So let's do something that's just a bit stupid: add a slider to select phone numbers.



Drag another slider into the bottom row. Use the Layout section of the Properties window to **reset its width**, set its ColumnSpan to **`2`**, set all of its margins to **`10`**, and set its vertical alignment to **`Center`** and horizontal alignment to **`Stretch`**. Then use the Common section to set AutoToolTipPlacement to **`TopLeft`**, Minimum **`1111111111`**, Maximum **`9999999999`**, and value **`7183876962`**. Give it the name **`bigSlider`**. Then double-click on it and add this ValueChanged event handler:

```
private void bigSlider_ValueChanged(
          object sender, RoutedPropertyChangedEventArgs<double> e)
{
    number.Text = bigSlider.Value.ToString("000-000-0000");
}
```

The zeroes and hyphens causes the method to format any 10-digit number as a US phone number.

Add C# code to make the rest of the controls work

We want each of the controls in your app to do the same thing: update the TextBlock in the upper right cell with a number, so when you check one of the radio buttons or pick an item from a ListBox or ComboBox, the TextBlock is updated with whatever value you chose.

① Add a Checked event handler to the first RadioButton control.

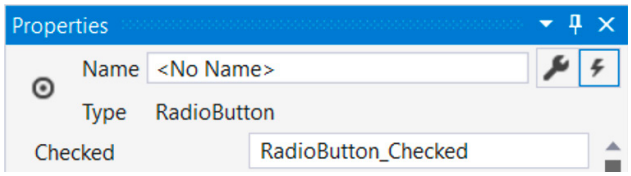
Double-click on the first RadioButton. The IDE will add a new event handler method called RadioButton_Checked (since you never gave the control a name, it just uses the type of control to generate the method). Add this line of code:

```
private void RadioButton_Checked(object sender, RoutedEventArgs e)
{
    if (sender is RadioButton radioButton)
       number.Text = radioButton.Content.ToString();
}
```

This statement uses the `is` keyword, which you'll learn about in Chapter 7. For now, just carefully enter it exactly like it appears on the page (and do the same for the other event handler method, too).

② Make the other RadioButtons use the same event handler.

Look closely at the XAML for the RadioButton that you just modified. The IDE added this property: `Checked="RadioButton_Checked"` – this is exactly like how the other event handlers were hooked up. **Copy this property to the other RadioButton tags** so they all have identical Checked properties—and ***now they're all connected to the same Checked event handler***. You can use the Events view in the Properties window to check that each RadioButton is hooked up correctly.



If you switch the Properties window to the Events view, you can select any of the RadioButton controls and make sure they all have the Checked event hooked up to the RadioButton_Checked event handler.

③ Make the ListBox update the TextBlock in the upper right cell.

When you did the exercise, you named your ListBox myListBox. Now you'll add an event handler that fires any time the user selects an item and uses the name to get the number that the user selected.

Double-click inside the *empty* space in the ListBox *below* the items to make the IDE will add an event handler method for the SelectionChanged event. Add this statement to it:

```
private void myListBox_SelectionChanged(
       object sender, SelectionChangedEventArgs e)
{
    if (myListBox.SelectedItem is ListBoxItem listBoxItem)
        number.Text = listBoxItem.Content.ToString();
}
```

Make sure you click on the empty space below the list items. If you click on an item, it will add an event handler for that item and not for the entire ListBox. You can also use the Properties window to add a SelectionChanged event.

4 Make the read-only combo box update the TextBlock.

Double-click on the read-only ComboBox to make Visual Studio add an event handler for the SelectionChanged event, which is raised any time a new item is selected in the ComboBox. Here's the code—it's really similar to the code for the ListBox:

```
private void readOnlyComboBox_SelectionChanged(
                  object sender, SelectionChangedEventArgs e)
{
    if (readOnlyComboBox.SelectedItem is ListBoxItem listBoxItem)
        number.Text = listBoxItem.Content.ToString();
}
```

5 Make the editable combo box update the TextBlock.

An editable combo box is like a cross between a ComboBox and a TextBox. You can choose items from a list, but you can also type in your own text. And since it works like a TextBox, we can add a PreviewTextInput event handler to make sure the user can only type numbers, just like we did with the TextBox. In fact, you even **reuse the same event handler** that you already added for the TextBox.

Go to the XAML for the editable ComboBox, put your cursor just before the closing caret **>** and **start typing *PreviewTextInput***. An IntelliSense window will pop up to help you complete the event name. Then **add an = equals sign**—as soon as you do, the IDE will prompt you to either choose a new event handler or select the one you already added. Choose the existing event handler.

```
<ComboBox x:Name="editableComboBox" Grid.Column="1" Grid.Row="1" IsEditable="True"
    HorizontalAlignment="Left" VerticalAlignment="Top" Width="120" Margin="270,10,0,0"
    PreviewTextInput="" >
                          <New Event Handler>
                          numberTextBox_PreviewTextInput
```

The previous event handlers used the list items to update the TextBlock. But users can enter any text they want into an editable ComboBox, so this time you'll **add a different kind of event handler**.

Edit the XAML again. This time, **type TextBoxBase.**—as soon as you type the period, the autocomplete will give suggestions. Choose **TextBoxBase.TextChanged** and type an equals sign. Now choose <New Event Handler> from the dropdown.

```
TextBoxBase.>
            SelectionChanged
            TextChanged
```

```
TextBoxBase.TextChanged="">
                          <New Event Handler>
                          numberTextBox_TextChanged
```

The IDE will add a new event handler to the code-behind. Here's the code for it:

```
private void editableComboBox_TextChanged(object sender, TextChangedEventArgs e)
{
    if (sender is ComboBox comboBox)
        number.Text = comboBox.Text;
}
```

Now run your program. All of the controls should work. Great job!

Controls give you the flexibility to make things easy for your users.

When you're building the UI for an app, there are so many choices that you make: what controls to use, where to put each one, what to do with their input. Picking one control instead of another gives your users an *implicit* message about how to use your app. For example, when you see a set of radio buttons, you know that you need to pick from a small set of choices, while an editable combo box tells you that there your choices are nearly unlimited. So don't think of UI design as a matter of making "right" or "wrong" choices. Instead, think of it as your way to make things as easy as possible for your users.

BULLET POINTS

- C# programs are organized into **classes**, classes contain **methods**, and methods contain **statements**.
- Each class belongs to a **namespace**. Some namespaces (like System.Collections.Generic) contain .NET classes.
- Classes can contain **fields**, which live outside of methods. Different methods can access the same field.
- When a method is marked **`public`** that means it can be called from other classes.
- **.NET Core console apps** are cross-platform programs that don't have a graphical user interface.
- The IDE **builds** your code to turn it into a **binary**, which is a file that can be executed.
- If you have a cross-platform .NET Core console app, you can use the **`dotnet`** command line program to **build binaries** for different operating systems.
- The **Console.WriteLine method** writes a string to the console output.
- Variables need to be **declared** before they can be used. You can set a variable's value at the same.
- The Visual Studio debugger lets you **pause your app** and inspect the value of variables.
- Controls **raise events** for lots of different things that change: mouse clicks, selection changes, text entry. Sometimes people say events are **triggered** or **fired**, which is the same as saying that they're raised.
- **Event handlers** are methods that are called when an event is raised to respond to—or **handle**—the event.
- TextBox controls can use the **PreviewTextInput event** to accept or reject text input.
- A **slider** is a great way to get number input, but a terrible way to choose a phone number.

Unity Lab

Write C# Code for Unity

Unity isn't *just* a powerful, cross-platform engine and editor for building 2D and 3D games and simulations. It's also a **great way to get practice writing C# code**.

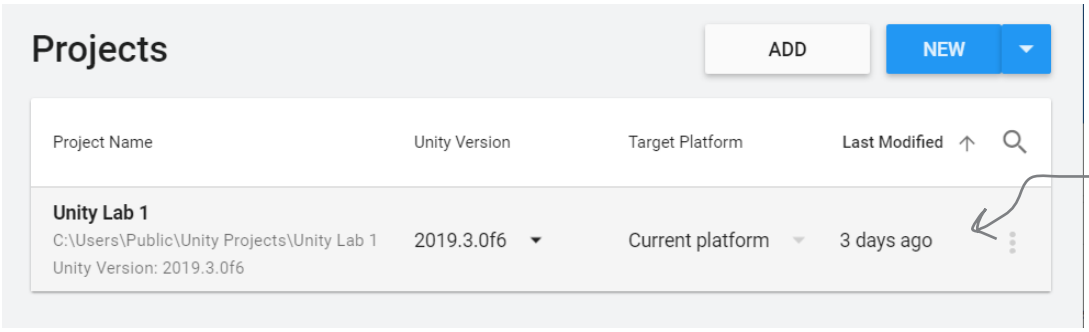
In the last Unity Lab you learned how to navigate around Unity and your 3D space, and started to create and explore GameObjects. Now it's time to write some code to take control of your GameObjects. The whole goal of that Lab was to get you oriented in the Unity editor (and give you an easy way to remind yourself of how to navigate around it if you need it).

In this Unity Lab, you'll start writing code to control your GameObjects. You'll write C# code to explore concepts you'll use in the rest of the Unity Labs, starting with adding a method that rotates the 8 Ball GameObject that you created in the last Unity Lab, and. And you'll start using the Visual Studio debugger with Unity to sleuth out problems in your games.

C# scripts add behavior to your GameObjects

Now that you can add a GameObject to your scene, you need a way to make it, well, do stuff. And that's where your C# skills come in. Unity uses **C# scripts** to define the behavior of everything in the game.

This Unity lab will introduce tools that you'll use to work with C# and Unity. You're going to build a simple "game" that's really just a little bit of visual eye candy: you'll make your 8 ball fly around the scene. Start by going to Unity Hub and **opening the same project** that you created in the first Unity Lab.



This Unity Lab picks up where the first one left off, so go to Unity Hub and open the project you created in the last Lab.

Here's what you'll do next:

1. **Attach a C# script to your GameObject.** You'll add a Script component to your Sphere GameObject. When you add it, Unity will create a simple class for you. You'll modify that class so that it drives the 8 ball sphere's behavior.

2. **Use Visual Studio to edit the script.** Remember how you set the Unity editor preferences to make Visual Studio the script editor? That means you can just double-click on the script in the Unity editor and it will open up in Visual Studio.

3. **Play your game in Unity.** There's a Play button at the top of the screen. When you press it, it starts executing all of the scripts attached to the GameObjects in your scene. You'll use that button to run the script that you added to the Sphere.

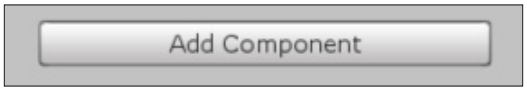
← The Play button does not save your game! So make sure you save early and save often. A lot of people get in the habit of saving the scene every time they run the game.

4. **Use Unity and Visual Studio together to debug your script.** You've already seen how valuable the Visual Studio debugger is when you're trying to track down problems in your C# code. That's why Unity and Visual Studio work together seamlessly to let add breakpoints, use the Locals window, and work with the other familiar tools in the Visual Studio debugger while your game is running.

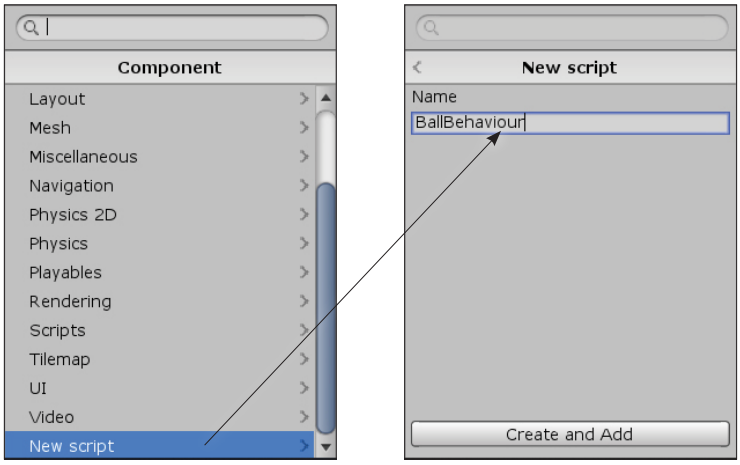
Add a C# script to your GameObject

Unity is more than an amazing platform for building 2D and 3D games. Many people use it for artistic work, data visualization, augmented reality, and more. But it's especially valuable to you, as a C# learner, because you can write code to control everything that you see in a Unity game. That makes Unity ***a great tool for learning and exploring C#***.

Let's start using C# and Unity right now. Make sure the Sphere GameObject is selected, then **click the Add Component button** at the bottom of the Inspector window.



When you click it, Unity pops up a window with all of the different kinds of components that you can add—and there are ***a lot*** of them. **Choose New Script** to add a new C# Script to your Sphere GameObject. You'll be prompted for a name. **Name your script `BallBehaviour`**.

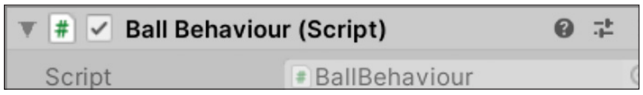


Watch it!

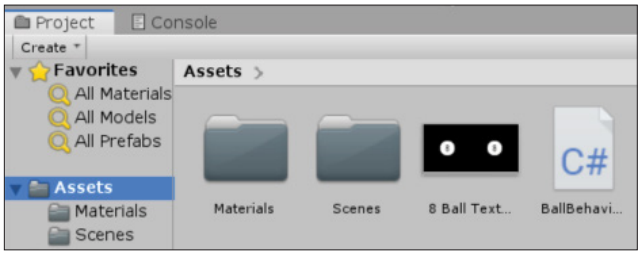
Unity code uses British spelling.

*If you're American (like us), or if you're used to the U.S. spelling of the word **behavior**, you'll need to be careful when you work with Unity scripts because the class names often feature the British spelling **behaviour**.*

Click the Create and Add button to add the script. Once you do, you'll see a component called *Ball Behaviour (Script)* in the Inspector window.



You'll also see the C# script in the Project window.



The Project window gives you a folder-based view of your project. Your Unity project is made up of files: media files, data files, C# scripts, textures, and more. Unity calls these files assets. The Project window was displaying a folder called "Assets" when you right-clicked inside it to import your texture, so Unity added it to that folder.

Did you notice a folder called Materials appeared in the Project window as soon as you dragged the 8 ball texture onto your sphere?

Write C# code to rotate your sphere

At the beginning of this Lab you told Unity to use Visual Studio as its external script editor. So go ahead and **double-click on your new C# script**. When you do, ***Unity will open your script in Visual Studio***. Your C# script contains a class called BallBehaviour with two empty methods called `Start` and `Update`.

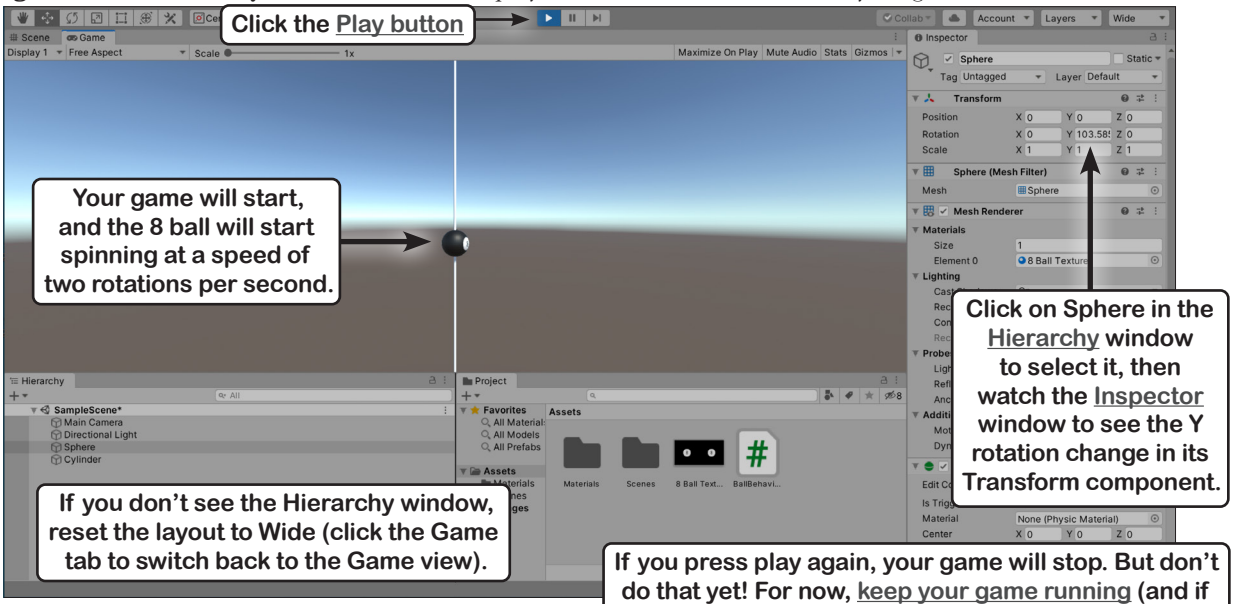
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BallBehaviour : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

The Hierarchy window shows you a list of every GameObject in the current scene. When Unity created your project, it added a scene called SampleScene with a camera and a light. You added a sphere to it, so your Hierarchy window will show all of those things.

Here's a line of code that will rotate your sphere. **Add it to your Update method**:

```
transform.Rotate(Vector3.up, 180 * Time.deltaTime);
```

Now **go back to the Unity editor** and click the play button in the toolbar to start your game:



Your Code Up Close

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BallBehaviour : MonoBehaviour
{
  // Start is called before the first frame update
  void Start()
  {

  }

  // Update is called once per frame
  void Update()
  {
     transform.Rotate(Vector3.up, 180 * Time.deltaTime);
  }
}
```

When Unity created the file with the C# script, it added using lines the top so it can use code in the UnityEngine namespace and other commonly used namespaces.

A frame is a fundamental concept of animation. Unity draws one still frame, then very quickly draws the next one very quickly, and your eye interprets these frames as movement. Unity calls the Update method before for every GameObject before each frame so it can move, rotate, or make any other changes that it needs to make. A faster computer will run at a higher frame rate—or number of frames per second (FPS)—than a slower one.

The transform.Rotate method causes a GameObject to rotate. The first parameter is the axis to rotate around. In this case, your code used Vector3.up, which tells it to rotate around the Y-axis. The second parameter is the number of degrees to rotate.

Different computers will run your game at different frame rates. If it's running at 30 FPS, we want one rotation every 60 frames. If it's running at 120 FPS, it should rotate once every 240 frames. Your game's frame rate may even change if it needs to run more or less complex code.

That's where the special Time.deltaTime value comes in handy. Every time the Unity engine calls a GameObject's Update method—once per frame—it sets Time.deltaTime to the fraction of a second since the last frame. Since want our ball to do a full rotation every two seconds, or 180 degrees per second, all we need to do is multiply it by Time.deltaTime to make sure that it rotates exactly as much as it needs to for that frame.

Inside your Update method, multiplying any value by Time.deltaTime turns it into that value per second.

Add a breakpoint and debug your game

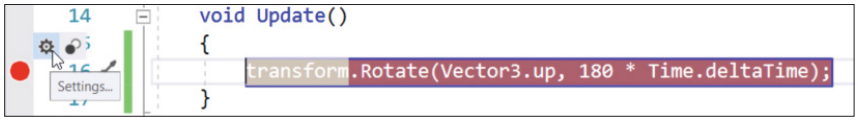
You can **attach Visual Studio to Unity** to debug your game in Visual Studio. Your code is running inside the Unity editor, and when Visual Studio is attached to another process running on your computer, attaching to that process lets you use the debugger to place breakpoints and watch variables *on the running program*. Let's explore how that works.

Find the debug button in the Visual Studio toolbar—it now says "Attach to Unity" (

). When you're editing a Unity project, ***Visual Studio will play the game in Unity when you start the debugger***.

Start the debugger—press the debug button or choose Start Debugging (F5) from the Debug menu, just like before. Since your game is already running, Visual Studio will attach to the game currently in progress.

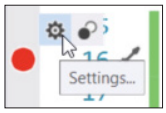
Next, **add a breakpoint** on the line that you added to the Update method.



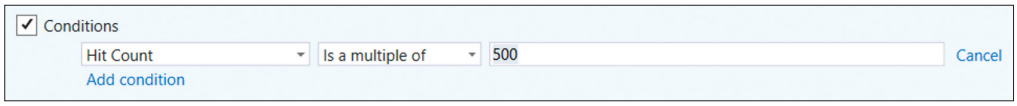
Congratulations, you're now debugging a game!

Since Visual Studio is attached to Unity, when you add the breakpoint it ***breaks immediately***.

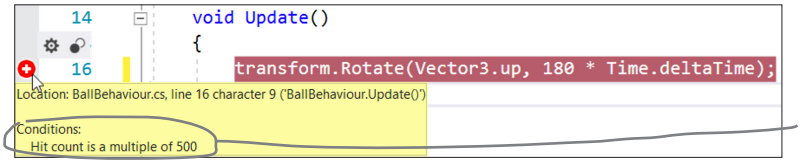
Next, you'll set a **Hit Count condition** to make your breakpoint stop only after it's been run 500 times, then skip another 500 frames before stopping. Hover over the red breakpoint dot in the left margin of the code editor. You'll see a gear for settings, and dots to enable and disable the breakpoint.



Click on the Settings gear to bring up the breakpoint settings. **Check the Conditions box**, choose **Hit Count**, and make the breakpoint trigger when the hit count **is a multiple of 500**.



When you close the settings window, a white plus appears inside the breakpoint dot () to show that it has a condition.



When your breakpoint has conditions, hover over it to see a tooltip window that describes those conditions.

Now the breakpoint will only pause the game every 500 times the Update method is run—or every 500 frames. So if your game is running at 60FPS, that means when you press Continue, the game will run for little over 8 seconds before it breaks again. So **press Continue, then switch back to Unity** and watch the ball spin until the breakpoint breaks.

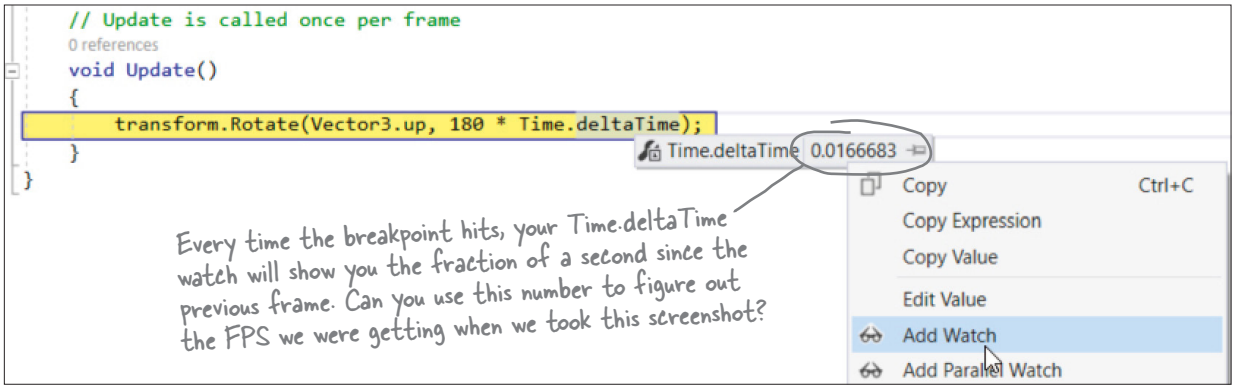
Having trouble getting the Visual Studio debugger to attach to Unity?

Unity and Visual Studio are built to work together! But you need to do a little configuration to make sure it works. First, make sure you followed the instructions at the start of the first Unity Lab to ***set the external script editor*** *in Unity, and make sure the* ***Editor Attaching checkbox is checked****. If you still don't see "Attach to Unity and Play" when Unity opens the script in Visual Studio, go to the Individual Tools section in the Visual Studio installer and make sure Visual Studio Tools for Unity is installed (Windows only). And a few users have found that unchecking Editor Attaching, restarting Unity, then re-checking it and restarting again can help fix it.*

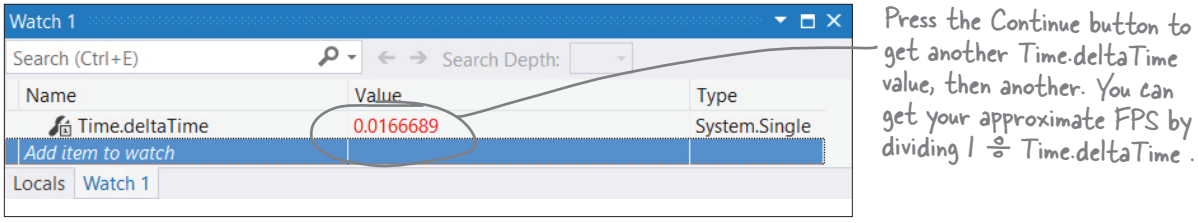
Use the debugger to understand time.deltaTime

You're going to be using Time.deltaTime in many of the Unity Labs projects. Let's take advantage of your breakpoint and use the debugger to really understand what's going on with time.deltaTime.

While your game is paused on the breakpoint in Visual Studio, **hover over Time.deltaTime** to see the fraction of a second that elapsed since the previous frame (you'll need to put your mouse cursor over `deltaTime`). Then **add a watch for time.deltaTime** by clicking on the value and choosing Add Watch from the pop-up menu.



Click Continue (use the Continue button or choose Continue (F5) from the Debug menu) in Visual Studio to resume your game. The ball will start rotating again, and after another 500 frames the breakpoint will trigger again. You can keep running the game for 500 frames at a time. Keep your eye on the Watch window each time it breaks.



Press Stop debugging (F5) to stop your program. Then start debugging again—since your game is still running, the breakpoint will continue to work. Once you're done debugging, **toggle your breakpoint again to delete it** so the IDE will still keep track of it but not break when it's hit. **Stop debugging** one more time to detach from Unity.

Go back to Unity and **stop your game**—and save it, because the Play button doesn't automatically save the game.

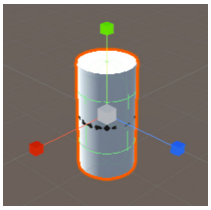
The Play button in Unity starts and stops your game. Visual Studio will stay attached to Unity even when the game is stopped.

Debug your game again and hover over **`Vector3.up`** to inspect its value—you'll have to put your mouse cursor over **up**. It has a value of `(0.0, 1.0, 0.0)`. What do you think that means?

Add a cylinder to show where the Y axis is

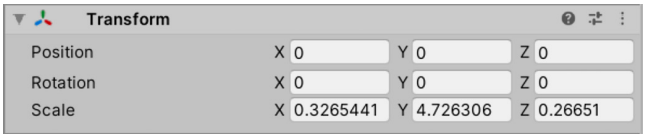
Your sphere is rotating around the Y axis at the very center of the scene. Let's add a very tall and very skinny cylinder to make it visible. **Choose 3D Object >> Cylinder** from the GameObject menu to create a new cylinder. Make sure it's selected in the Hierarchy window, then check the Inspector window and make sure that Unity created it at position (0, 0, 0) – if not, use the context menu ( ) to reset it.

Let's make the cylinder very skinny. Choose the Scale Tool from the toolbar: either click on it ( ) or press the R key. You should see the Scale Gizmo appear on your cylinder:



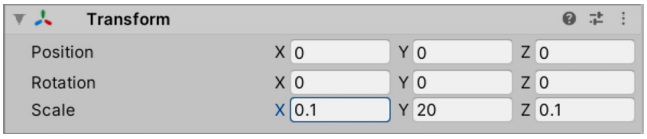
The Scale Gizmo looks a lot like the Move Gizmo, except that it has cubes instead of cones at the end of each axis. Your new cylinder is sitting on top of the sphere—you might see just a little of the sphere showing through the middle of the cylinder. When you make the cylinder narrower by changing its scale along the X and Z axis, the sphere will get uncovered.

Click and drag the green cube up to elongate your cylinder along the Y axis. Then click on the red cube and drag it towards the cylinder to make it very narrow along the X axis, and do the same with the blue cube to make it very narrow along the Z axis. Watch the Transform panel in the Inspector as you change the cylinder's scale—the Y scale will get larger, and the X and Z will get much smaller.

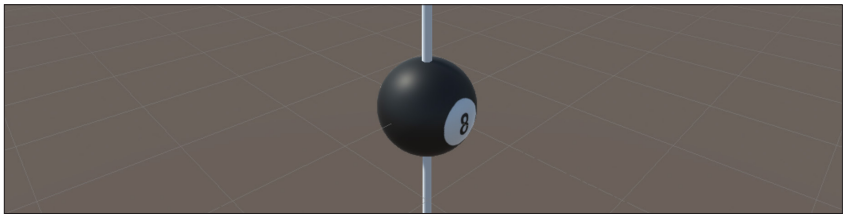


Click on the X label in the scale row in the Transform window and drag up and down. Make sure you click the actual X label to the left of the input box with the number. When you click the label it turns blue, and a blue box appears around the X value. As you drag your mouse up and down, the number in the box goes up and down, and the Scene view updates the scale in as you change it. Look closely as you drag—the scale can be positive and negative.

Now **select the number inside the X box and type .1** – the cylinder gets very skinny. Press tab and type 20, then press tab again and type .1, and press enter.



Now your sphere has a very long cylinder going through it that shows the Y axis where Y = 0.



Add fields to your class for the rotation angle and speed

In Chapter 1 you learned how C# classes can have **fields** that store values methods can use. Let's modify your code to use fields. Add these four lines just under the class declaration, **immediately after the first curly brace {**:

```
public class BallBehaviour : MonoBehaviour
{
    public float XRotation = 0;
    public float YRotation = 1;
    public float ZRotation = 0;
    public float DegreesPerSecond = 180;
```

These are like the fields that you added to the animal match game in Chapter 1. They're variables that keep track of their values—each time Update is called it reuses the same field over and over again.

The XRotation, YRotation, and ZRotation fields contain a value between 0 and 1, which you'll combine to create a **vector** that determines the direction that the ball will rotate:

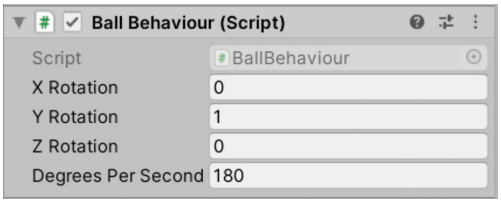
```
new Vector3(XRotation, YRotation, ZRotation)
```

You'll learn a lot more about fields and the new keyword in the next chapter.

The DegreesPerSecond field contains the number of degrees to rotate per second, which you'll multiply by Time.deltaTime just like before. **Modify your Update method to use the fields**. This new code creates a Vector3 variable called axis and passes it to the transform.Rotate method:

```
    void Update()
    {
        Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
        transform.Rotate(axis, DegreesPerSecond * Time.deltaTime);
    }
```

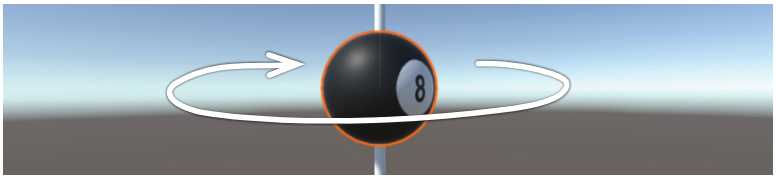
Select the Sphere in the Hierarchy window. Your fields now show up in the Script component. When the Script component renders fields, it adds spaces between the capital letters to make them easier to read.



When you add fields to the class in your Unity script, the Script component displays input boxes that let you modify those fields. If you modify them while the game is not running, the updated values will get saved with your scene. You can also modify while the game is running, but they'll revert when you stop the game.

Run your game again. ***While it's running***, select Sphere in the hierarchy and change the degrees per second to 360 or 90—the ball starts to spin at twice or half the speed. Stop your game—degrees per second field resets back to 180.

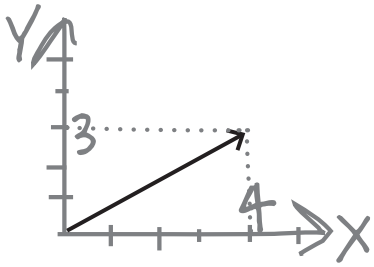
While the game is stopped, use the Unity editor to change the X Rotation to field to 1 and the Y Rotation field to 0. Start your game—the ball will rotate away from you. Click the X Rotation label and drag it up and down to change the value while the game is running,. As soon as the number turns negative, the ball starts rotating towards you. Make it positive again and it starts rotating away from you.



When you use the Unity editor to set the YRotation field to 1 and then start your game, the ball rotates clockwise around the Y axis.

Use Debug.DrawRay to explore how 3D vectors work

A **vector** is a value with a **length** (or magnitude) and a **direction**. If you ever learned about vectors in a math class, you probably saw lots of diagrams like this one of a 2D vector:



Here's a diagram of a 2-dimensional vector. You can represent it with two numbers: its value on the X axis (4) and its value on the Y axis (3), which you'd typically write as (4, 3).

That's not hard to understand... on an intellectual level. But even those of us took a math class that covered vectors don't always have an ***intuitive*** grasp of how vectors work, especially in 3D. And here's another area where we can use C# and Unity as a tool for learning and exploration.

Use Unity to visualize vectors in 3D

You're going to add code to your game to help you really "get" how 3D vectors work. Start by having a closer look at the first line of your Update method:

```
Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
```

So what does this line do, exactly?

- **It has a type: `Vector3`.** Every variable declaration starts with a type. Instead of using string, int, or bool, you're declaring it with the type Vector3. This is a type that Unity uses for 3D vectors.
- **It has a variable name: `axis`.** It has a type: Vector3. And it has a type, just like the similar variable declarations that you saw in Chapter 1—except instead of `string`, `int`, or `bool` you're declaring it with the type **`Vector3`**, Unity's 3D vector type.
- **It uses the new keyword to create a `Vector3`.** It uses the XRotation, YRotation, and ZRotation fields to create a vector with those values.

So what does that 3D vector look like? There's no need to guess—we can use one of Unity's useful debugging tools to draw that vector for us. **Add this line of code to the end of your Update method:**

```
void Update()
{
    Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
    transform.Rotate(axis, DegreesPerSecond * Time.deltaTime);
    Debug.DrawRay(Vector3.zero, axis, Color.yellow);
}
```

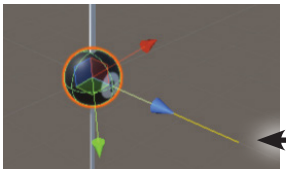
The Debug.DrawRay method is a special method that Unity gives you to help you debug your games. It draws a **ray**—which is a vector that goes from one point to another—and takes parameters for its start point, end point, and color. But there's one catch: ***the ray only appears in the Scene view***. The methods in Unity's Debug class are designed so that they don't interfere with your game. They typically only affect how your game interacts with the Unity editor.

Run the game to see the ray in the Scene view

Now run your game again. You won't see anything different in the Game view because the Debug.DrawRay is a tool for debugging that doesn't affect gameplay at all. So the Scene tab to **switch to the scene view**. You may also need to **reset the Wide layout** by choosing Wide from the layout dropdown.

Now you're back in the familiar Scene view. Then do these things to get a real sense of how 3D vectors work:

- Set the X rotation to 0, Y rotation to 0, and Z rotation to 3. The ball was mid-rotation, so use the context menu ( ) in the Transform component to reset its position. You should now see a yellow ray coming directly out of the Z axis and the ball rotating around it.



The vector (0, 0, 3) extends 3 units along the Z axis. Look closely at the grid in the Unity editor—the vector is exactly 3 units long. Try clicking and dragging the Z Rotation label in the Script component in the Inspector. The ray will get larger or smaller as you drag. When the Z value in the vector is negative, the ball rotates the other direction.

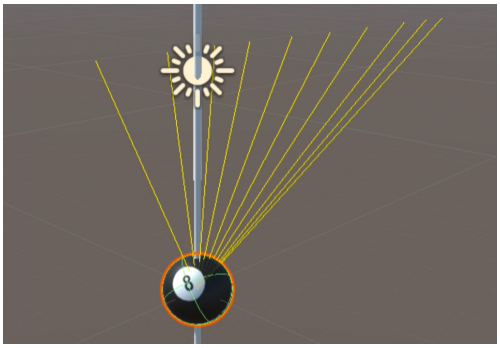
- Set the Z Rotation back to 3. Experiment with dragging the X Rotation and Y Rotation values to see what they do to the ray. Make sure to reset the Transform component each time you change them.
- Use the Hand Tool and the Scene Gizmo to get a better view. Click the X cone on the Scene Gizmo to set it to the view from the right. Keep clicking the cones on the Scene Gizmo until you see the view from the front. It's easy to get lost—you can **reset the Wide layout to get back to a familiar view**.

Add a duration to the Ray so it leaves a trail

You can add a fourth argument to your Debug.DrawRay method call that specifies the number of seconds the ray should stay on the screen. Add .5f to make each ray stay on screen for half a second:

```
Debug.DrawRay(Vector3.zero, axis, Color.yellow, .5f);
```

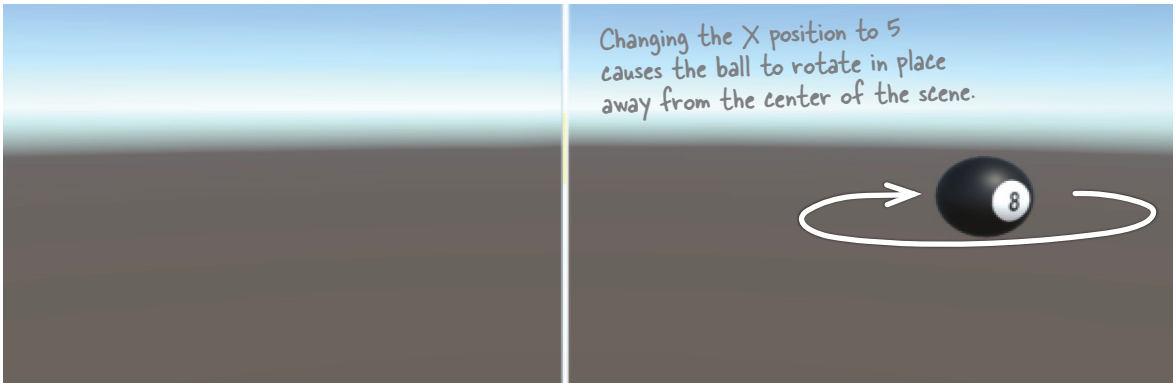
Now run the game again and switch to the Scene view. Now when you drag the numbers up and down, you'll see a trail of rays left behind. This looks really interesting, but more importantly, it's a great tool to visualize 3D vectors.



Making your ray leave a trail is a good way to help you develop an intuitive sense of how 3D vectors work.

Rotate your ball around a point in the scene

Your code calls transform.Rotate method to rotate your ball around its center, which changes its X, Y, and Z rotation values. **Select Sphere in the Hierarchy window and change its X position to 5** in the Transform component. Then **use the context menu to reset the Script component** to reset its fields so it rotates around the Y axis again. When you run the game, the ball will be at position (5, 0, 0) and rotating around its own Y axis.



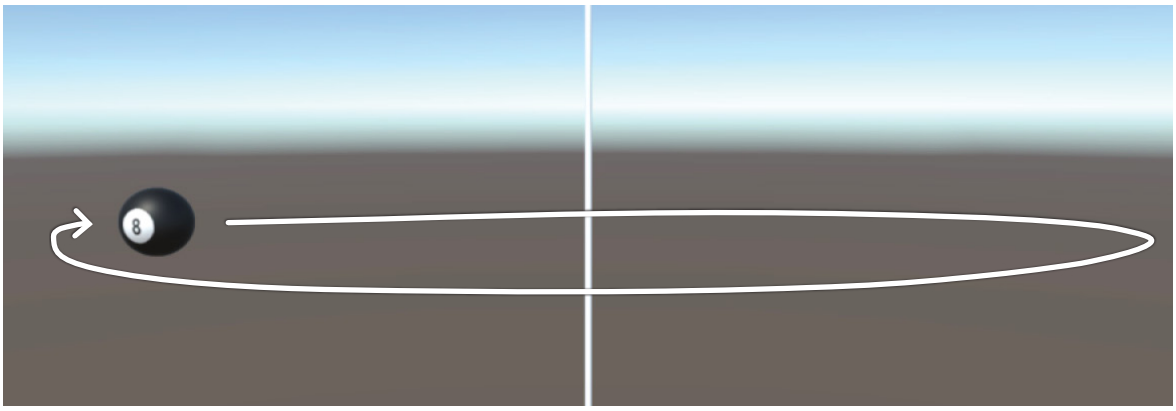
Let's modify the Update method to use a different kind of rotation. Now we'll make the ball rotate around the center point of the scene, coordinate [0, 0, 0] using the **transform.RotateAround method**, which rotates a GameObject around a point in the scene. (This is *different* from the transform.Rotate you used earlier, which rotates it a GameObject around its center.) Its first parameter is the point to rotate around. We'll use **`Vector3.zero`** for that parameter, which is a shortcut for writing `new Vector3(0, 0, 0)`.

Here's the new Update method:

This new Update method rotates the ball around the point (0, 0, 0) in the scene.

```
void Update()
{
    Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
    transform.RotateAround(Vector3.zero, axis, DegreesPerSecond * Time.deltaTime);
    Debug.DrawRay(Vector3.zero, axis, Color.yellow, .5f);
}
```

Now run your code. This time it rotates the ball in a big circle around the center point:

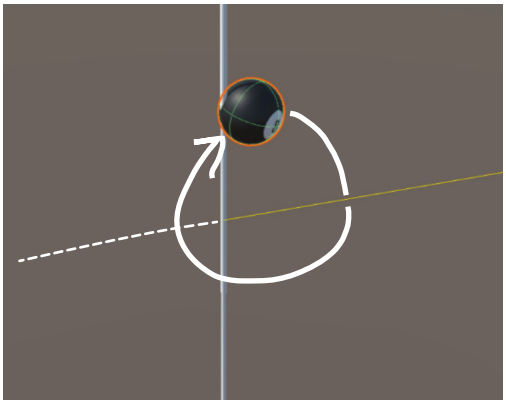


Use Unity to take a closer look at rotation and vectors

You're going to be working with 3D objects and scenes in the rest of the Unity Labs throughout the book. But even those of us who spend a lot of time playing 3D video games don't have a perfect feel for how vectors, 3D objects, and how to move and rotate in a 3D space. Luckily, Unity is a great tool to **explore how 3D objects work**. Let's start experimenting right now.

While your code is running, try changing parameters to experiment with the rotation:

- **Switch back to the Scene view** so you can see the yellow ray that Debug.DrawRay renders in your BallBehaviour.Update method.
- Use the Hierarchy window to **select the Sphere**. You should see its components in the Inspector window.
- Change the **X Rotation, Y Rotation, and Z Rotation values** in the Script component to 10 so you see the vector rendered as a long ray. Use the Hand Tool (Q) to rotate the Scene View until you can clearly see the ray.
- Use the Transform component's context menu (⋮) to **reset the Transform component**. Since the center of the sphere is now at the zero point in the scene (0, 0, 0), it will rotate around its own center.
- Then **change the X position in** the Transform component to 2. The ball should now be rotating around the vector. You'll see the ball cast a shadow on the Y axis cylinder as it flies by.



While the game is running, set the X, Y, and Z rotation fields in the Ball Behaviour (Script) component to 10, reset the sphere's Transform component, and change its X position to 2 – as soon as you do, it starts rotating around the ray.

Try **repeating the last three steps** for different values of X, Y, and Z rotation, resetting the Transform component each time so you start from a fixed point. Then try clicking the rotation field labels and dragging them up and down—see if you can get a feel for how the rotation works.

Unity is a great tool to explore how 3D objects work. by modifying properties on your GameObjects in real time.

Get creative!

This is your chance to **experiment on your own with C# and Unity**. You've seen the basics of how you combine C# and Unity GameObjects. Take some time and play around with the different Unity tools and methods that you've learned about in the first two Unity labs.

- Add Cubes, Cylinders, or Capsules to your scene. Attach new scripts to them—make sure you give each script a unique name!—and make them rotate in different ways.
- Try putting your rotating GameObjects in different positions around the scene. See if you can make interesting visual patterns out of multiple rotating GameObjects.
- Try adding a light to the scene. What happens when you use Transform.rotateAround to rotate the new light around various axes?
- Here's quick coding challenge. Try modifying a script to use += to add a value to one of the fields in your BallBehaviour script. Make sure you multiply that value by Time.deltaTime. Try adding an if statement that resets the field to 0 if it gets too large.

Before you run the code, try to figure out what it will do. Does it act the way you expected it to act? Trying to predict how the code you added will act is a great technique for getting better at C#.

Take the time to experiment with the tools and techniques you just learned. This is a great way to take advantage of Unity and Visual Studio as tools for exploration and learning.

BULLET POINTS

- The **Scene Gizmo** always displays the camera's orientation.
- You can **attach a C# script** to any GameObject. The script's Update method will be called once per frame.
- The **transform.Rotate method** causes a GameObject to rotate a number of degrees around an axis.
- Inside your Update method, multiplying any value by **Time.deltaTime** turns it into that value per second.
- You can **attach** the Visual Studio debugger to Unity to debug your game while it's running. It will stay attached to Unity even when your game is not running.
- Adding a **Hit Count condition** to a breakpoint to makes it break after the statement has executed a certain number of times.
- A **field** is a variable that lives inside of a class outside of its methods, and it retains its value between method calls.
- When you add fields to the class in your Unity script, the Script component displays **input boxes that let you modify those fields**. It inserts spaces between capital letters in the field names to make them easier to read.
- You can create 3D vectors using **`new Vector3`**. (You'll learn more about the `new` keyword in Chapter 3.)
- The **Debug.DrawRay method** draws a vector in the Scene view (but not the Game view). You can use vectors as a debugging tool, but also as a learning tool.
- The **transform.RotateAround method** rotates a GameObject around a point in the scene.