O'REILLY®

Fourth Edition

Head First

C#

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

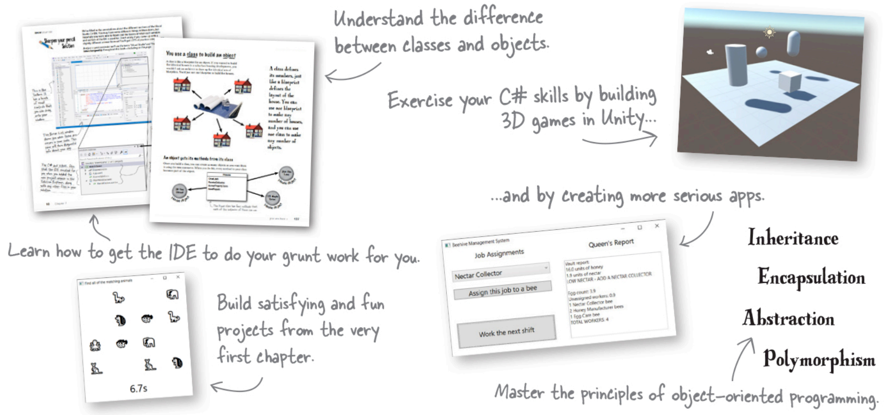
Andrew Stellman
& Jennifer Greene

A Brain-Friendly Guide

Head First C#

What will you learn from this book?

Dive into C# and create apps, user interfaces, games, and more using this fun and highly visual introduction to C#, .NET Core, and Visual Studio. With this completely updated guide, which covers C# 8.0 and Visual Studio 2019, beginning programmers like you will build a fully functional game in the opening chapter. Then you'll learn how to use classes and object-oriented programming, create 3D games in Unity, and query data with LINQ. And you'll do it all by solving puzzles, doing hands-on exercises, and building real-world applications. By the time you're done, you'll be a solid C# programmer—and you'll have a great time along the way!


Understand the difference between classes and objects.
Exercise your C# skills by building 3D games in Unity...
...and by creating more serious apps.
Learn how to get the IDE to do your grunt work for you.
Build satisfying and fun projects from the very first chapter.
Inheritance
Encapsulation
Abstraction
Polymorphism
Master the principles of object-oriented programming.

What's so special about this book?

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

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

—Ryan White
Game Developer

"Andrew and Jennifer have written a concise, authoritative, and most of all, fun introduction to C# development."

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

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

—Andy Parker
Fledgling C# programmer

.NET

US $64.99 **CAN $85.99**

ISBN: 978-1-491-97670-8

56499

9 781491 976708

O'REILLY®

Praise for *Head First C#*

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

—Ryan White, Game Developer

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

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

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

—Jeff Counts, Senior C# Developer

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

—Lindsey Bieda, Lead Software Engineer

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

—Shalewa Odusanya, Principal

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

—Johnny Halife, Partner

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

—Rebeca Dunn-Krahn, founding Partner, Sempahore Solutions

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

—Andy Parker, fledgling C# Programmer

More Praise for *Head First C#*

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

—Chris Burrows, Software Engineer

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

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

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

—David Sterling, Principal Software Developer

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

—Joseph Albahari, inventor of LINQPad, and coauthor of *C# 8.0 in a Nutshell* and *C# 8.0 Pocket Reference*

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

—Giuseppe Turitto, Director of Engineering

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

—Bill Mietelski, Advanced Systems Analyst

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

—Krishna Pala, MCP

Praise for other *Head First* books

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

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

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

— Aaron LaBerge, SVP Technology & Product Development, ESPN

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

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

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

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

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

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

— Eric Wuehler

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

— Satish Kumar

Head First C#

Fourth Edition

WOULDN'T IT BE DREAMY IF THERE WAS A C# BOOK THAT WAS MORE FUN THAN MEMORIZING A DICTIONARY? IT'S PROBABLY NOTHING BUT A FANTASY...

Andrew Stellman
Jennifer Greene

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Head First C#

Fourth Edition

by Andrew Stellman and Jennifer Greene

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

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

Series Creators:	Kathy Sierra, Bert Bates
Cover Designer:	Ellie Volckhausen
Brain Image on Spine:	Eric Freeman
Editors:	Nicole Taché, Amanda Quinn
Proofreader:	Rachel Head
Indexer:	Potomac Indexing, LLC
Illustrator:	Jose Marzan
Page Viewers:	Greta the miniature bull terrier and Samosa the Pomeranian

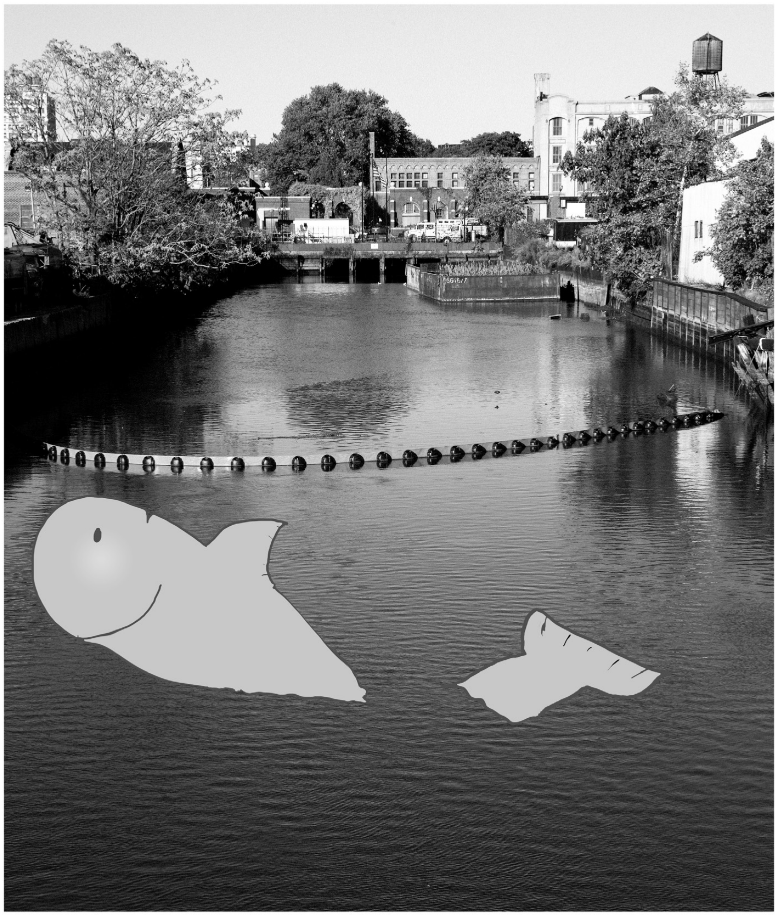
Printing History:

November 2007: First Edition
May 2010: Second Edition
August 2013: Third Edition
December 2020: Fourth Edition

ISBN: 978-1-491-97670-8

[LSI] [2020-12-18]

This book is dedicated to the loving memory of Sludgie the Whale, who swam to Brooklyn on April 17, 2007.



You were only in our canal for a day, but you'll be in our hearts forever.

THANKS FOR READING OUR BOOK! WE REALLY LOVE WRITING ABOUT THIS STUFF, AND WE HOPE YOU GET A LOT OUT OF IT...

...BECAUSE WE KNOW YOU'RE GOING TO HAVE A GREAT TIME LEARNING C#.

This photo (and the photo of the Gowanus Canal) by Nisha Sondhe

Andrew Stellman, despite being raised a New Yorker, has lived in Minneapolis, Geneva, and Pittsburgh… *twice*, first when he graduated from Carnegie Mellon's School of Computer Science, and then again when he and Jenny were starting their consulting business and writing their first book for O'Reilly.

Andrew's first job after college was building software at a record company, EMI-Capitol Records—which actually made sense, as he went to LaGuardia High School of Music & Art and the Performing Arts to study cello and jazz bass guitar. He and Jenny first worked together at a company on Wall Street that built financial software, where he was managing a team of programmers. Over the years he's been a vice president at a major investment bank, architected large-scale real-time backend systems, managed large international software teams, and consulted for companies, schools, and organizations, including Microsoft, the National Bureau of Economic Research, and MIT. He's had the privilege of working with some pretty amazing programmers during that time, and likes to think that he's learned a few things from them.

When he's not writing books, Andrew keeps himself busy writing useless (but fun) software, playing (and making) both music and video games, practicing krav maga, tai chi, and aikido, and owning a crazy Pomeranian.

Jennifer Greene studied philosophy in college but, like everyone else in the field, couldn't find a job doing it. Luckily, she's a great software engineer, so she started out working at an online service, and that's the first time she really got a good sense of what good software development looked like.

She moved to New York in 1998 to work on software quality at a financial software company. She's managed teams of developers, testers, and PMs on software projects in media and finance since then.

Jenny has traveled all over the world to work with different software teams and build all kinds of cool projects.

She loves traveling, watching Bollywood movies, reading the occasional comic book, playing video games, and hanging out with her huge Siberian cat, Sascha, and her miniature bull terrier, Greta.

Jenny and Andrew have been building software and writing about software engineering together since they first met in 1998. Their first book, *Applied Software Project Management*, was published by O'Reilly in 2005. Other Stellman and Greene books for O'Reilly include *Beautiful Teams* (2009), *Learning Agile* (2014), and their first book in the Head First series, *Head First PMP* (2007), now in its fourth edition.

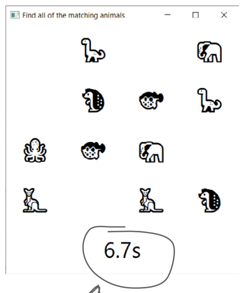
They founded Stellman & Greene Consulting in 2003 to build a really neat software project for scientists studying herbicide exposure in Vietnam vets. In addition to building software and writing books, they've consulted for companies and spoken at conferences and meetings of software engineers, architects, and project managers.

Learn more about them on their website, **Building Better Software**: https://www.stellman-greene.com.

Follow **@AndrewStellman** and **@JennyGreene** on Twitter

☮❤👾 Jenny and Andrew

Table of Contents (the summary)



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.

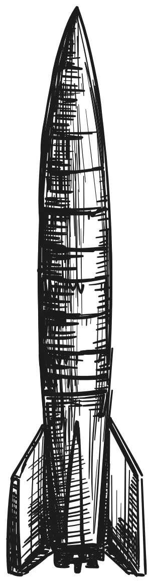
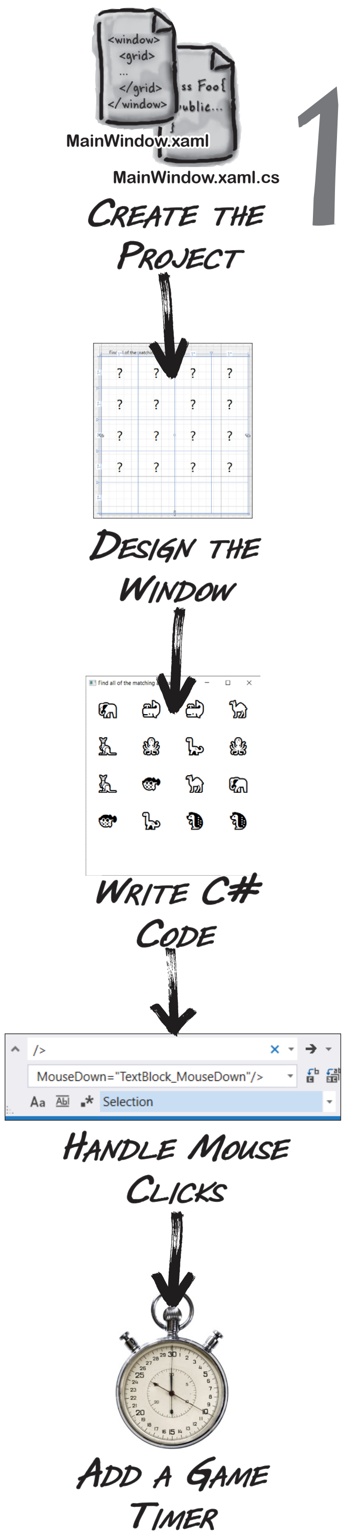


Table of Contents (the real thing)

Intro

Your brain on C#. You're sitting around trying to *learn* something, but your *brain* keeps telling you all that learning *isn't important*. Your brain's saying, "Better leave room for more important things, like which wild animals to avoid and whether nude archery is a bad idea." So how *do* you trick your brain into thinking that your life really depends on learning C#?



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

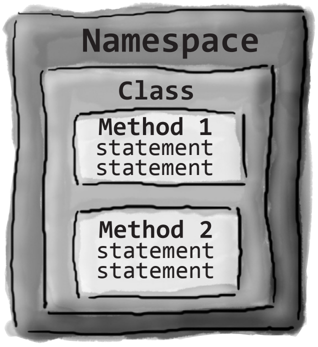
2

dive into C#

Statements, classes, and code

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

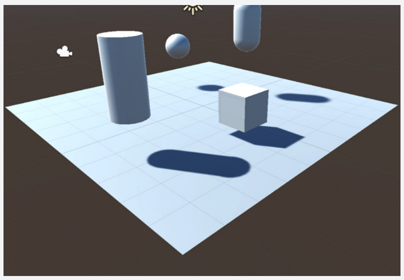
You can get a lot of work done using the IDE, but there's only so far it can take you. Visual Studio is one of the most advanced software development tools ever made, but a **powerful IDE** is only the beginning. It's time to **dig in to C# code**: how it's structured, how it works, and how you can take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

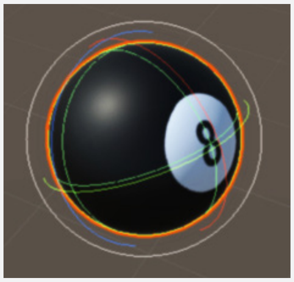


Unity Lab 1

Explore C# with Unity

Welcome to your first **Head First C# Unity Lab**. Writing code is a skill, and like any other skill, getting better at it takes **practice and experimentation**. Unity will be a really valuable tool for that. In this lab, you can begin practicing what you've learned about C# in Chapters 1 and 2.



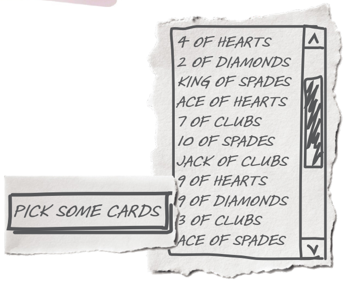


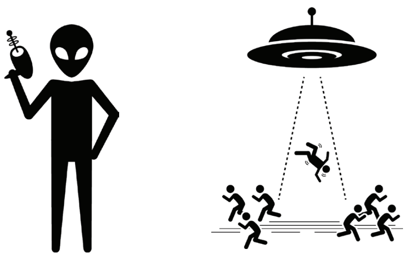
3 objects...get oriented!

Making code make sense

Every program you write solves a problem.

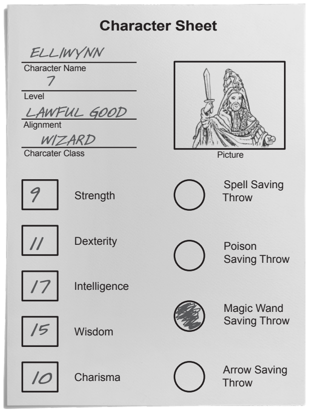
When you're building a program, it's always a good idea to start by thinking about what *problem* your program's supposed to solve. That's why **objects** are really useful. They let you structure your code based on the problem it's solving so that you can spend your time *thinking about the problem* you need to work on rather than getting bogged down in the mechanics of writing code. When you use objects right—and really put some thought into how you design them—you end up with code that's *intuitive* to write, and easy to read and change.



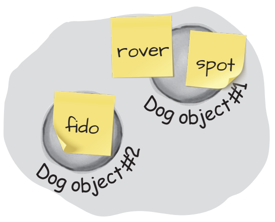


4 types and references: Getting the reference

What would your apps be without data? Think about it for a minute. Without data, your programs are…well, it's actually hard to imagine writing code without data. You need **information** from your users, and you use that to look up or produce new information to give back to them. In fact, almost everything you do in programming involves **working with data** in one way or another. In this chapter, you'll learn the ins and outs of C#'s **data types** and **references**, see how to work with data in your program, and even learn a few more things about **objects** (*guess what…objects are data, too!*).



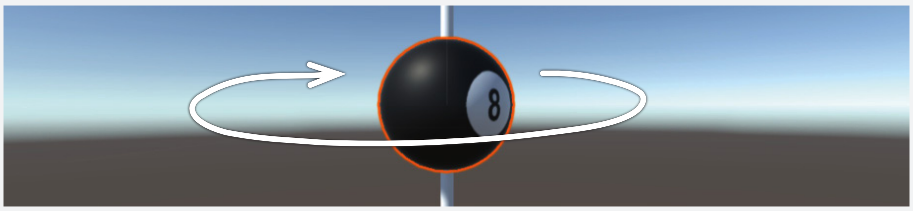
Creating a reference is like writing a name on a sticky note and sticking it to the object. You're using it to label an object so you can refer to it later.



Unity Lab 2

Write C# Code for Unity

Unity isn't *just* a powerful, cross-platform engine and editor for building 2D and 3D games and simulations. It's also a **great way to get practice writing C# code**. In this lab, you'll get more practice writing C# code for a project in Unity.

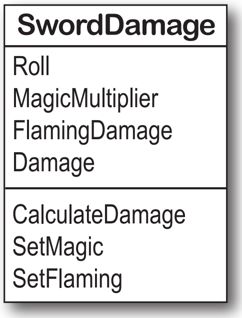


5 encapsulation

Keep your privates...private

Ever wished for a little more privacy?

Sometimes your objects feel the same way. Just like you don't want anybody you don't trust reading your journal or paging through your bank statements, good objects don't let ***other*** objects go poking around their fields. In this chapter, you're going to learn about the power of **encapsulation**, a way of programming that helps you make code that's flexible, easy to use, and difficult to misuse. You'll **make your objects' data private**, and add **properties** to protect how that data is accessed.

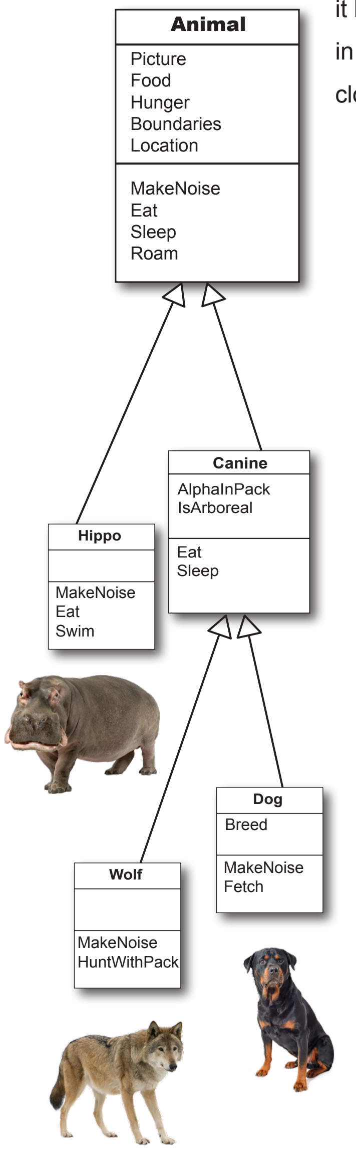


```
RealName: "Herb Jones"

Alias: "Dash Martin"

Password: "the crow flies at midnight"
```

6 inheritance
Your object's family tree

Sometimes you *DO* want to be just like your parents.

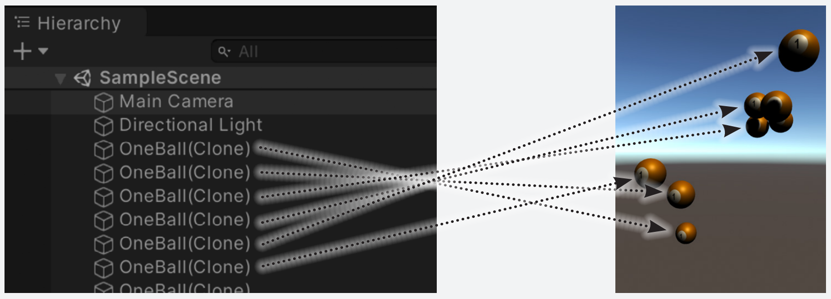
Ever run across a class that ***almost*** does exactly what you want ***your*** class to do? Found yourself thinking that if you could just ***change a few things***, that class would be perfect? With **inheritance**, you can **extend** an existing class so your new class gets all of its behavior—with the **flexibility** to make changes to that behavior so you can tailor it however you want. Inheritance is one of the most powerful concepts and techniques in the C# language: with it you'll **avoid duplicate code**, **model the real world** more closely, and end up with apps that are **easier to maintain** and **less prone to bugs**.



Unity Lab 3

GameObject Instances

C# is an object-oriented language, and since these Head First C# Unity Labs are all **about getting practice writing C# code**, it makes sense that these labs will focus on creating objects.

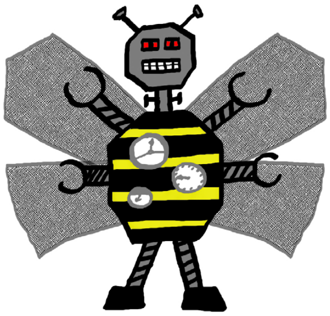


7 interfaces, casting, and "is"

Making classes keep their promises

Actions speak louder than words.

Sometimes you need to group your objects together based on the **things they can do** rather than the classes they inherit from. That's where **interfaces** come in—they let you work with any class that can do the job. But with **great power comes great responsibility**, and any class that implements an interface must promise to **fulfill all of its obligations**…or the compiler will break its kneecaps, see?



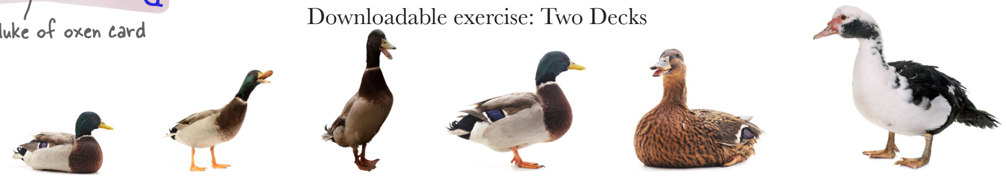
8 enums and collections

Organizing your data

When it rains, it pours.

In the real world, you don't receive your data in tiny little bits and pieces. No, your data's going to come at you in **loads, piles, and bunches**. You'll need some pretty powerful tools to organize all of it, and that's where **enums** and collections come in. Enums are types that let you define valid values to categorize your data. Collections are special objects that store many values, letting you **store, sort, and manage** all the data that your programs need to pore through. That way, you can spend your time thinking about writing programs to work with your data, and let the collections worry about keeping track of it for you.

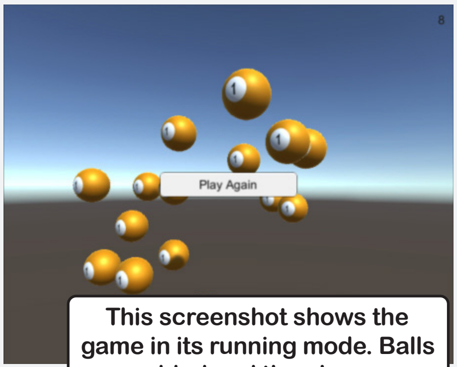
The rarely-played duke of oxen card



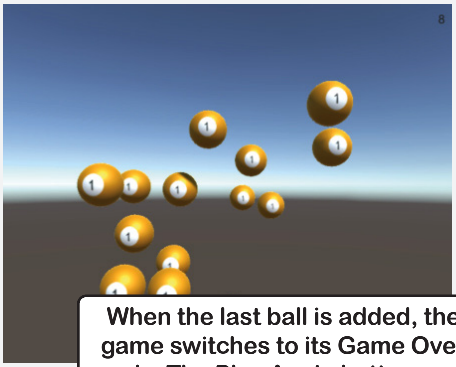
Unity Lab 4

User Interfaces

In the last Unity Lab you started to build a game, using a prefab to create GameObject instances that appear at random points in 3D space and fly in circles. This Unity Lab picks up where the last one left off, allowing you to apply what you've learned about interfaces in C# and more.



This screenshot shows the game in its running mode. Balls are added and the player can click on them to score.



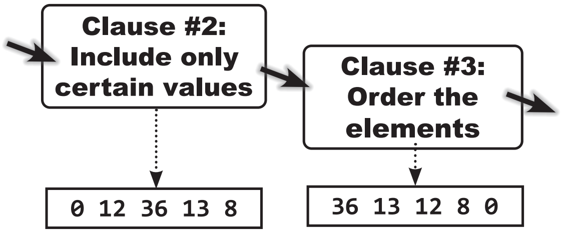
When the last ball is added, the game switches to its Game Over mode. The Play Again button pops up and no more balls get added.

9 LINQ and lambdas

Get control of your data

You're ready for a whole new world of app development.

Using WinForms to build Windows Desktop apps is a great way to learn important C# concepts, but there's *so much more* you can do with your programs. In this chapter, you'll use **XAML** to design your Windows Store apps, you'll learn how to **build pages to fit any device**, **integrate** your data into your pages with **data binding**, and use Visual Studio to cut through the mystery of XAML pages by exploring the objects created by your XAML code.

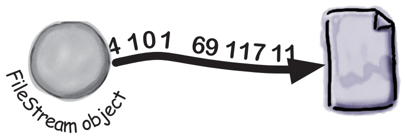

Clause #2: Include only certain values
0 12 36 13 8
Clause #3: Order the elements
36 13 12 8 0

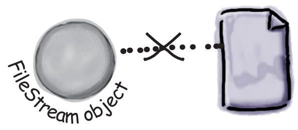
10 reading and writing files

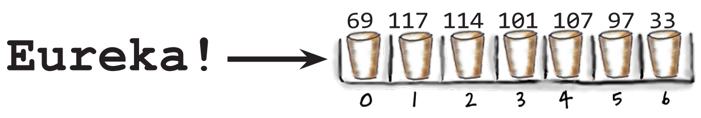
Save the last byte for me!

Sometimes it pays to be a little persistent.

So far, all of your programs have been pretty short-lived. They fire up, run for a while, and shut down. But that's not always enough, especially when you're dealing with important information. You need to be able to **save your work**. In this chapter, we'll look at how to **write data to a file**, and then how to **read that information back in** from a file. You'll learn about the .NET **stream classes**, and also take a look at the mysteries of **hexadecimal** and **binary**.



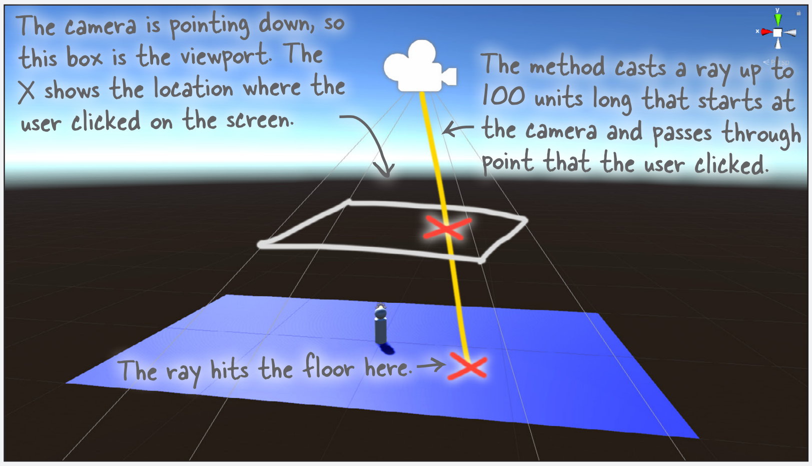




Unity Lab 5

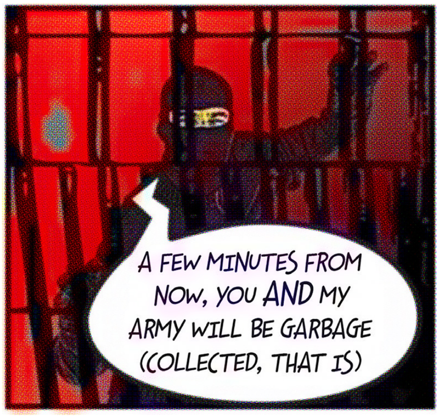
Raycasting

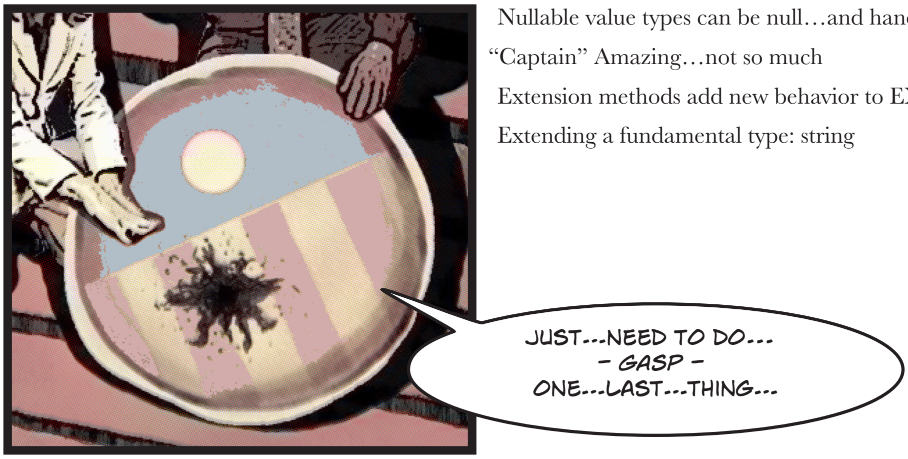
When you set up a scene in Unity, you're creating a virtual 3D world for the characters in your game to move around in. But in most games, things aren't directly controlled by the player. So how do these objects find their way around a scene? In this lab, we'll look at how C# can help.


The camera is pointing down, so this box is the viewport. The X shows the location where the user clicked on the screen.
The method casts a ray up to 100 units long that starts at the camera and passes through point that the user clicked.
The ray hits the floor here.

CAPTAIN AMAZING THE DEATH OF THE OBJECT

Head First C#
Four bucks
Chapter 11


A FEW MINUTES FROM NOW, YOU AND MY ARMY WILL BE GARBAGE (COLLECTED, THAT IS)

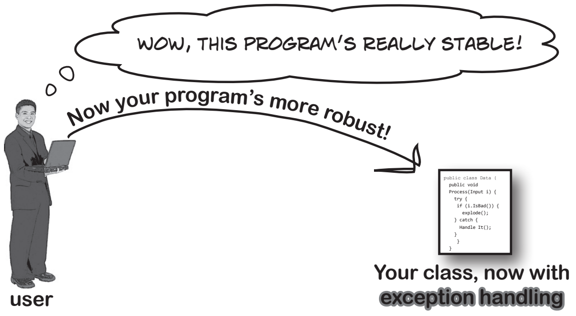


12 exception handling

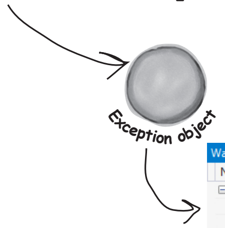
Putting out fires gets old

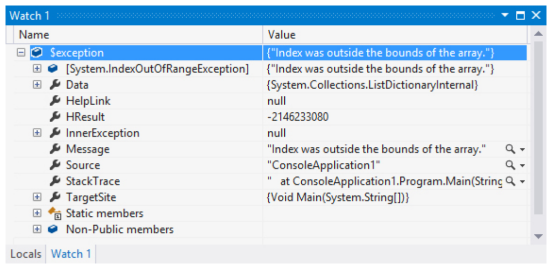
Programmers aren't meant to be firefighters.

You've worked your tail off, waded through technical manuals and a few engaging *Head First* books, and you've reached the pinnacle of your profession. But you're still getting panicked phone calls in the middle of the night from work because **your program crashes**, or **doesn't behave like it's supposed to**. Nothing pulls you out of the programming groove like having to fix a strange bug…but with **exception handling**, you can write code to **deal with problems** that come up. Better yet, you can even plan for those problems, and **keep things running** when they happen.


WOW, THIS PROGRAM'S REALLY STABLE!
Now your program's more robust!
user
Your class, now with exception handling

```
int[] anArray = {3, 4, 1, 11};
int aValue = anArray[15];
```





Watch 1

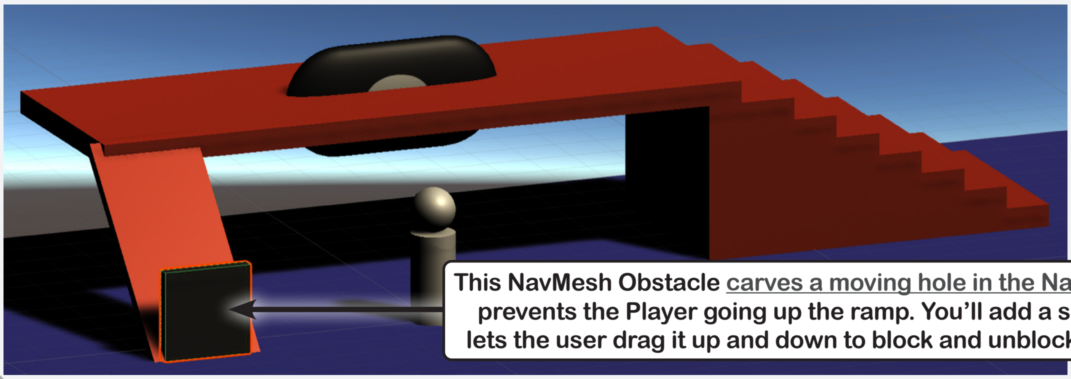
Name	Value
$exception	{"Index was outside the bounds of the array."}
[System.IndexOutOfRangeException]	{"Index was outside the bounds of the array."}
Data	{System.Collections.ListDictionaryInternal}
HelpLink	null
HResult	-2146233080
InnerException	null
Message	"Index was outside the bounds of the array."
Source	"ConsoleApplication1"
StackTrace	" at ConsoleApplication1.Program.Main(Strinç
TargetSite	{Void Main(System.String[])}
Static members	
Non-Public members	

Locals | Watch 1

Unity Lab 6

Scene Navigation

In the last Unity Lab, you created a scene with a floor (a plane) and a player (a sphere nested under a cylinder), and you used a NavMesh, a NavMesh Agent, and raycasting to get your player to follow your mouse clicks around the scene. In this lab, you'll add to the scene with the help of C#.


This NavMesh Obstacle carves a moving hole in the NavMesh that prevents the Player going up the ramp. You'll add a script that lets the user drag it up and down to block and unblock the ramp.

how to use this book

Intro

In this section, we answer the burning question:
"So why DID they put that in a C# programming book?"

Who is this book for?

If you can answer "yes" to all of these:

1. Do you want to **learn C#** (and pick up some knowledge of game development and Unity along the way)?
2. Do you like to tinker? Do you learn by doing, rather than just reading?
3. Do you prefer **interesting and stimulating conversation** to **dry**, **dull**, **academic lectures**?

this book is for you.

Who should probably avoid this book?

If you can answer "yes" to any of these:

1. Are you more interested in theory than practice?
2. Does the idea of doing projects and writing code make you bored and a little twitchy?
3. Are you **afraid to try something different**? Do you think a book about serious topic like development needs to be serious all the time?

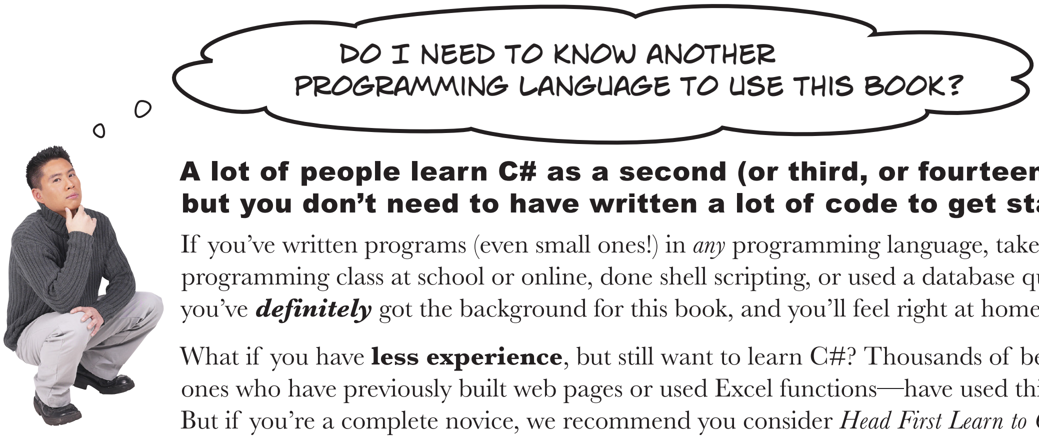
you might consider trying another book first.

Code Kata learning path

Are you an **advanced developer** with experience in multiple languages who wants to ramp up on C# and Unity *fast*?

Are you an **impatient learner** who feels comfortable jumping right into code?

If you answered *YES!* to both of those questions, then we included a **code kata** learning path just for you. Look for the Code Kata appendix at the end of the book to learn more.


DO I NEED TO KNOW ANOTHER PROGRAMMING LANGUAGE TO USE THIS BOOK?

A lot of people learn C# as a second (or third, or fourteenth) language, but you don't need to have written a lot of code to get started.

If you've written programs (even small ones!) in *any* programming language, taken an introductory programming class at school or online, done shell scripting, or used a database query language, then you've ***definitely*** got the background for this book, and you'll feel right at home.

What if you have **less experience**, but still want to learn C#? Thousands of beginners—especially ones who have previously built web pages or used Excel functions—have used this book to learn C#. But if you're a complete novice, we recommend you consider *Head First Learn to Code* by Eric Freeman.

If you're still on the fence about whether or not *Head First C#* is right for you, try doing the first four chapters. You can download them for free from https://github.com/head-first-csharp/fourth-edition. If you're still comfortable after that, then you've got the right book! If they leave your head spinning, you should definitely consider reading *Head First Learn to Code* – after that, you'll be 100% ready for this book.

We know what you're thinking.

"How can *this* be a serious C# programming book?"

"What's with all the graphics?"

"Can I actually *learn* it this way?"

And we know what your *brain* is thinking.

Your brain craves novelty. It's always searching, scanning, *waiting* for something unusual. It was built that way, and it helps you stay alive.

So what does your brain do with all the routine, ordinary, normal things you encounter? Everything it *can* to stop them from interfering with the brain's *real* job—recording things that *matter*. It doesn't bother saving the boring things; they never make it past the "this is obviously not important" filter.

How does your brain *know* what's important? Suppose you're out for a day hike and a tiger jumps out in front of you. What happens inside your head and body?

Neurons fire. Emotions crank up. *Chemicals surge*.

And that's how your brain knows…

This must be important! Don't forget it!

But imagine you're at home, or in a library. It's a safe, warm, tiger-free zone. You're studying. Getting ready for an exam. Or trying to learn some tough technical topic your boss thinks will take a week, ten days at the most.

Just one problem. Your brain's trying to do you a big favor. It's trying to make sure that this *obviously* unimportant content doesn't clutter up scarce resources. Resources that are better spent storing the really *big* things. Like tigers. Like the danger of fire. Like how you should never have posted those "party" photos on your Facebook page.

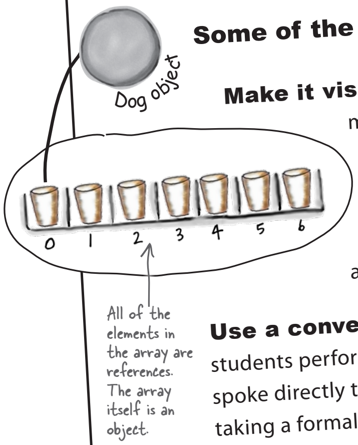
And there's no simple way to tell your brain, "Hey brain, thank you very much, but no matter how dull this book is, and how little I'm registering on the emotional Richter scale right now, I really *do* want you to keep this stuff around."

GREAT. ONLY 700 MORE DULL, DRY, BORING PAGES.
Your brain thinks THIS isn't worth saving.

We think of a "Head First" reader as a learner.

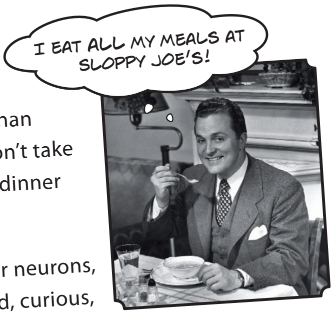
So what does it take to *learn* something? First you have to *get* it, then make sure you don't *forget* it. It's not about pushing facts into your head. Based on the latest research in cognitive science, neurobiology, and educational psychology, *learning* takes a lot more than text on a page. We know what turns your brain on.

Some of the Head First learning principles:


Dog object
0 1 2 3 4 5 6
All of the elements in the array are references. The array itself is an object.

Make it visual. Images are far more memorable than words alone, and make learning much more effective (up to 89% improvement in recall and transfer studies). They also make things more understandable.

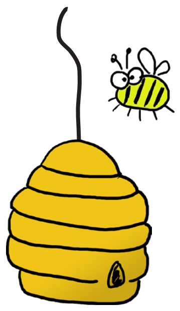
Put the words within or near the graphics they relate to, rather than at the bottom or on another page, and learners will be up to *twice* as likely to be able to solve problems related to the content.



Use a conversational and personalized style. In recent studies, students performed up to 40% better on post-learning tests if the content spoke directly to the reader, using a first-person, conversational style rather than taking a formal tone. Tell stories instead of lecturing. Use casual language. Don't take yourself too seriously. Which would *you* pay more attention to: a stimulating dinner party companion, or a lecture?

Get the learner to think more deeply. Unless you actively flex your neurons, nothing much happens in your head. A reader has to be motivated, engaged, curious, and inspired to solve problems, draw conclusions, and generate new knowledge. And for that, you need challenges, exercises, and thought-provoking questions, and activities that involve both sides of the brain and multiple senses.

Get—and keep—the reader's attention. We've all had the "I really want to learn this but I can't stay awake past page one" experience. Your brain pays attention to things that are out of the ordinary, interesting, strange, eye-catching, unexpected. Learning a new, tough, technical topic doesn't have to be boring. Your brain will learn much more quickly if it's not.



Touch their emotions. We now know that your ability to remember something is largely dependent on its emotional content. You remember what you care about. You remember when you *feel* something. No, we're not talking heart-wrenching stories about a boy and his dog. We're talking emotions like surprise, curiosity, fun, "what the…?", and the amazing "A-ha! I got this!" feeling that comes when you solve a puzzle, learn something everybody else thinks is hard—or maybe just realize you've learned so much *great new stuff* and it feels so good to be able to use it.

Metacognition: thinking about thinking

If you really want to learn, and you want to learn more quickly and more deeply, pay attention to how you pay attention. Think about how you think. Learn how you learn.

Most of us did not take courses on metacognition or learning theory when we were growing up. We were *expected* to learn, but rarely *taught* to learn.

I WONDER HOW I CAN TRICK MY BRAIN INTO REMEMBERING THIS STUFF...

But we assume that if you're holding this book, you really want to learn how to build programs in C#. And you probably don't want to spend a lot of time on it. If you want to use what you read in this book, you need to *remember* what you read. And for that, you've got to *understand* it. To get the most from this book, or *any* book or learning experience, take responsibility for your brain. Your brain on *this* content.

The trick is to get your brain to see the new material you're learning as Really Important. Crucial to your well-being. As important as a tiger. Otherwise, you're in for a constant battle, with your brain doing its best to keep the new content from sticking.

So just how *DO* you get your brain to treat C# like it was a hungry tiger?

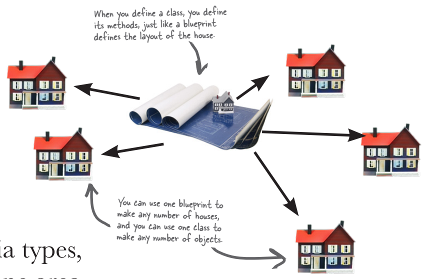
There's the slow, tedious way, or the faster, more effective way. The slow way is about sheer repetition. You obviously know that you *are* able to learn and remember even the dullest of topics if you keep pounding the same thing into your brain. With enough repetition, your brain says, "This doesn't *feel* important, but they keep looking at the same thing *over* and *over* and *over*, so I suppose it must be."

The faster way is to do ***anything that increases brain activity***, especially different *types* of brain activity. The things on the previous page are a big part of the solution, and they're all things that have been proven to help your brain work in your favor. For example, studies show that putting words *within* the pictures they describe (as opposed to somewhere else on the page, like in a caption or in the body text) causes your brain to try to make sense of how the words and pictures relate, and this causes more neurons to fire. More neurons firing = more chances for your brain to *get* that this is something worth paying attention to, and possibly recording.

A conversational style helps because people tend to pay more attention when they perceive that they're in a conversation, since they're expected to follow along and hold up their end. The amazing thing is, your brain doesn't necessarily *care* that the "conversation" is between you and a book! On the other hand, if the writing style is formal and dry, your brain perceives it the same way you experience being lectured to while sitting in a roomful of passive attendees. No need to stay awake.

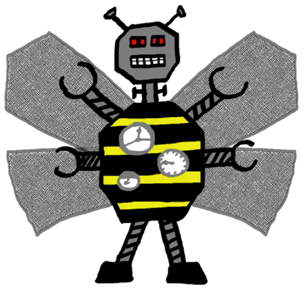
But pictures and conversational style are just the beginning.

Here's what WE did


When you define a class, you define its methods, just like a blueprint defines the layout of the house.
You can use one blueprint to make any number of houses, and you can use one class to make any number of objects.

We used ***pictures***, because your brain is tuned for visuals, not text. As far as your brain's concerned, a picture really *is* worth a thousand words. And when text and pictures work together, we embedded the text *in* the pictures because your brain works more effectively when the text is *within* the thing the text refers to, as opposed to in a caption or buried in the text somewhere.

We used ***redundancy***, saying the same thing in *different* ways and with different media types, and *multiple senses*, to increase the chance that the content gets coded into more than one area of your brain.



We used concepts and pictures in ***unexpected*** ways because your brain is tuned for novelty, and we used pictures and ideas with at least *some* ***emotional*** *content*, because your brain is tuned to pay attention to the biochemistry of emotions. That which causes you to *feel* something is more likely to be remembered, even if that feeling is nothing more than a little ***humor***, ***surprise***, or ***interest.***

We used a personalized, ***conversational style***, because your brain is tuned to pay more attention when it believes you're in a conversation than if it thinks you're passively listening to a presentation. Your brain does this even when you're *reading*.

We included dozens of ***activities***, because your brain is tuned to learn and remember more when you ***do*** things than when you *read* about things. And we made the paper puzzles and code exercises challenging yet doable, because that's what most people prefer.

We used ***multiple learning styles***, because *you* might prefer step-by-step procedures, while someone else wants to understand the big picture first, and someone else just wants to see an example. But regardless of your own learning preference, *everyone* benefits from seeing th same content represented in multiple ways.

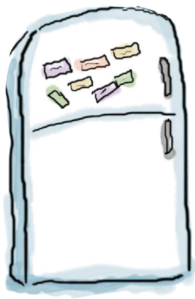
BULLET POINTS

We included content for ***both sides of your brain***, because the more of your brain you engage, the more likely you are to learn and remember, and the longer you can stay focused. Since working one side of the brain often means giving the other side a chance to rest, you can be more productive at learning for a longer period of time.

And we included ***stories*** and exercises that present ***more than one point of view***, because your brain is tuned to learn more deeply when it's forced to make evaluations and judgments.

We included ***challenges***, with exercises, and asked ***questions*** that don't always have a straight answer, because your brain is tuned to learn and remember when it has to *work* at something. Think about it—you can't get your *body* in shape just by *watching* people at the gym. But we did our best to make sure that when you're working hard, it's on the *right* things. That ***you're not spending one extra dendrite*** processing a hard-to-understand example, or parsing difficult, jargon-laden, or overly terse text.

We used ***people***. In stories, examples, pictures, etc., because, well, because *you're* a person. And your brain pays more attention to *people* than it does to *things*.



Here's what YOU can do to bend your brain into submission

So, we did our part. The rest is up to you. These tips are a starting point; listen to your brain and figure out what works for you and what doesn't. Try new things.

Cut this out and stick it on your refrigerator.

1. **Slow down. The more you understand, the less you have to memorize.**

   Don't just *read*. Stop and think. When the book asks you a question, don't just skip to the answer. Imagine that someone really *is* asking the question. The more deeply you force your brain to think, the better chance you have of learning and remembering.

2. **Do the exercises. Write your own notes.**

   We put them in, but if we did them for you, that would be like having someone else do your workouts for you. And don't just *look* at the exercises. **Use a pencil.** There's plenty of evidence that physical activity *while* learning can increase the learning.

3. **Read the "There are no Dumb Questions" sections.**

   That means all of them. They're not optional sidebars—***they're part of the core content!*** Don't skip them.

4. **Make this the last thing you read before bed. Or at least the last challenging thing.**

   Part of the learning (especially the transfer to long-term memory) happens *after* you put the book down. Your brain needs time on its own, to do more processing. If you put in something new during that processing time, some of what you just learned will be lost.

5. **Drink water. Lots of it.**

   Your brain works best in a nice bath of fluid. Dehydration (which can happen before you ever feel thirsty) decreases cognitive function.

6. **Talk about it. Out loud.**

   Speaking activates a different part of the brain. If you're trying to understand something, or increase your chance of remembering it later, say it out loud. Better still, try to explain it out loud to someone else. You'll learn more quickly, and you might uncover ideas you hadn't known were there when you were reading about it.

7. **Listen to your brain.**

   Pay attention to whether your brain is getting overloaded. If you find yourself starting to skim the surface or forget what you just read, it's time for a break. Once you go past a certain point, you won't learn faster by trying to shove more in, and you might even hurt the process.

8. **Feel something.**

   Your brain needs to know that this *matters*. Get involved with the stories. Make up your own captions for the photos. Groaning over a bad joke is *still* better than feeling nothing at all.

9. **Write a lot of code!**

   There's only one way to *really* learn C# so it sticks: **write a lot of code**. And that's what you're going to do throughout this book. Coding is a skill, and the only way to get good at it is to practice. We're going to give you a lot of practice: every chapter has exercises that pose a problem for you to solve. Don't just skip over them—a lot of the learning happens when you solve the exercises. If you get stuck, don't be afraid to **peek at the solution**! We included a solution to each exercise for a reason: it's easy to get snagged on something small. But try to solve the problem before you look at the solution. And definitely get it working before you move on to the next part of the book.

README

This is a learning experience, not a reference book. We deliberately stripped out everything that might get in the way of learning whatever it is we're working on at that point in the book. And the first time through, you need to begin at the beginning, because the book makes assumptions about what you've already seen and learned.

We use a lot of diagrams to make tough concepts easier to understand.

The activities are NOT optional.

The puzzles and activities are not add-ons; they're part of the core content of the book. Some of them are to help with memory, some for understanding, and some to help you apply what you've learned. ***Don't skip the written problems.*** The pool puzzles are the only things you don't *have* to do, but they're good for giving your brain a chance to think about twisty little logic puzzles—and they're definitely a great way to really speed up the learning process.

You should do ALL of the "Sharpen your pencil" activities.

Sharpen your pencil

The redundancy is intentional and important.

One distinct difference in a Head First book is that we want you to *really* get it. And we want you to finish the book remembering what you've learned. Most reference books don't have retention and recall as a goal, but this book is about *learning*, so you'll see some of the same concepts come up more than once.

Activities marked with the Exercise (running shoe) logo are really important! Don't skip them if you're serious about learning C#.

Exercise

Do all the exercises!

The one big assumption that we made when we wrote this book is that you want to learn how to program in C#. So we know you want to get your hands dirty right away, and dig right into the code. We gave you a lot of opportunities to sharpen your skills by putting exercises in every chapter. We've labeled some of them "Do this!"—when you see that, it means that we'll walk you through all of the steps to solve a particular problem. But when you see the Exercise logo with the running shoes, then we've left a big portion of the problem up to you to solve, and we gave you the solution that we came up with. Don't be afraid to peek at the solution—**it's not cheating**! But you'll learn the most if you try to solve the problem first.

We've also included all the exercise solutions source code with the rest of the code from this book. You can find all of it on our GitHub page: https://github.com/head-first-csharp/fourth-edition.

If you see the Pool Puzzle logo, the activity is optional. If you don't like twisty logic, you probably won't like these either.

The "Brain Power" questions don't have answers.

For some of them, there is no right answer, and for others, part of the learning experience is for you to decide if and when your answers are right. In some of the Brain Power questions you will find hints to point you in the right direction.

We're targeting C# 8.0, Visual Studio 2019, and Visual Studio 2019 for Mac.

This book is all about helping you learn C#. The team at Microsoft that develops and maintains C# releases updates to the language. **C# 8.0** is the current version at the time this book is going to into production. We also lean very heavily on Visual Studio, Microsoft's integrated development environment (IDE), as a tool for learning, teaching, and exploring C#. The screenshots in this book were taken with the **latest versions of Visual Studio 2019 and Visual Studio 2019 for Mac** available at the time of production. We included instructions for installing Visual Studio in Chapter 1, and for installing Visual Studio for Mac in the *Visual Studio for Mac Learner's Guide* appendix.

We're on the cusp of C# 9.0, which will be released not long after this book comes out. It has some great new features! The features of C# that are part of the core learning in this book will be unchanged, so you will be able to use this book with future versions of C#. The Microsoft teams that maintain Visual Studio and Visual Studio for Mac routinely release updates, and *very rarely* those changes will affect screenshots in this book.

The Unity Lab sections in this book target **Unity 2020.1**, the latest version of Unity available as this book is going into production. We included instructions for installing Unity in the first Unity Lab.

All of the code in this book is released under an open source license that lets you use it for your own projects. You can download it from our GitHub page (https://github.com/head-first-csharp/fourth-edition). You can also download PDFs with lots of additional learning material covering C# features not included in this book, including some of the latest C# features.

Game design... and beyond

How we use games in this book

You're going to be writing code for lots of projects throughout this book, and many of those projects are games. We didn't do this just because we love games. Games can be ***effective tools for learning and teaching C#***. Here's why:

- Games are **familiar**. You're about to immerse yourself in a lot of new concepts and ideas. Giving you something familiar to grab onto can make the learning process go more smoothly.
- Games make it easier to **explain projects**. When you do any of the projects in this book, the first thing you need to do is understand what we're asking you to build—and that can be surprisingly difficult. When we use games for our projects, that makes it easier for you to quickly figure out what we're asking and dive right into the code.
- Games are **fun to write**! Your brain is much more receptive to new information when you're having fun, so including coding projects where you'll build games is, well, a no-brainer (excuse the pun).

We use games throughout this book to *help you learn broader C# and programming concepts*. They're an important part of the book. You should do all of the game-related projects in the book, even if you're not interested in game development. (The Unity Labs are optional, but strongly recommended.)

The technical review team

Tatiana Mac

Lindsey Bieda

Lisa Kellner

Ashley Godbold

Not pictured (but just as amazing) are the reviewers from the third and second editions: Rebeca Dunn-Krahn, Chris Burrows, Johnny Halife, and David Sterling.

And from the first edition: Jay Hilyard, Daniel Kinnaer, Aayam Singh, Theodore Casser, Andy Parker, Peter Ritchie, Krishna Pala , Bill Meitelski, Wayne Bradney, Dave Murdoch, and especially Bridgette Julie Landers

And super special thanks to our wonderful readers—especially Alan Ouellette, Jeff Counts, Terry Graham, Sergei Kulagin, Willian Piva, and Greg Combow—who let us know about issues that they found while reading our book, and professor Joe Varrasso at Mohawk College for being an early adopter of our book for his course.

Thank you all so much!!

"If I have seen further it is by standing on the shoulders of Giants." – Isaac Newton

The book you're reading has very few errors in it, and we give a TON of credit for its high quality to our amazing team of technical reviewers—the giants who kindly lent us their shoulders. To the review team: we're so incredibly grateful for the work that you all did for this book. Thank you so much!

Lindsey Bieda is a software engineer living in Pittsburgh, PA. She owns more keyboards than any human probably should. When she's not coding she's hanging out with her cat, Dash, and drinking tea. Her projects and ramblings can be found at rarlindseysmash.com.

Tatiana Mac is an independent American engineer who works directly with organizations to build clear and coherent products and design systems. She believes the trifecta of accessibility, performance, and inclusion can work symbiotically to improve our social landscape digitally and physically. When ethically minded, she thinks technologists can dismantle exclusionary systems in favor of community-focused, inclusive ones.

We totally agree with Tatiana on this!

Dr. Ashley Godbold is a programmer, game designer, author, artist, mathematician, teacher, and mom. She works full-time as a software engineering coach at a major retailer and also runs a small indie video game studio, Mouse Potato Games. She is a Unity Certified Instructor and teaches college courses in computer science, mathematics, and game development. She has written *Mastering Unity 2D Game Development (2nd Edition)* and *Mastering UI Development with Unity*, as well as created video courses entitled *2D Game Programming in Unity* and *Getting Started with Unity 2D Game Development*.

And we really want to thank **Lisa Kellner**—this is the 12th (!!!) book that she's reviewed for us. *Thank you so much!*

We also want to give special thanks to **Joe Albahari** and **Jon Skeet** for their incredible technical guidance and really careful and thoughtful review of the first edition, which truly set us up for the success we've had with this book over the years. We benefited so much from your input—even more, in fact, than we realized at the time.

Acknowledgments

Our editor:

Nicole Taché

First and foremost, we want to thank our amazing editor, **Nicole Taché**, for everything you've done for this book. You did so much to help us get it out the door, and gave a ton of incredible feedback. Thank you so much!

The O'Reilly team:

Katherine Tozer

There are so many people at O'Reilly we want to thank that we hope we don't forget anyone. First, last, and always, we we want to thank **Mary Treseler**, who's been with us on our journey with O'Reilly from the very beginning. Special thanks to production editor **Katherine Tozer**, indexer **Joanne Sprott**, and **Rachel Head** for her sharp proofread—all of whom helped get this book from production to press in record time. A huge and heartfelt thanks to **Amanda Quinn**, **Olivia MacDonald**, and **Melissa Duffield** for being instrumental in getting this whole project on track. And a big shout-out to our other friends at O'Reilly: **Andy Oram**, **Jeff Bleiel**, **Mike Hendrickson**, and, of course, **Tim O'Reilly**. If you're reading this book right now, then you can thank the best publicity team in the industry: **Marsee Henon**, **Kathryn Barrett**, and the rest of the wonderful folks at Sebastopol.

And we want to give a shout-out to some of our favorite O'Reilly authors:

- **Dr. Paris Buttfield-Addison**, **Jon Manning**, and **Tim Nugent**, whose book *Unity Game Development Cookbook* is just simply amazing. We're eagerly looking forward to *Head First Swift* from Paris and Jon.
- **Joseph Albahari** and **Eric Johannsen**, who wrote the thoroughly indispensable *C# 8.0 in a Nutshell.*

And finally...

Thank you so much to **Cathy Vice** of Indie Gamer Chick fame for her amazing piece on epilepsy that we used in Chapter 10, and for fighting the good fight for epilepsy advocacy. And *takk skal du ha* to **Patricia Aas** for her phenomenal video on learning C# as a second language that we use in our Code Kata appendix, and for her feedback on how to help advanced learners use this book.

Jon Galloway

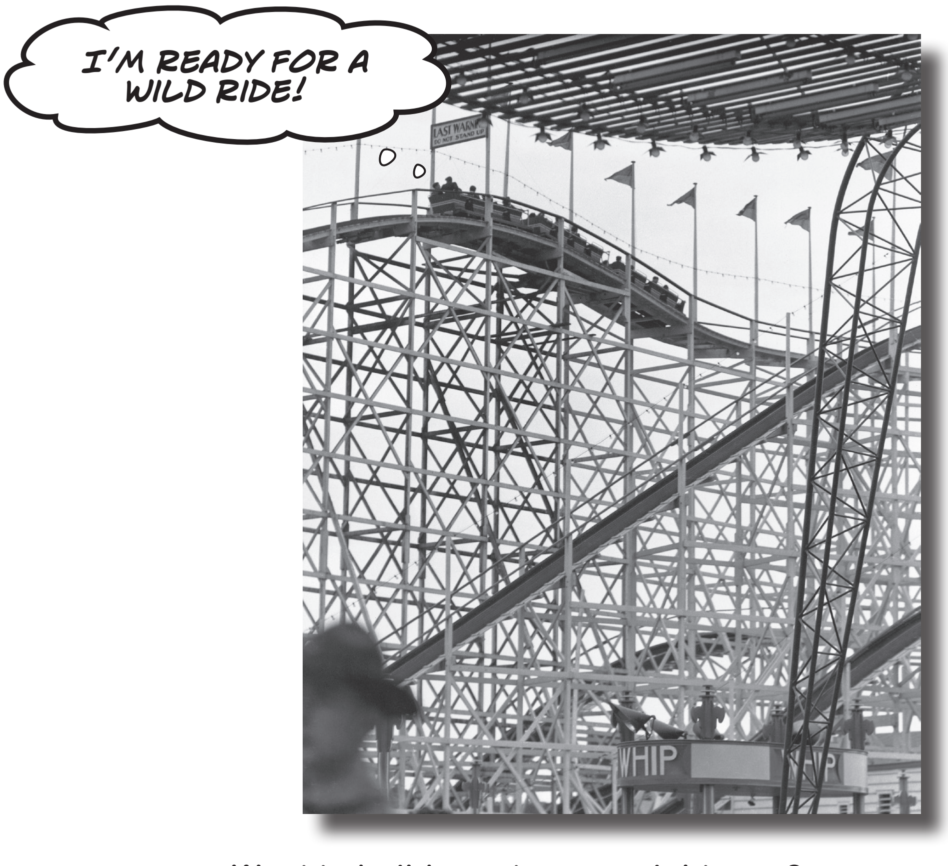
And an ***enormous thank you to our friends at Microsoft*** who helped us so much with this book—your support through this project was amazing. We're so grateful to **Dominic Nahous** (congratulations on the baby!), **Jordan Matthiesen**, and **John Miller** from the Visual Studio for Mac team, and to **Cody Beyer**, who was instrumental in getting our whole partnership with that team started. Thank you to **David Sterling** for an awesome review of the third edition, and **Immo Landwerth** for helping us nail down the topics we should cover in this edition. Extra special thanks to **Mads Torgersen**, Program Manager for the C# language, for all the wonderful guidance and advice he's given us over the years. You all are fantastic.

We're especially grateful to **Jon Galloway**, who provided so much amazing code for the Blazor projects throughout the book. Jon is a senior program manager on the .NET Community Team. He's coauthored several books on .NET, helps run the .NET Community Standups, and cohosts the *Herding Code* podcast. Thank you so much!

O'Reilly online learning

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 **Visual Studio**, you've got an **amazing development environment** with highly intuitive features that make coding as easy as possible. Not only is Visual Studio a great tool for writing code, it's also a **really 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. When you learn C#, you're learning more than just a language. C# unlocks the whole world of .NET, an incredibly powerful open source platform for building all sorts of applications.

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

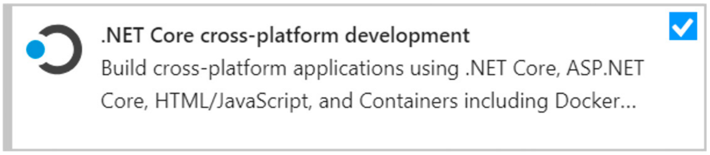
Visual Studio Code is an amazing open source, cross-platform code editor, but it's not tailored to .NET development the way Visual Studio is. That's why we can use Visual Studio throughout this book as a tool for learning and exploration.

Visual Studio is your gateway to C#

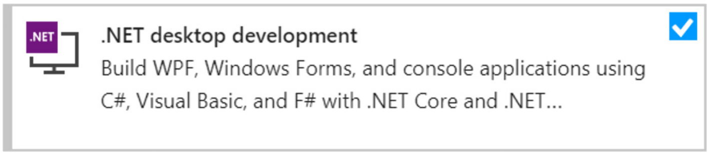
If you haven't installed Visual Studio 2019 yet, this is the time to do it. Go to https://visualstudio.microsoft.com and **download the Visual Studio Community edition**. (If it's already installed, run the Visual Studio Installer to update your installed options.)

If you're on Windows...

Make sure to check the options to install support for .NET Core cross-platform development and .NET desktop development. But don't check the *Game development with Unity* option—you'll be doing 3D game development with Unity later in the book, but you'll install Unity separately.



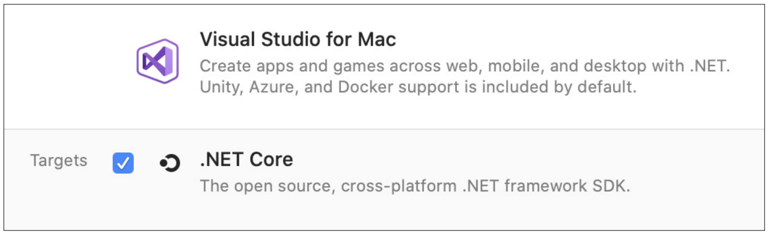
.NET Core cross-platform development ✓
Build cross-platform applications using .NET Core, ASP.NET Core, HTML/JavaScript, and Containers including Docker...



.NET desktop development ✓
Build WPF, Windows Forms, and console applications using C#, Visual Basic, and F# with .NET Core and .NET...

If you're on a Mac...

Download and run the Visual Studio for Mac installer. Make sure the .NET Core target is checked.



Visual Studio for Mac
Create apps and games across web, mobile, and desktop with .NET. Unity, Azure, and Docker support is included by default.

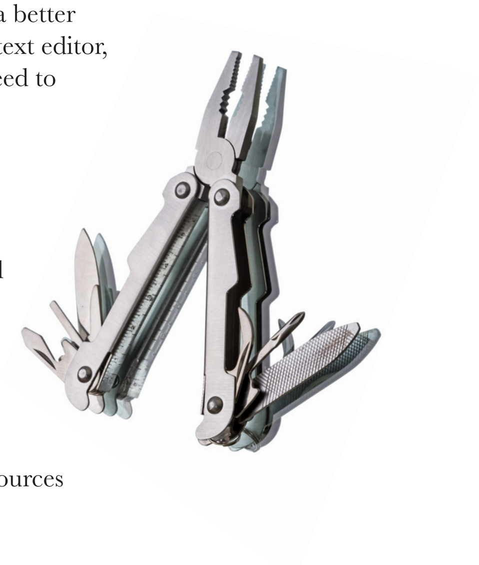
Targets ✓ .NET Core
The open source, cross-platform .NET framework SDK.

You can do the ASP.NET projects on Windows too! Just make sure that the "ASP.NET and web development" option is checked when you install Visual Studio.

Most projects in this book are .NET Core console apps, which work on both Windows and Mac. Some chapters have a project—like the animal matching game later in this chapter—that are Windows desktop projects. For these projects, use the Visual Studio for Mac Learner's Guide appendix. It has a *complete replacement* for Chapter 1, and ASP.NET Core Blazor versions of the other WPF projects.

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).

If you're using Visual Studio for Mac, you'll build the same great-looking apps, but instead of using XAML you'll do it by combining C# with HTML.

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

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.

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*.

We'll often refer to Visual Studio as just "the IDE" throughout this book.

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

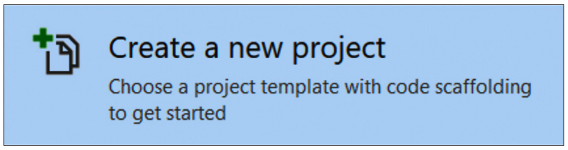
Create your first project in Visual Studio

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.

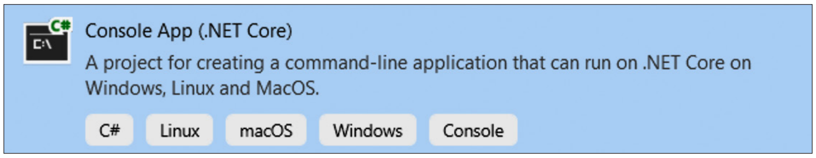
1. **Create a new Console App (.NET Core) project.**
   Start up Visual Studio 2019. When it first starts up, it shows you a "Create a new project" window with a few different options. Choose **Create a new project**. Don't worry if you dismiss the window—you can always get it back by choosing File >> New >> Project from the menu.



Create a new project

Choose a project template with code scaffolding to get started

Choose the **Console App (.NET Core)** project type by clicking on it, then press the **Next** button.

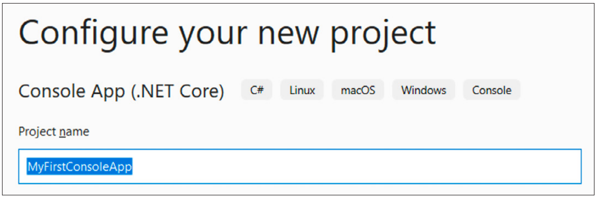


Console App (.NET Core)

A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.

C# Linux macOS Windows Console

Name your project **MyFirstConsoleApp** and click the **Create** button.



Configure your new project

Console App (.NET Core) C# Linux macOS Windows Console

Project name

MyFirstConsoleApp

*If you're using **Visual Studio for Mac**, the code for this project—and <u>all</u> .NET Core Console App projects in this book—will be the same, but some IDE features will be different. Go to the **Visual Studio for Mac Learner's Guide** appendix for the Mac version of this chapter.*

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 a file called *Program.cs* with this code:

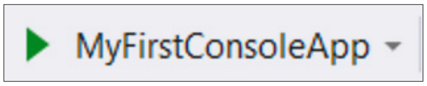
```
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```

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. We'll take a much closer look at classes and methods in Chapter 2.

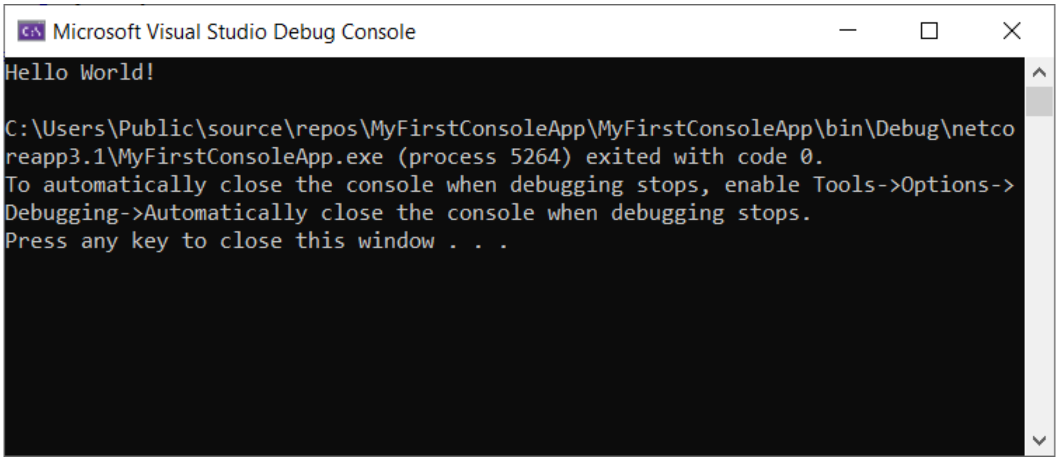
3 **Run your new app.**

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



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 the program:



```
Hello World!

C:\Users\Public\source\repos\MyFirstConsoleApp\MyFirstConsoleApp\bin\Debug\netco
reapp3.1\MyFirstConsoleApp.exe (process 5264) 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 . . .
```

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 throughout the book.

Let's build a game!

You've built your first C# app, and that's great! Now that you've done that, let's build something a little more complex. We're going to build an **animal matching game**, where a player is shown a grid of 16 animals and needs to click on pairs to make them disappear.

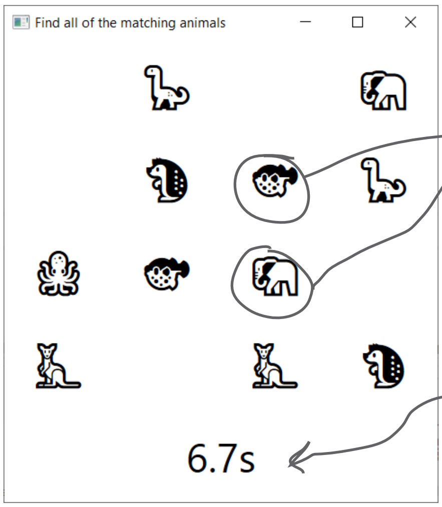
Building different kinds of projects is an important tool in your C# learning toolbox. We chose WPF (or Windows Presentation Foundation) for some of the projects in this book because it gives you tools to design highly detailed user interfaces that run on many different versions of Windows, even very 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 for Mac. See the Visual Studio for Mac Learner's Guide 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.

The Mac versions of the WPF projects use ASP.NET Core. You can build ASP.NET Core projects on Windows, too.

Here's the animal matching 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 matches in as little time as possible.

Your animal matching game is a WPF app

Console apps are great if you just need to input and output text. 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 matching 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.

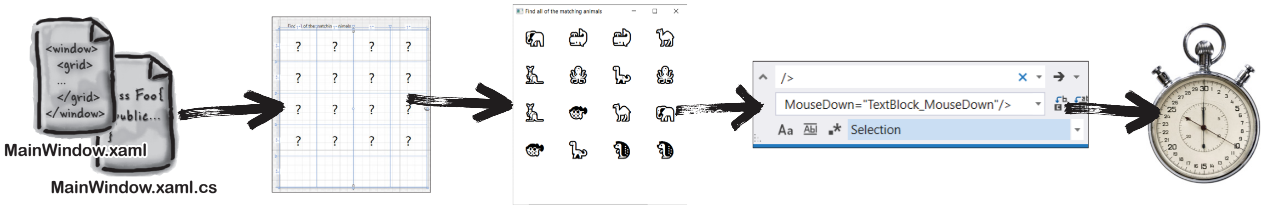
By the time you're done with this project, you'll be a lot more familiar with the tools that you'll rely on throughout this book to learn and explore C#.

Here's how you'll build your game

The rest of this chapter will walk you through building your animal matching 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.

This project can take anywhere from 15 minutes to an hour, depending on how quickly you type. We learn better when we don't feel rushed, so give yourself plenty of time.



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?

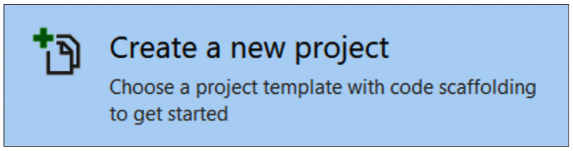
It may seem obvious what a game is. But think about it for a minute—it's not as simple as it seems.

- 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. 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

Go ahead and **start up a new instance of Visual Studio 2019** and create a new project:

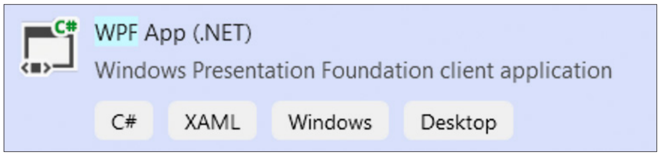


Create a new project

Choose a project template with code scaffolding to get started

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)** and click Next:

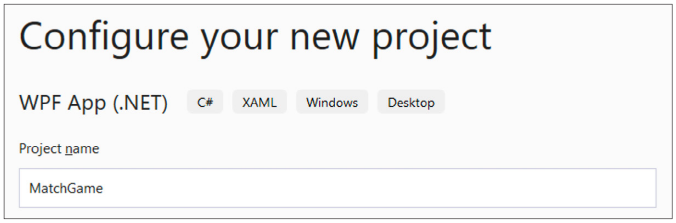


WPF App (.NET)

Windows Presentation Foundation client application

C# XAML Windows Desktop

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



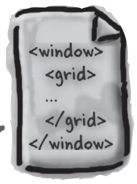
Configure your new project

WPF App (.NET) C# XAML Windows Desktop

Project name

MatchGame

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



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

MainWindow.xaml

Visual Studio created a project folder full of files for you

As soon as you created the new project, Visual Studio 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*.

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

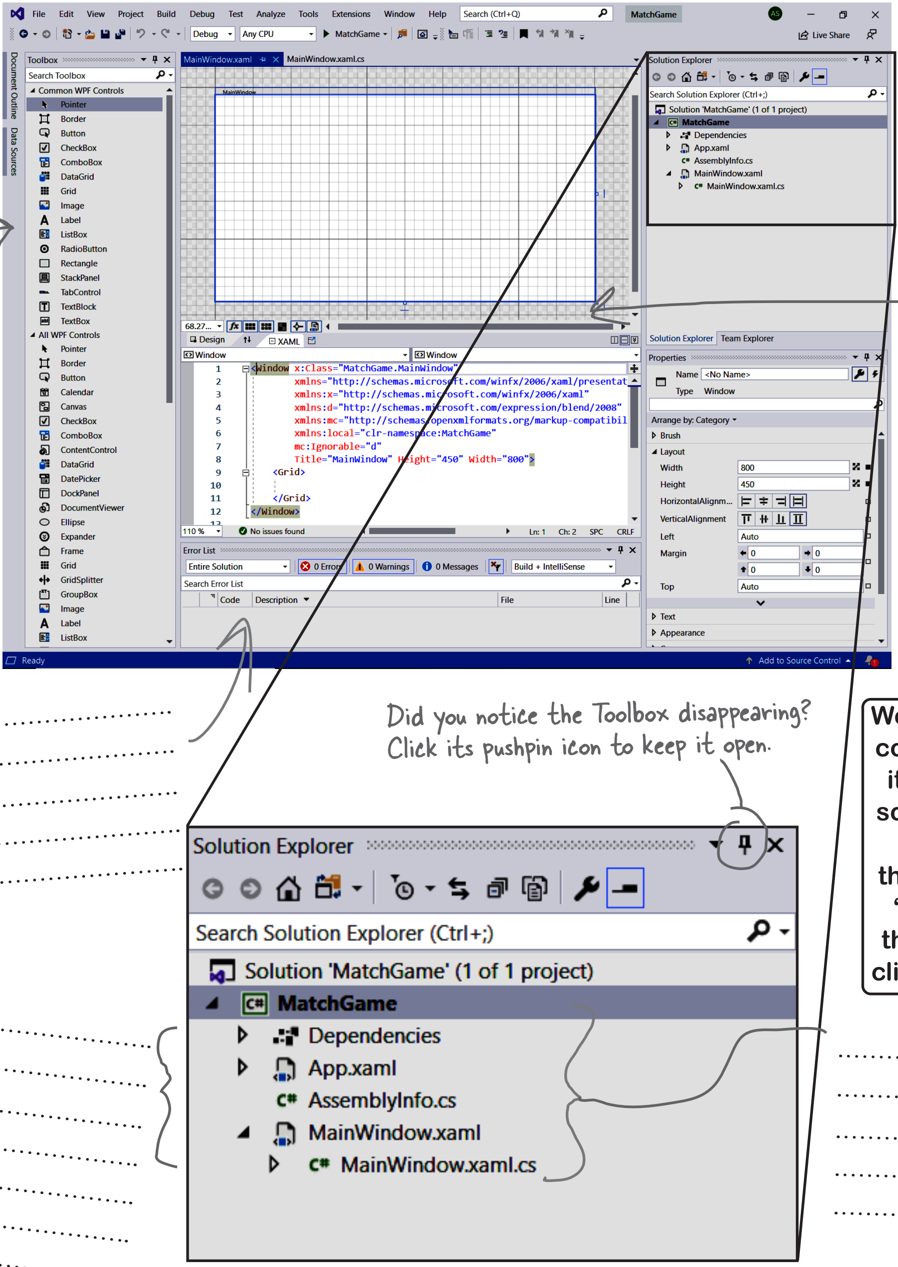
MainWindow.xaml.cs

If you run into any trouble with this project, go to our GitHub page and look for a link to a video walkthrough: https://github.com/head-first-csharp/fourth-edition.

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 Visual Studio IDE does. We've done one to get you started. See if you can take an educated guess at the others.



```
<Window x:Class="MatchGame.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentat
        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-compatibil
        xmlns:local="clr-namespace:MatchGame"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <Grid>

    </Grid>
</Window>
```

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

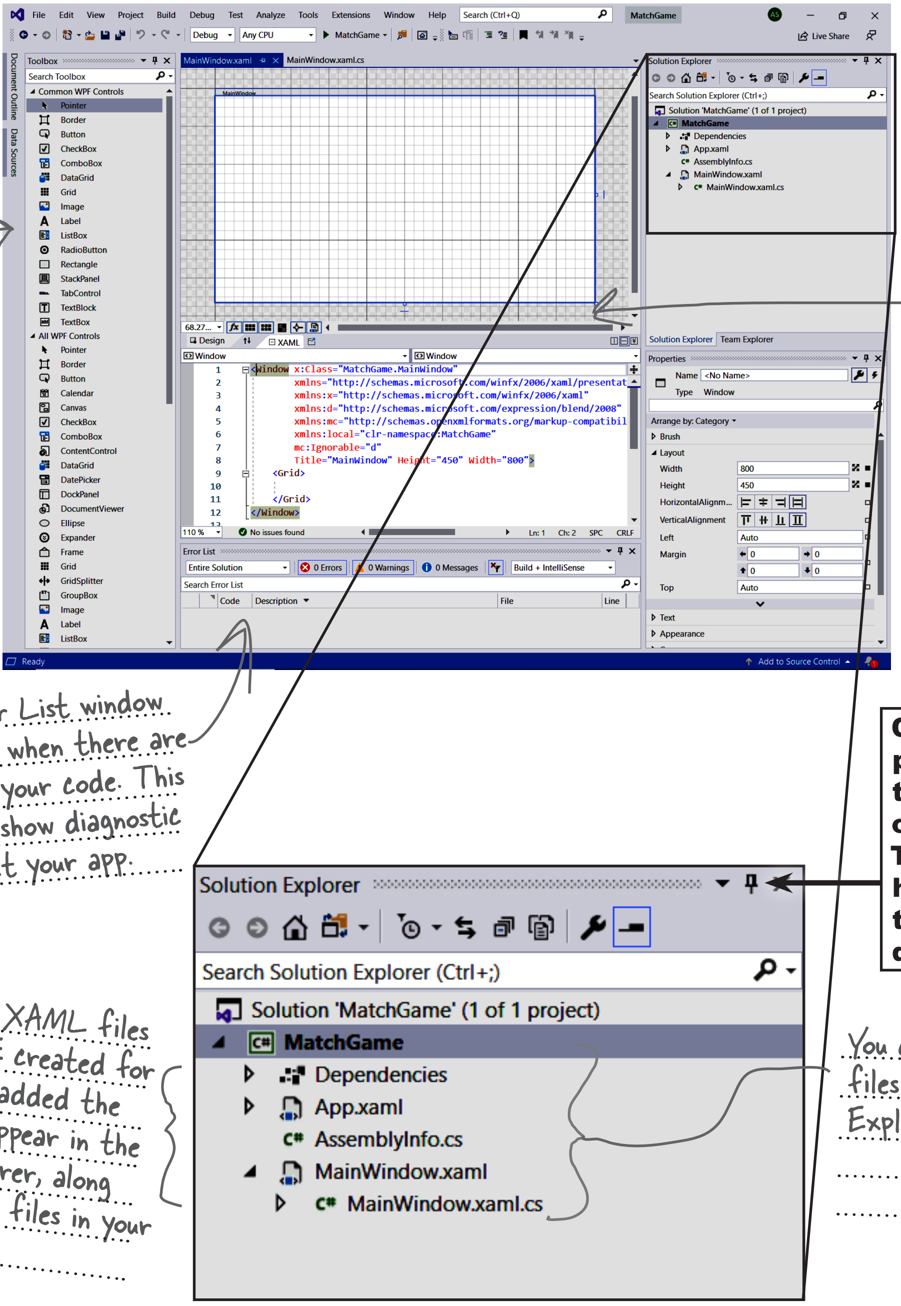
Did you notice the Toolbox disappearing? Click its pushpin icon to keep it open.

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.

And just a quick reminder: we'll use the terms "Visual Studio" and "the IDE" ***interchangeably*** *throughout this book—including on this page.*



```
<Window x:Class="MatchGame.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentat
        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-compatibil
        xmlns:local="clr-namespace:MatchGame"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <Grid>

    </Grid>
</Window>
```

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.

The Properties 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.

Clicking this pushpin icon turns 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.

there are no Dumb Questions

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

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

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

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

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

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

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

Q: 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 go back to your project and type in that tag right after `<Grid>` in your XAML code, a gray 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?

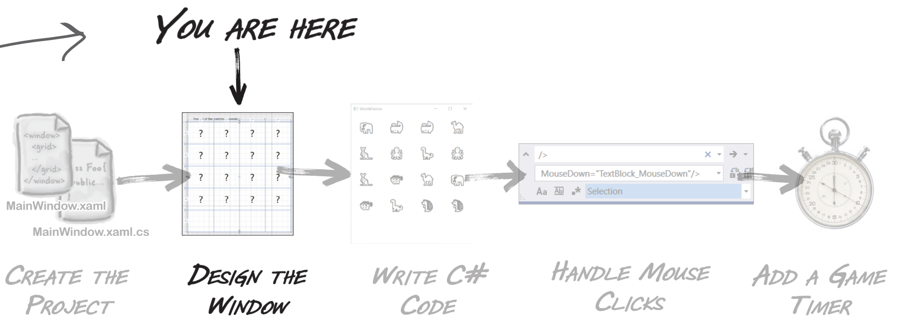
A: If you click on the **Reset Window Layout** command under the Window menu, the IDE will restore the default window layout for you. Then use the **View>>Other Windows** menu to open the Toolbox and Error List windows. That will make your screen look like the ones in this chapter.

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

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

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

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 has 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 markup language that C# developers use to design user interfaces. 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 small window layout in XAML:

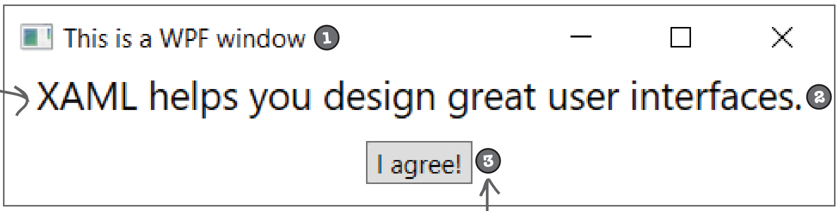
```
<Window x:Class="MyWPFApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="This is a WPF window" Height="100" Width="400">❶
    <StackPanel HorizontalAlignment="Center" VerticalAlignment="Center">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

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

Look for the corresponding numbers in the screenshot below.

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.

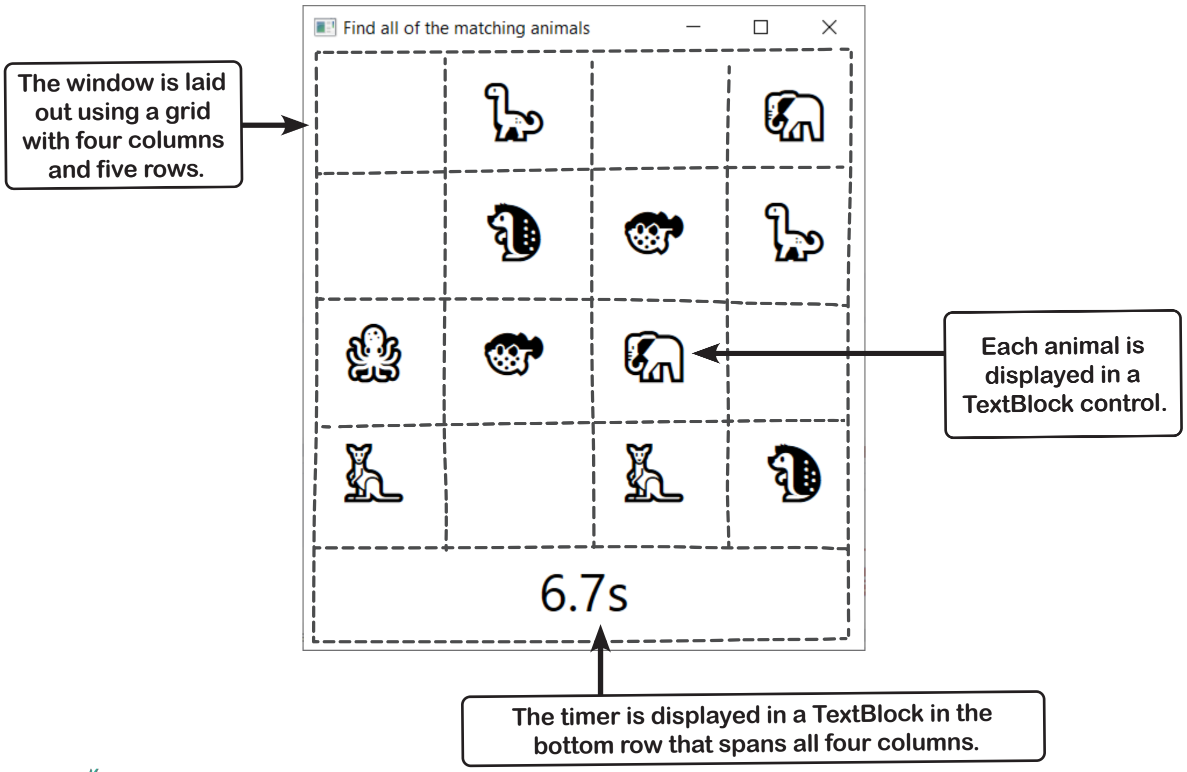


These numbers in the screenshot show the parts of the UI that correspond to the similar numbers we added to the XAML code.

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:


Find all of the matching animals
The window is laid out using a grid with four columns and five rows.
Each animal is displayed in a TextBlock control.
6.7s
The timer is displayed in a TextBlock in the bottom row that spans all four columns.

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 typing. ***So just to be clear:***

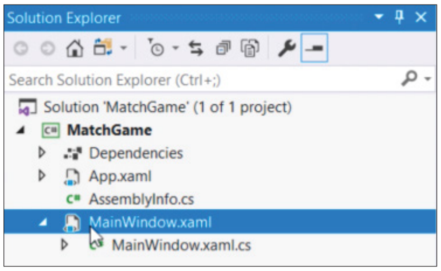
XAML is code that defines the user interface. C# is code that defines the behavior.

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. You'll also get familiar with Visual Studio's XAML designer, a powerful tool for designing great-looking user interfaces for your apps.

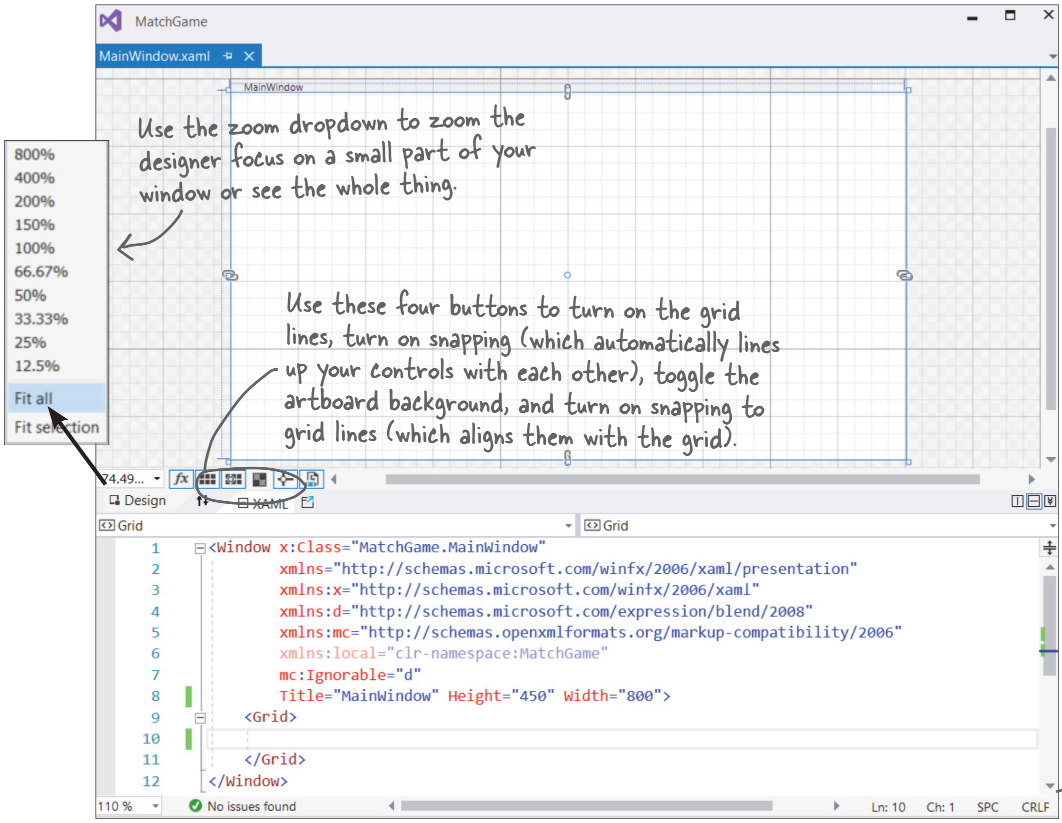
1 **Select the main window.**

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.



```
<Window x:Class="MatchGame.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:MatchGame"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <Grid>

    </Grid>
</Window>
```

Use the zoom dropdown to zoom the designer focus on a small part of your window or see the whole thing.

Use these four buttons to turn on the grid lines, turn on snapping (which automatically lines up your controls with each other), toggle the artboard background, and turn on snapping to grid lines (which aligns them with the grid).

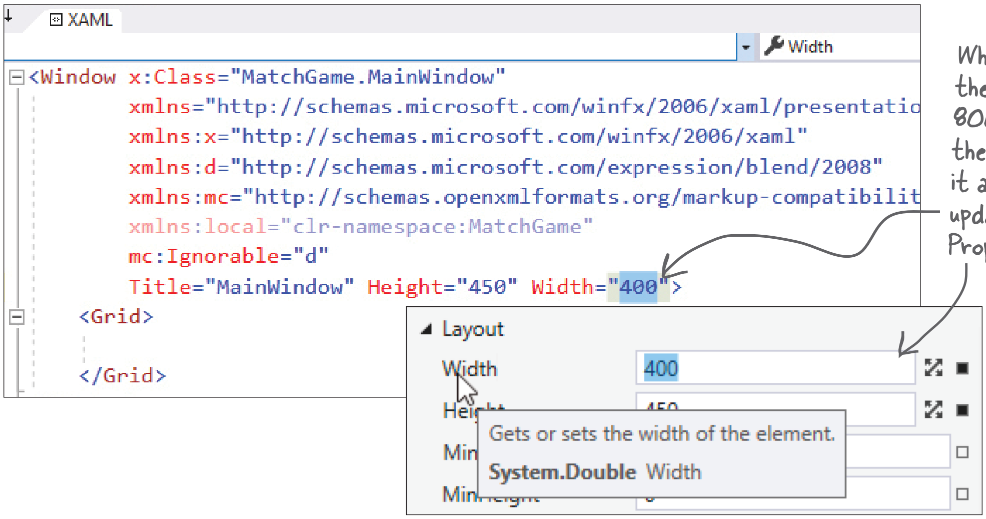
The designer shows you a preview of the window that you're editing. Any change you make here causes the XAML to be updated below.

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 eight 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.



```
<Window x:Class="MatchGame.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentatio
        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-compatibilit
        xmlns:local="clr-namespace:MatchGame"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="400">
    <Grid>

    </Grid>
```

Layout
Width 400
Gets or sets the width of the element.
System.Double Width

When you change the width from 800 to 400 in the XAML editor, it automatically updates the Properties window.

3 **Change the window title.**

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

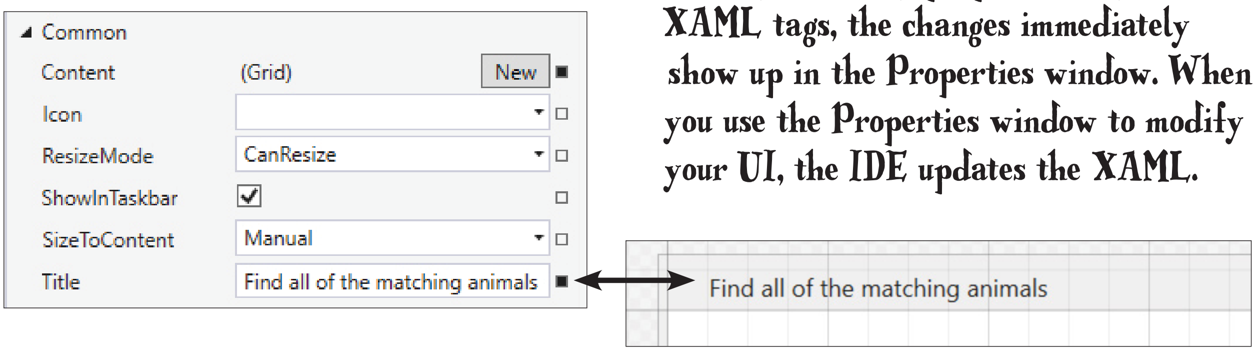
```
Title="MainWindow" Height="450" Width="400">
```

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. When you use the Properties window to modify your UI, the IDE updates the XAML.



Common

Content	(Grid) New
Icon	
ResizeMode	CanResize
ShowInTaskbar	✓
SizeToContent	Manual
Title	Find all of the matching animals

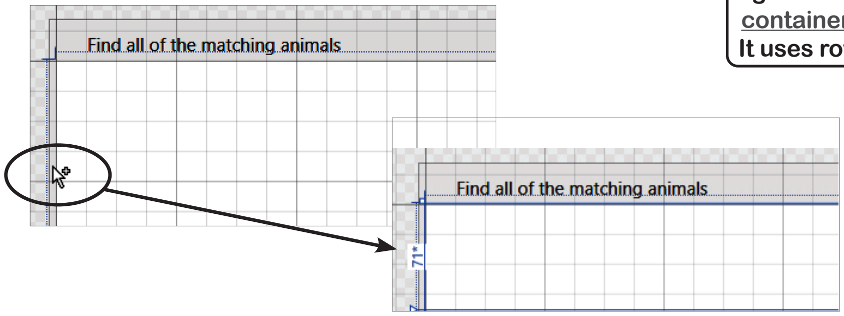
Find all of the matching animals

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.


Find all of the matching animals
Find all of the matching animals
71*

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! Now add the rows and columns:

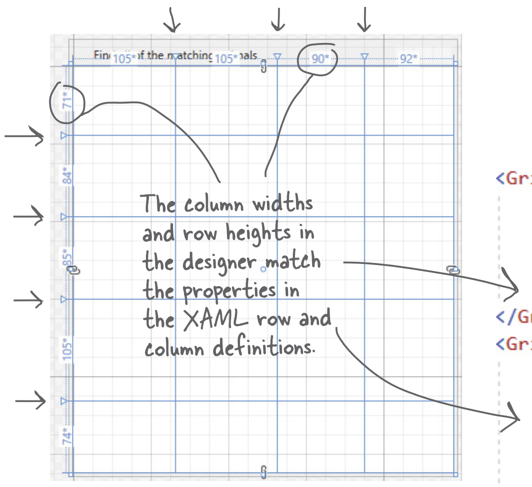
- ★ Repeat four more times to add a total of five rows.
- ★ Hover over the top of the window and click to add four columns. Your window should look like the screenshot below (but your numbers will be different—that's OK).
- ★ Go back to the XAML. It now has a set of `ColumnDefinition` and `RowDefinition`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

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

Things may look a bit different in your IDE.

*All of the screenshots in this book were taken with **Visual Studio Community 2019 for Windows**. If you're using the Professional or Enterprise edition, you might see a few minor differences.*

Don't worry, everything will still work exactly the same.


105*
105*
90*
92*
71*
84*
85*
105*
74*
The column widths and row heights in the designer match the properties in the XAML row and column definitions.

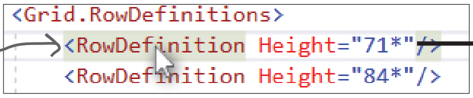
```
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="105*"/>
    <ColumnDefinition Width="105*"/>
    <ColumnDefinition Width="90*"/>
    <ColumnDefinition Width="92*"/>
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
    <RowDefinition Height="71*"/>
    <RowDefinition Height="84*"/>
    <RowDefinition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

Make the rows and columns equal size

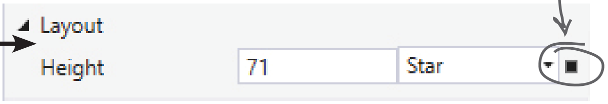
When our game displays the animals for the player to match, we want them to be evenly spaced. Each animal will be contained in a cell in the grid, and the grid will automatically adjust to the size of the window, so we need the rows and columns to all be the same size. Luckily, XAML makes it really easy 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.



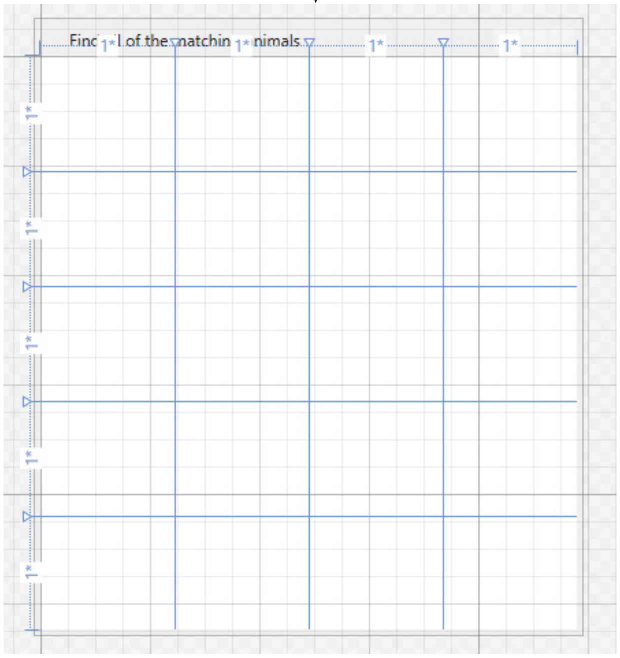
```
<Grid.RowDefinitions>
    <RowDefinition Height="71*"/>
    <RowDefinition Height="84*"/>
```



Go to the Properties window and **click the square** to the right of the Height propert, then **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. 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.

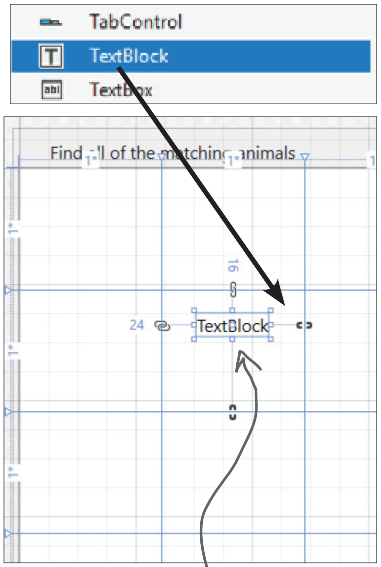
Expand the Common WPF Controls section in the Toolbox and **drag a TextBlock into the cell in the second column and second row**. The IDE will add a `TextBlock` tag between the `Grid` start and end tags:

```
<TextBlock Text="TextBlock"
   HorizontalAlignment="Left" VerticalAlignment="Center"
   Margin="560,0,0,0" TextWrapping="Wrap" />
```

The XAML for this TextBlock has five properties:

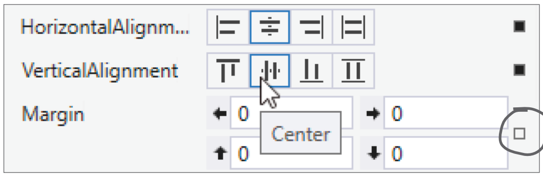
- `Text` tells the TextBlock what text to display in the window.
- `HorizontalAlignment` justifies the text left, right, or center.
- `VerticalAlignment` aligns it to the top, middle, or bottom of its box.
- `Margin` sets its offset from the top, sides, or bottom of its container.
- `TextWrapping` tells it whether or not to add line breaks to wrap the text.

Your properties may be in a different order, and the Margin property will have different numbers because they depend on where in the cell you dragged it. All of these properties can be modified or reset using the IDE's Properties window.



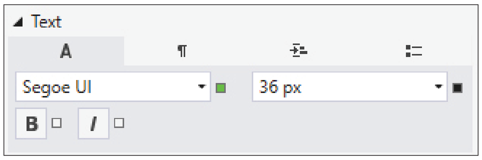
When you drag the control from the toolbox into a cell, the IDE adds a TextBlock to your XAML and sets its row, column, and margin.

We want each animal to be centered. **Click on the label** in the designer, then go to the Properties window and click ▷ Layout to expand the **Layout section**. Click **Center** for both the horizontal and vertical alignment properties, and then use the square at the right of the window **to reset the Margin property**.

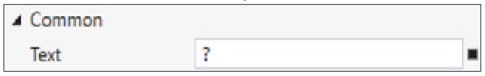


Click this square and choose "Reset" to reset the margins.

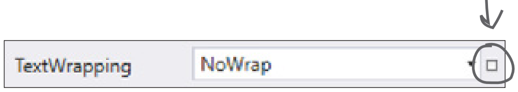
We also want the animals to be bigger, so **expand the Text section** in the Properties window and **change the font size** to `36 px`. Then go to the Common section and change the Text property to `?` to make it display a question mark.



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 side of the window to reset the TextWrapping property.



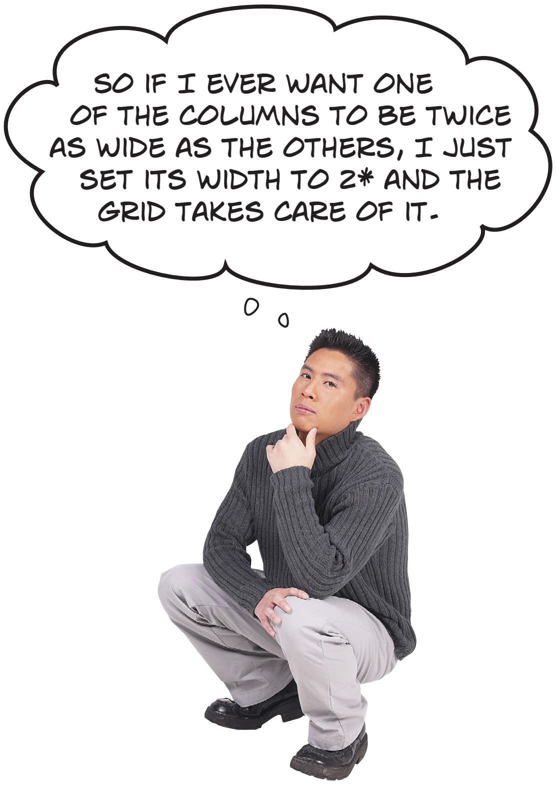
there are no Dumb Questions

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 takes up 1/78th (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.


SO IF I EVER WANT ONE OF THE COLUMNS TO BE TWICE AS WIDE AS THE OTHERS, I JUST SET ITS WIDTH TO 2* AND THE GRID TAKES CARE OF IT.

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 OK 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 16 cells in the grid contains an identical one—to complete this exercise, you'll need to *add 15 more TextBlocks to your app*. A few of things to keep in mind:

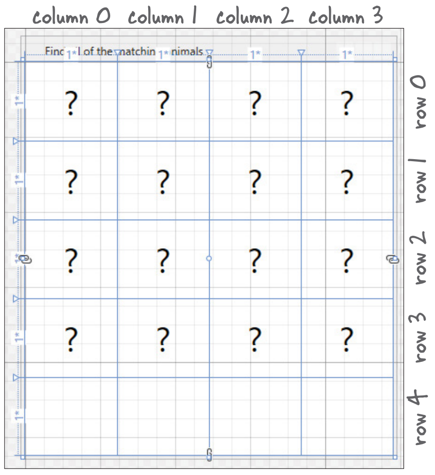
- Rows and columns are numbered starting with 0, which is also the default value. So if you leave out the Grid.Row or Grid.Column property, the TextBlock will appear in the leftmost row or top column.
- You can edit your UI in the designer, or copy and paste the XAML. Try both—see what works for you!

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"/>
```

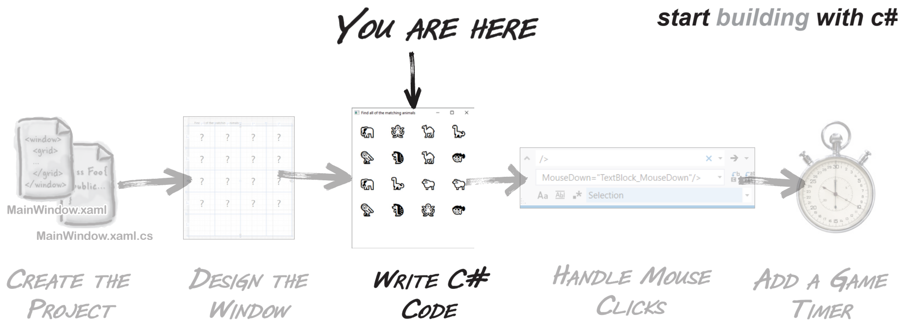
These four TextBlock controls all have their Grid.Row property set to 1, so they're in the second row from the top (because the first row is 0).

```
  <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 OK if you included Grid.Row or Grid.Column properties with a value of 0. We left them out because 0 is the default value.

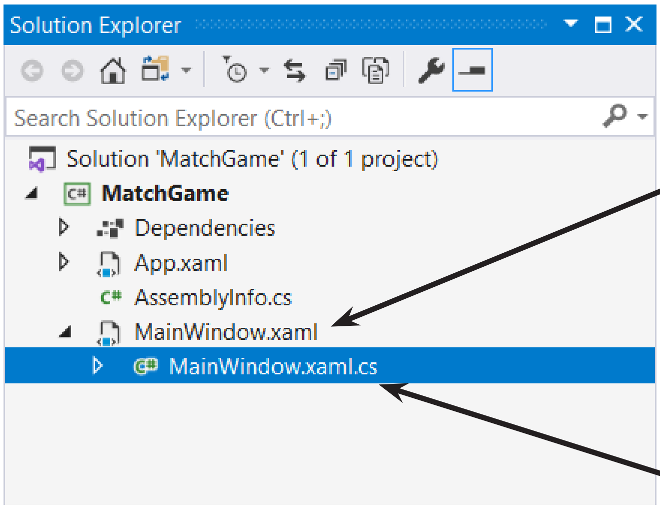
```
  <TextBlock Text="?" FontSize="36" Grid.Row="3" HorizontalAlignment="Center" VerticalAlignment="Center"/>
  <TextBlock Text="?" FontSize="36" Grid.Row="3" Grid.Column="1"
        HorizontalAlignment="Center" VerticalAlignment="Center"/>
  <TextBlock Text="?" FontSize="36" Grid.Row="3" Grid.Column="2"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

There's a lot of code here, but it's really just <u>the same line repeated 16 times</u> with small variations. Every line that starts with `<TextBlock` has the <u>same four properties</u> (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.

Watch it!

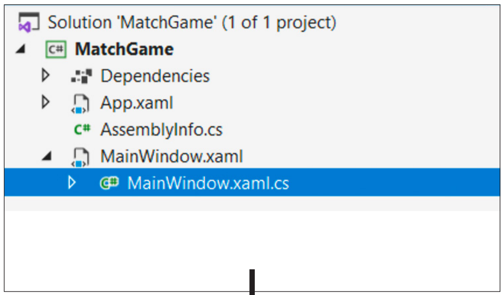
When you enter C# code, it has to be exactly right.

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.*

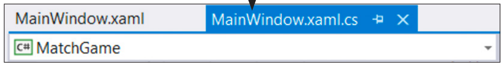
Generate a method to set up the game

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.



Use the tabs at the top of the window to switch between the C# editor and XAML designer.



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.

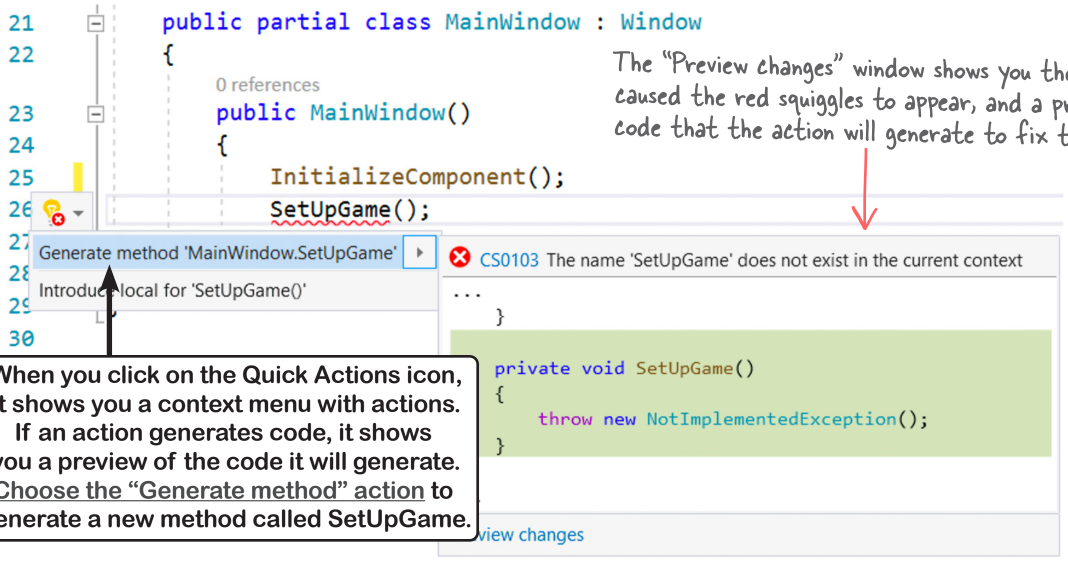
It's OK if you're not 100% clear on what a method is yet.

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.



```
public partial class MainWindow : Window
{
    0 references
    public MainWindow()
    {
        InitializeComponent();
        SetUpGame();
```

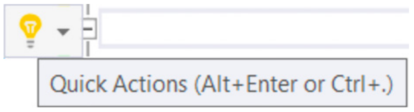
Generate method 'MainWindow.SetUpGame'
Introduce local for 'SetUpGame()'

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

```
...
    }

    private void SetUpGame()
    {
        throw new NotImplementedException();
    }
```

The "Preview changes" window shows you the error that caused the red squiggles to appear, and a preview of the code that the action will generate to fix the error.

When you click on the Quick Actions icon, it shows you a context menu with actions. If an action generates code, it shows you a preview of the code it will generate. Choose the "Generate method" action to generate a new method called SetUpGame.



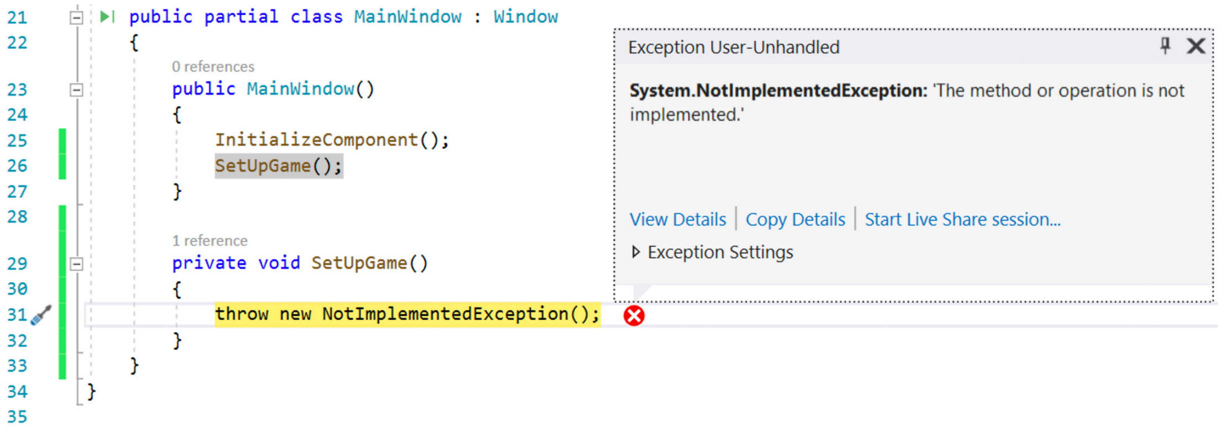
Any time you see the light bulb icon, it's telling you that you've selected code that has a quick action available, which means there's a task that Visual Studio can automate for you. You can either click the light bulb or press Alt+Enter or Ctrl+. (period) to see the available quick actions.

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**:



```
public partial class MainWindow : Window
{
    0 references
    public MainWindow()
    {
        InitializeComponent();
        SetUpGame();
    }

    1 reference
    private void SetUpGame()
    {
        throw new NotImplementedException();
    }
}
```

Exception User-Unhandled

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

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

Exception Settings

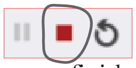
Things may seem like they're broken, but this is actually exactly what we expected to happen! The IDE paused your program, and highlighted the most recent line of code that ran. Take a closer look at it:

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

The method that the IDE generated literally told C# to throw an exception. 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. 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

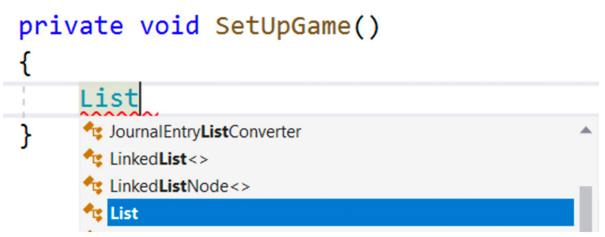
This is a special method called a constructor, and you'll learn more about how it works in Chapter 5.

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.

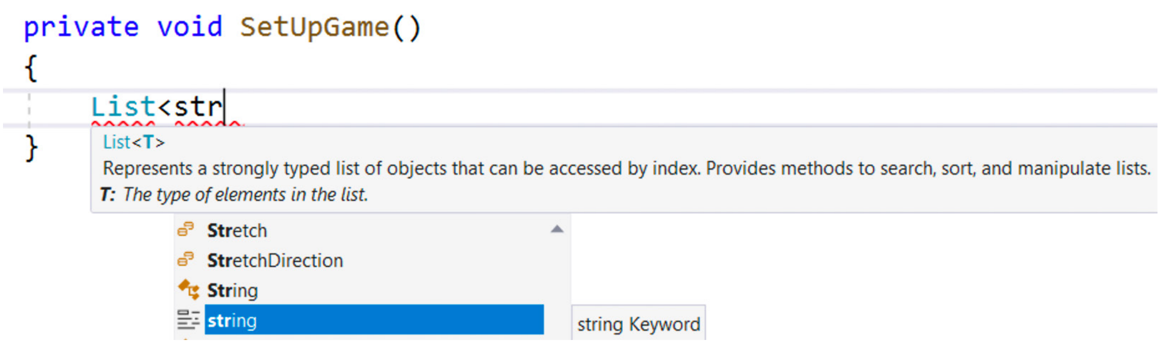
You just used the IDE to help you add a method to your app, but it's OK if you're still not 100% clear on what a method is. You'll learn much more about methods and how C# code is structured in the next chapter.

1. **Start adding 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 you 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
}
```

Choose `List` from the IntelliSense pop-up. 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.

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:

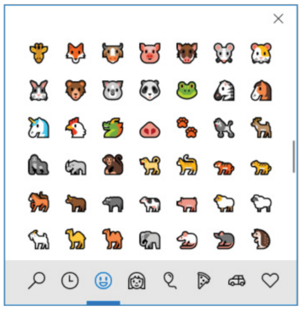
```
List<string> animalEmoji = new List<string>()
{

}
```

While the emoji panel is up, you can type a word like "octopus" and it will replace it with an emoji.

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 <u>pairs</u> 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.



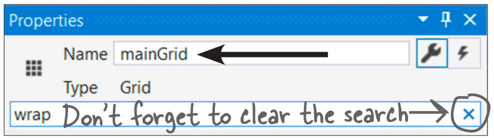
The emoji panel is built into Windows 10. Just press Windows logo key + period to bring it up.

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.



Properties

Name mainGrid

Type Grid

wrap

Don't forget to clear the search

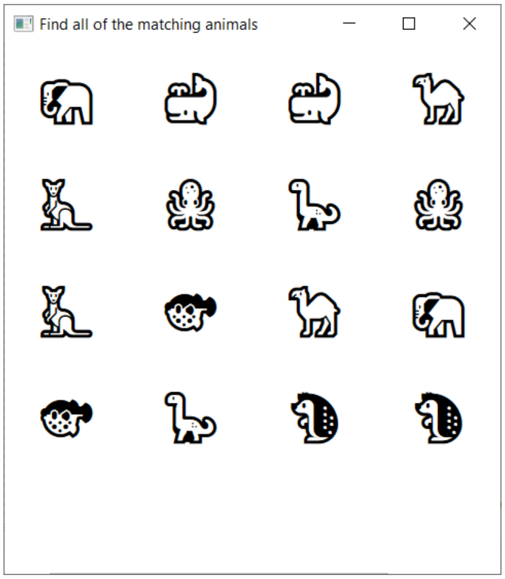
The red squiggly line under `mainGrid` is the IDE telling you there's an error: your program won't build because there's nothing with that name anywhere in the code. **Go back to the XAML editor** and click on the `<Grid>` tag, then go to the Properties window and enter `mainGrid` in the Name box.

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

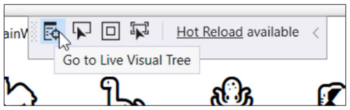
If you get an exception when you run your game, make sure you have exactly 8 pairs of emoji in your animalEmoji list and 16 <TextBlock ... /> tags in your XAML.

Run your program

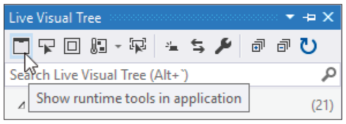
Click the ▶ MatchGame ▾ button in the IDE's toolbar to start your program running. A window will pop up with your eight pairs of animals in random positions:



When your program first runs, you might 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:


Live Visual Tree
Search Live Visual Tree (Alt+`)
Show runtime tools in application
(21)

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. 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

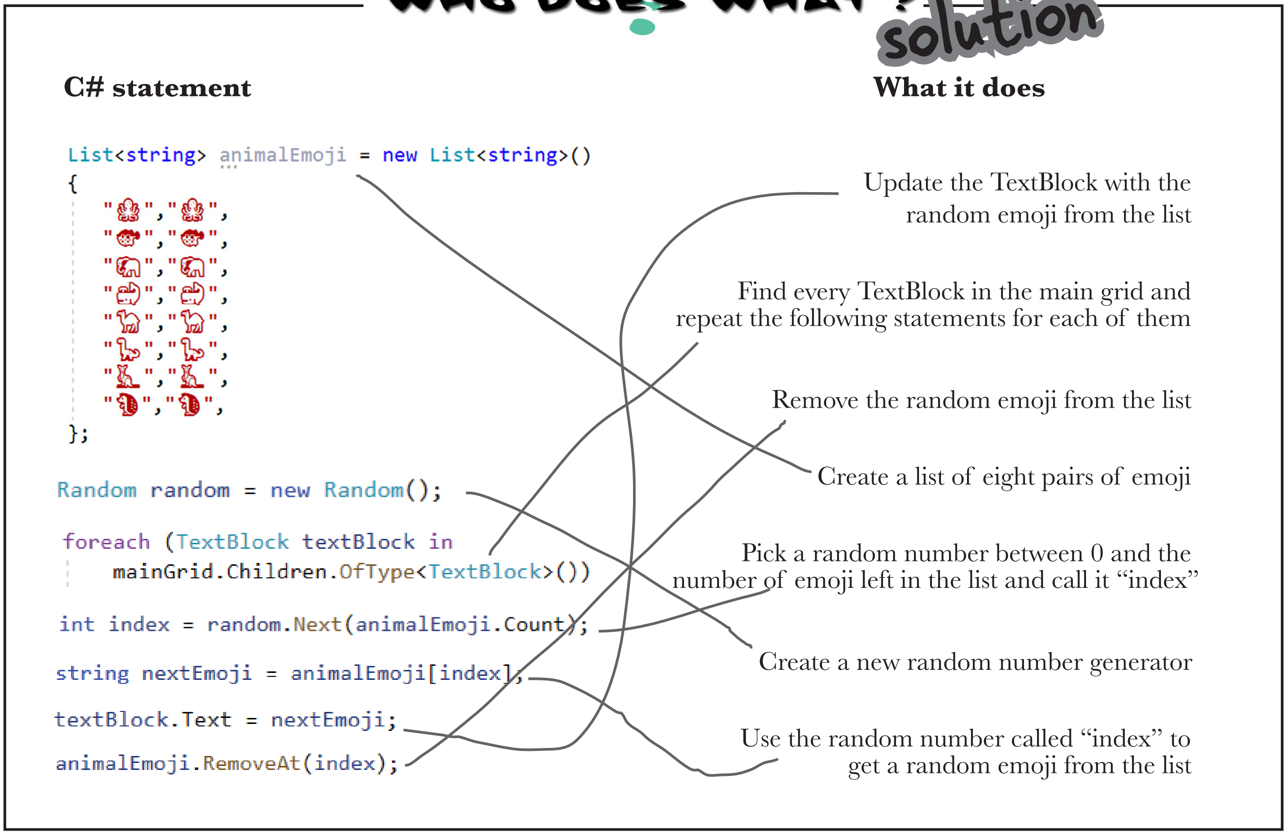
Create a list of eight pairs of emoji

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

```
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"

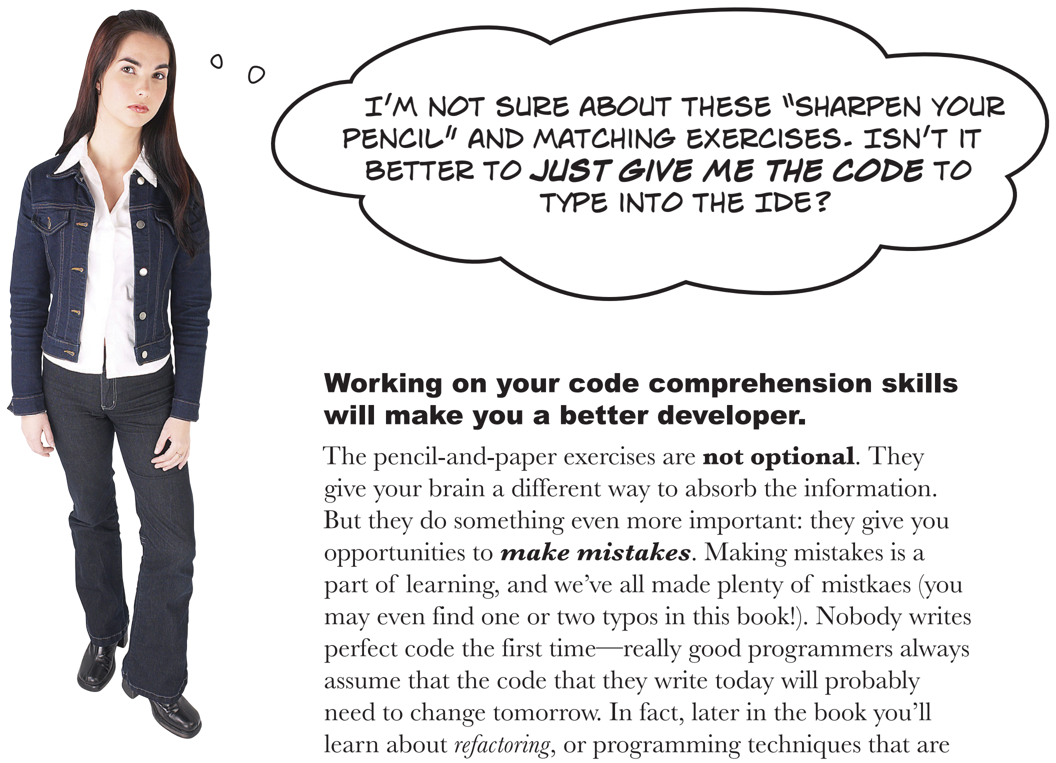
Create a new random number generator

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

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 you'll learn about *refactoring*, or programming techniques that are all about improving your code after you've written it.

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

BULLET POINTS

- Visual Studio is **Microsoft's IDE**—or **integrated development environment**—that simplifies and assists in editing and managing your C# code files.
- **.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 *really convenient* if you could roll back to a previous version of your code? Well, you're in luck! 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. Visual Studio makes it easy for you to add your projects to source control.

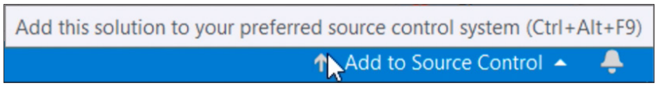
Git is a popular version control system, and Visual Studio will publish your source to any Git **repository** (or **repo**). We think **GitHub** is one of the easiest Git providers to use. You'll need a GitHub account to push code to it, but if you don't have one you'll be able to create it shortly.

Relax

Adding your project to source control is optional.

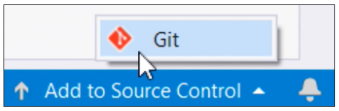
Maybe you're working on a computer on an office network that doesn't allow access to GitHub, the Git provider we're recommending. Maybe you just don't feel like doing it. Whatever the reason, you can skip this step—or, alternatively, you can publish it to a private repository if you want to keep a backup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

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

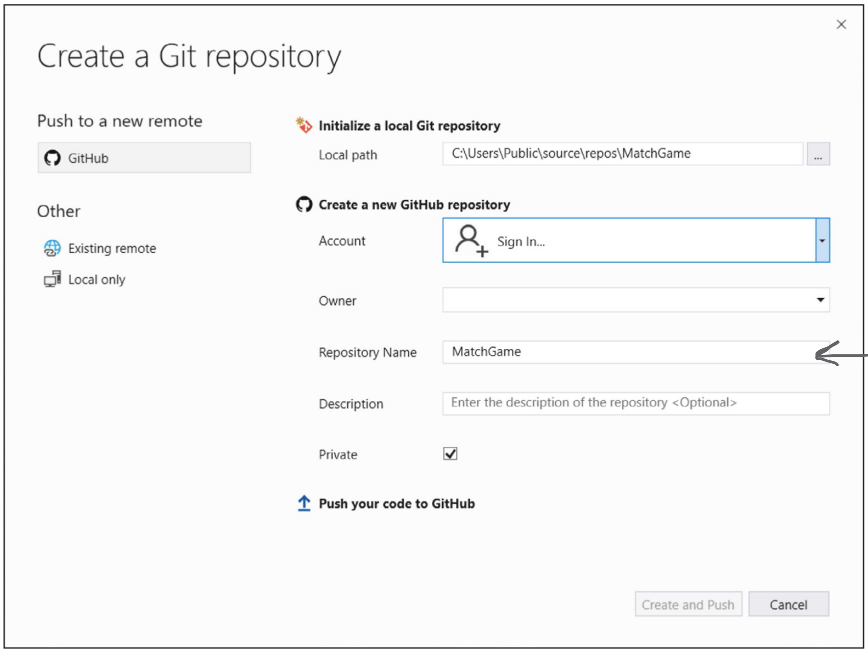

Add this solution to your preferred source control system (Ctrl+Alt+F9)
Add to Source Control

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 a very 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.

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



Click Git. Visual Studio will display the ***Create a Git repository*** window:



Create a Git repository

Push to a new remote
GitHub

Other
Existing remote
Local only

Initialize a local Git repository
Local path: C:\Users\Public\source\repos\MatchGame

Create a new GitHub repository
Account: Sign In...
Owner:
Repository Name: MatchGame
Description: Enter the description of the repository <Optional>
Private: ☑

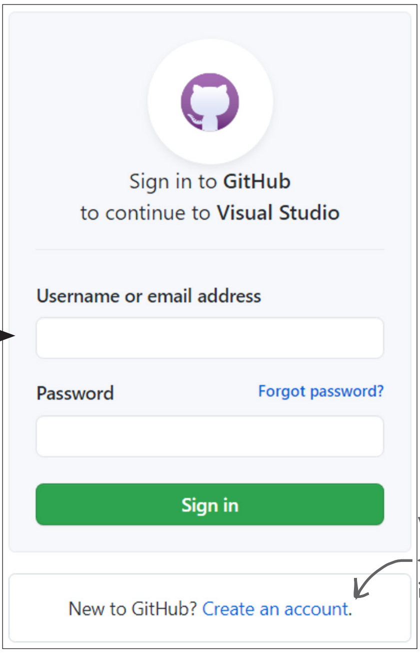
Push your code to GitHub

Create and Push | Cancel

Visual Studio will create a repository in your GitHub account. By default it will have the same name as your project.

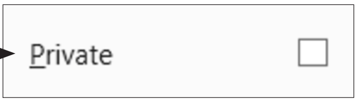
Git is an open-source version control system. There are multiple third-party services like GitHub that provide Git services (like storage space for your code and web access to your repositories). You can go to https://git-scm.com to learn more about Git.

Click . Visual Studio will launch a **GitHub sign-in form** in a browser window. Enter your GitHub username and password. (If you've set up two-factor authentication, you'll also be asked to use it.) Once you log in, you may be prompted to authorize GitHub to grant permissions to Visual Studio—if you get this screen, click the "Authorize GitHub" button to allow Visual Studio to create repositories and push code.

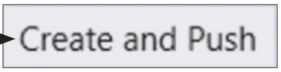

Sign in to GitHub
to continue to Visual Studio
Username or email address
Password
Forgot password?
Sign in
New to GitHub? Create an account.

You can sign up for a free GitHub account if you don't have one.

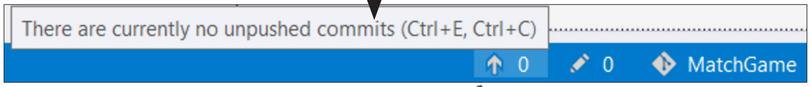
Once you're logged into GitHub, you'll be returned to the "Create a Git Repository" window in Visual Studio. If you want other people to see your code, **uncheck the Private checkbox** to make your new repository public.



Click the **Create and Push button** to create your new GitHub repository and publish your code to it. Once you push to GitHub, the Git status in the status bar will update to show you that there are no **commits**—or saved versions of your code—that haven't been ***pushed*** to a location that's outside of your computer. That means your project is now in sync with a repository in your GitHub account.

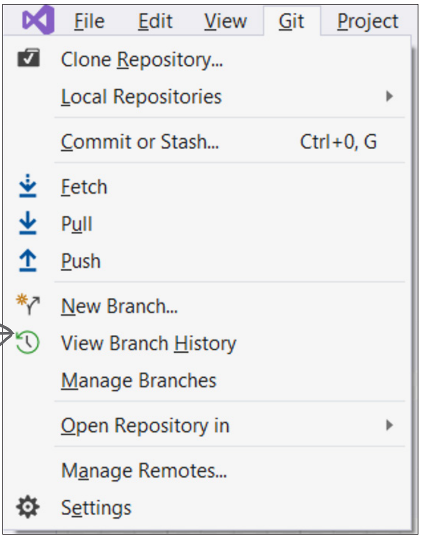


Once you're logged into GitHub, use this button to publish your project to your account.



Keep an eye on the status bar. If you see a number like ↑ 2 it's telling you that there are unpushed commits that you can push to the GitHub repo.

Once you've published your code to GitHub, you can use the commands in the Git menu 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 repo, 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 code that's built alongside your C# code. In the next chapter you'll learn about how you can 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.

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* (we'll get into exactly what a namespace is in the next chapter). 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.

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

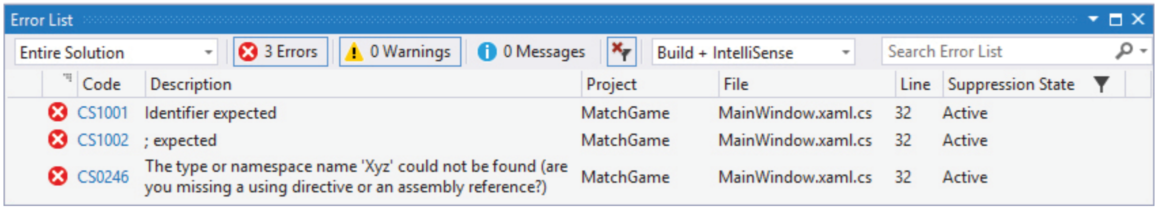
A: Yes. When you get down to it, all C# code works the same way: one statement executes, then the next one, and then the next one. The reason desktop apps seem more complex is because some methods are only called when certain things happen, like when the window is displayed or the user clicks on a button.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

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.

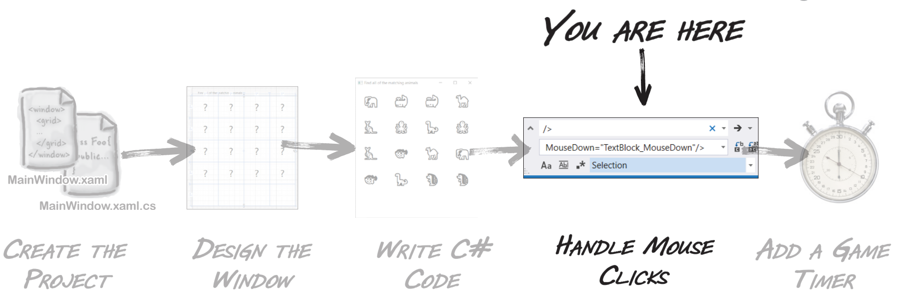
Go to the first line of code in your new SetUpGame method. Press Enter twice, then add this on its own line: `Xyz`

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:



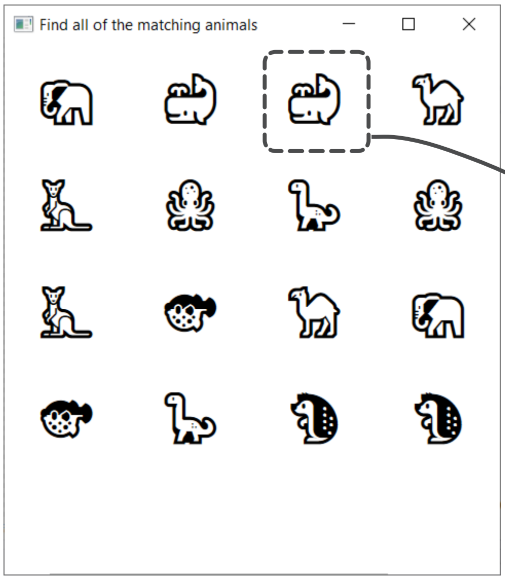
	Code	Description	Project	File	Line	Suppression State
⊗	CS1001	Identifier expected	MatchGame	MainWindow.xaml.cs	32	Active
⊗	CS1002	; expected	MatchGame	MainWindow.xaml.cs	32	Active
⊗	CS0246	The type or namespace name 'Xyz' could not be found (are you missing a using directive or an assembly reference?)	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. The first animal the player clicks on disappears. If the second animal the player clicks on matches the first, that one disappears too. If it doesn't, the first animal reappears. We'll make all of 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.

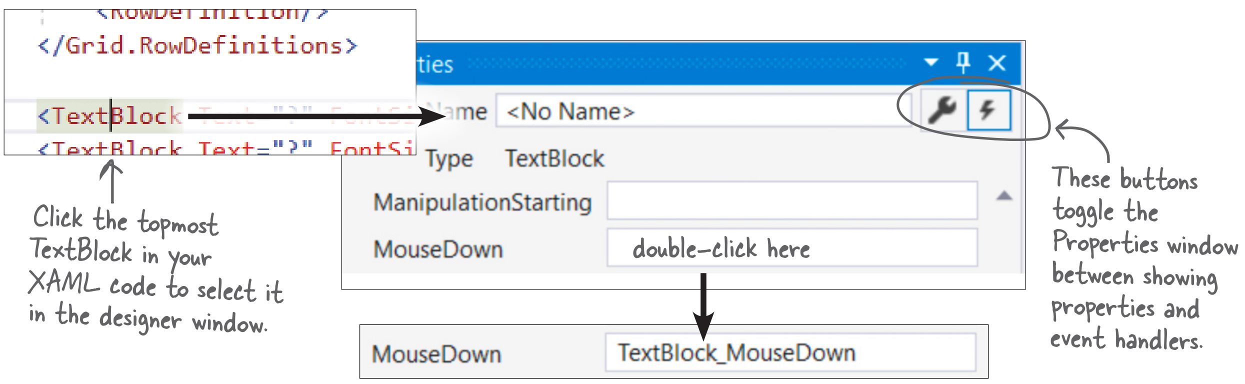
```
TextBlock_MouseDown() {

  /* 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 you 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 the 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 keypresses, 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.**



</Grid.RowDefinitions>
<TextBlock
<TextBlock Text="?" FontSi
ties
Name <No Name>
Type TextBlock
ManipulationStarting
MouseDown double-click here
MouseDown TextBlock_MouseDown
Click the topmost TextBlock in your XAML code to select it in the designer window.
These buttons toggle the Properties window between showing properties and event handlers.

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"/>
```

You might not have noticed that, because the IDE also **added a new method** to the code-behind—the code that's joined with the XAML—and immediately 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. Then we'll need to hook up all of the other TextBlocks so they call it, too.

An <u>event handler</u> is a method that your app calls in response to an event like a mouse click, keypress, 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 OK if you're not 100% right! The goal is to start training your brain to recognize C# as something you can read and make sense of.

```
TextBlock lastTextBlockClicked;
bool findingMatch = false;

private void TextBlock_MouseDown(object sender, MouseButtonEventArgs e)
{
    TextBlock textBlock = sender as TextBlock;
    if (findingMatch == false)
    {
        textBlock.Visibility = Visibility.Hidden;
        lastTextBlockClicked = textBlock;
        findingMatch = true;
    }
    else if (textBlock.Text == lastTextBlockClicked.Text)
    {
        textBlock.Visibility = Visibility.Hidden;
        findingMatch = false;
    }
    else
    {
        lastTextBlockClicked.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 == lastTextBlockClicked.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 OK if you're not 100% right! The goal is to start training your brain to recognize C# as something you can read and make sense of.

```
TextBlock lastTextBlockClicked;
bool findingMatch = false;

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

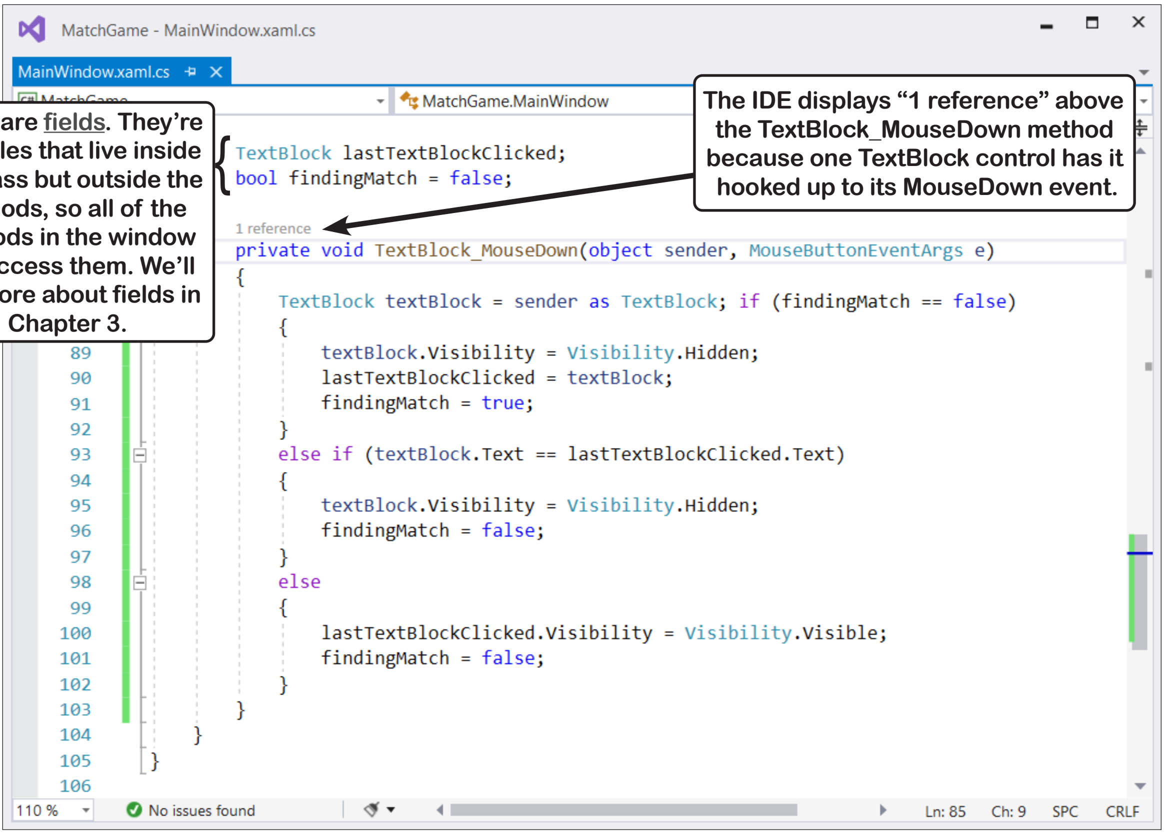
Add the TextBlock_MouseDown code

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:



```
TextBlock lastTextBlockClicked;
bool findingMatch = false;

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

These are <u>fields</u>. They're variables that live inside the class but outside the methods, so all of the methods in the window can access them. We'll talk more about fields in Chapter 3.

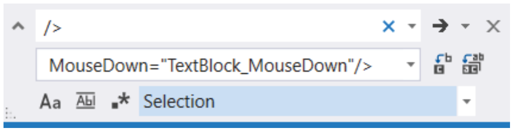
The IDE displays "1 reference" above the TextBlock_MouseDown method because one TextBlock control has it hooked up to its MouseDown event.

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. 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.



```
/>
MouseDown="TextBlock_MouseDown"/>
Selection
```

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.

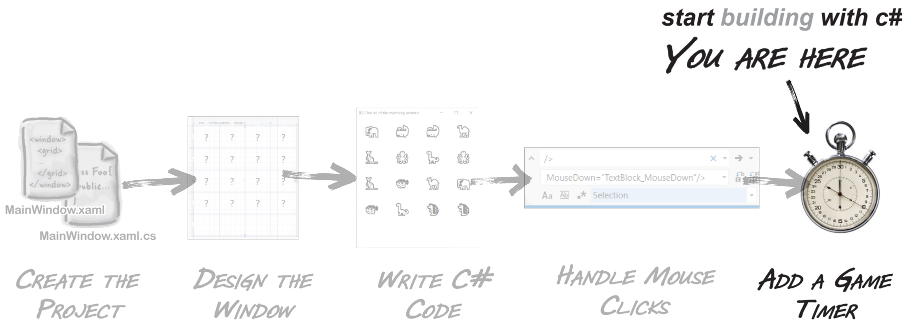
   Make sure that the method now shows **16 references** in the C# editor (choose Build Solution from the Build menu to update it). If you see 17 references, you accidentally attached the event handler to the Grid. You definitely don't want that—if you do, you'll get an exception when you click an animal.

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.

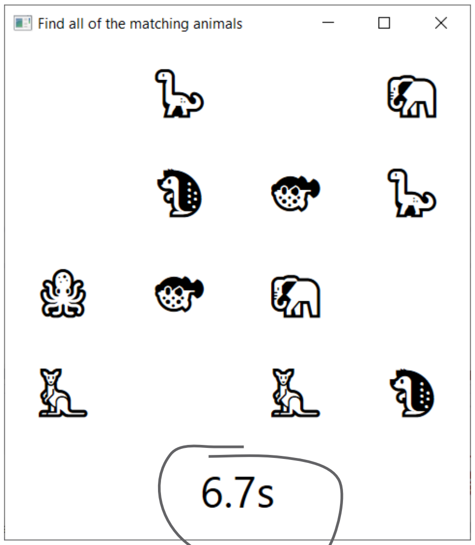
BRAIN POWER

You've reached a checkpoint in your project! Your game might not be finished yet, 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!

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

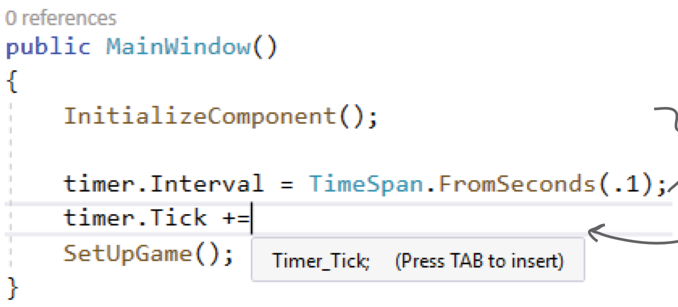
```
namespace MatchGame
{
    using System.Windows.Threading;
```

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. Click at the beginning of the line where you call the SetUpGame method to move the editor's cursor there. 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:

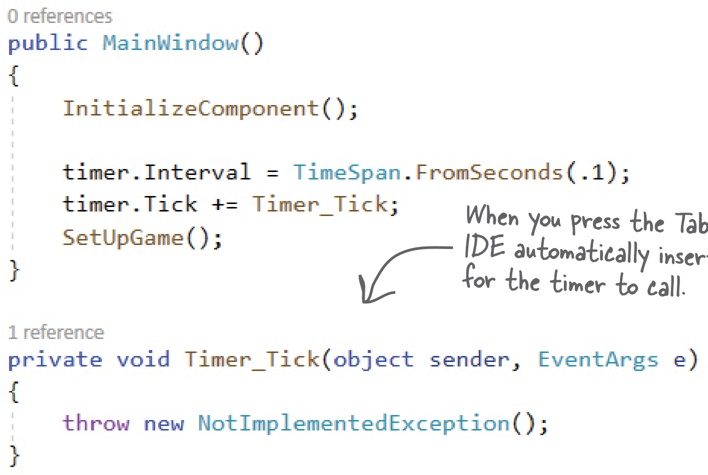


```
0 references
public MainWindow()
{
    InitializeComponent();

    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.

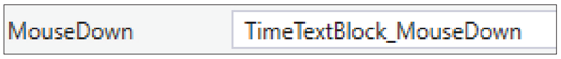
4. Press the Tab key. The IDE will finish the line of code and add a Timer_Tick method:

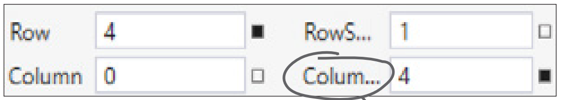


```
0 references
public MainWindow()
{
    InitializeComponent();

    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:


MouseDown TimeTextBlock_MouseDown


Row 4 ■ RowS... 1 □
Column 0 □ Colum... 4 ■

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.

- ★ Drag a **TextBlock** into the lower-left square.
- ★ Use the **Name box** at the top of the Properties window to give it the name **`timeTextBlock.`**
- ★ Reset its **margins**, **center** it in the cell, and set the **FontSize** property to 36px and **Text** property to "Elapsed time" (just like you did with other controls).
- ★ Find the **ColumnSpan** property and set it to 4.
- ★ Add a **MouseDown event handler** called TimeTextBlock_MouseDown.

Here's what the XAML will look like—carefully compare it with your code in the IDE:

```
<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).

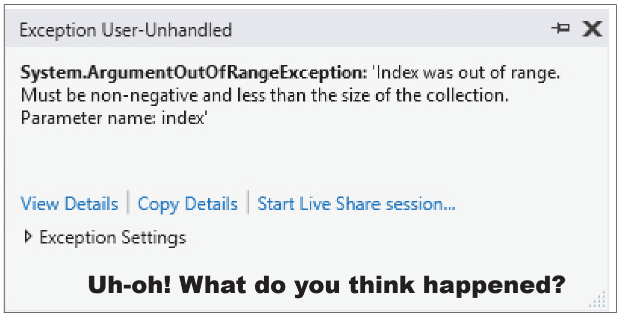
7 Now you have everything you need to finish the Timer_Tick method, which updates the new TextBlock with the elapsed time and stops the timer once the player has found all of the matches:

```
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 first look closely at the error message and highlighted line in the IDE.

Can you guess what caused the error?


Exception User-Unhandled

System.ArgumentOutOfRangeException: 'Index was out of range. Must be non-negative and less than the size of the collection.
Parameter name: index'

Uh-oh! What do you think happened?

Use the debugger to troubleshoot the exception

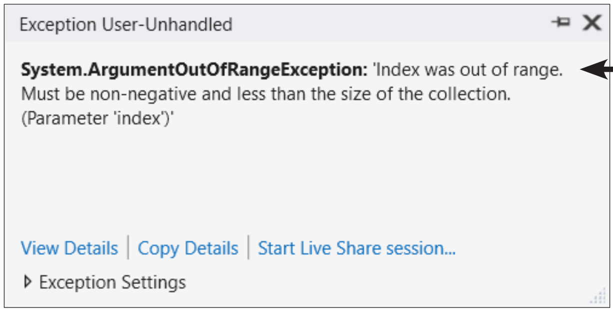
You might have heard the word "bug" before. You might have even said something like this to your friends at some point in the past: "That game is really buggy, it has so many glitches." Every bug has an explanation—everything in your program happens for a reason—but not every bug is easy to track down.

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

Debug this!

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:

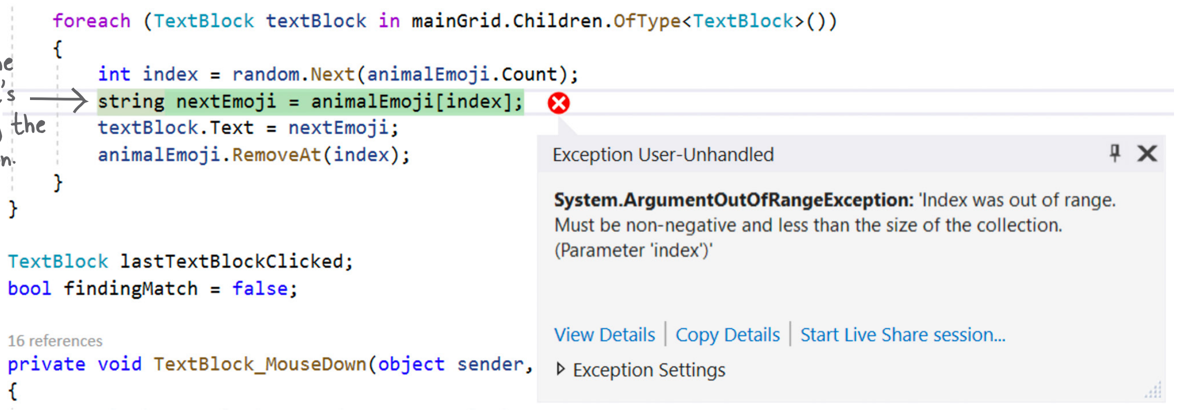

Exception User-Unhandled

System.ArgumentOutOfRangeException: 'Index was out of range. Must be non-negative and less than the size of the collection. (Parameter 'index')'

An exception is C#'s way of telling you that something went wrong when your code was running. Every exception has a type: this one is an ArgumentOutOfRangeException. Exceptions also have useful messages to help you figure out what went wrong. This exception's message says, "Index was out of range." That's useful information to help us figure out what went wrong.

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

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 lastTextBlockClicked;
bool findingMatch = false;

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

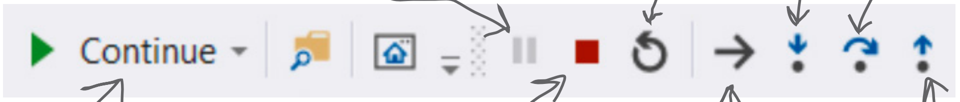
Exception User-Unhandled

System.ArgumentOutOfRangeException: 'Index was out of range. Must be non-negative and less than the size of the collection. (Parameter 'index')'

This 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.



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

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 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.

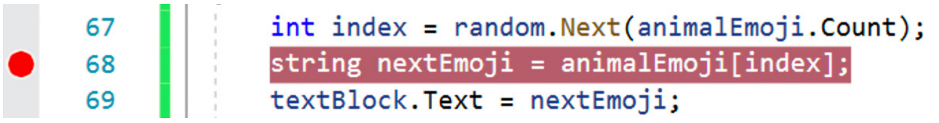
You've already used the Stop Debugging button to halt your app.

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:



```
int index = random.Next(animalEmoji.Count);
string nextEmoji = animalEmoji[index];
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.)

____

____

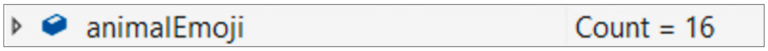
Sharpen your pencil Solution

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.

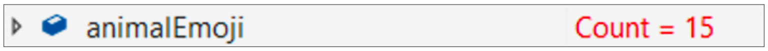
3 **Gather evidence so you can figure out what's causing the problem.**

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:



```
animalEmoji                Count = 16
```

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



```
animalEmoji                Count = 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?

Behind the Scenes

foreach is a kind of <u>loop</u> that runs on every element in a collection.

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 number in a list. ints

The above `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.

We're introducing a new concept here—but just briefly, so there's no mystery about how your code works. We'll talk a lot more about loops in Chapter 2. Then in Chapter 3 we'll come back to `foreach` loops, and you'll write one 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 when you first saw it. We find rereading code once you have more context really helps get it into your brain…so don't worry if the concepts still seem a little nebulous now.

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. What caused it to run out of emoji to add?

Sleuth it out

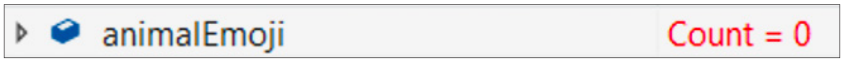
Your program worked before you made the most recent change. Then you added a TextBlock…and then it stopped working. Right inside of a loop that iterates through all of the TextBlocks. A clue…how very, very interesting.

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:



```
animalEmoji    Count = 0
```

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. 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.

Add this code to fix the bug.

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.

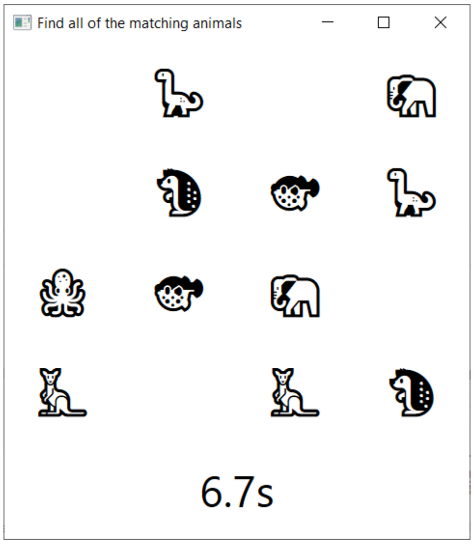
Then add this statement to the middle block of the `if/else` in TextBlock_MouseDown:

```
else if (textBlock.Text == lastTextBlockClicked.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?

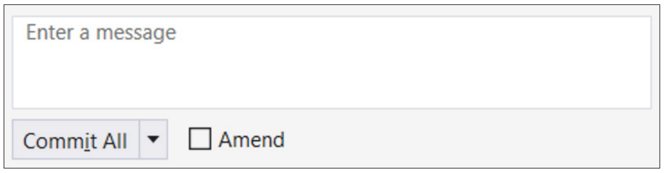


Go to https://github.com/head-first-csharp/fourth-edition to view and download the complete code for this project and all of the other projects in this book.

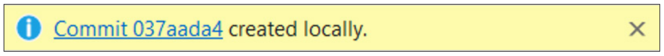
Update your code in source control

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. Choose **Commit or Stash... (Ctrl+0, G)** from the Git menu. Enter a **commit message** that describes what changed.



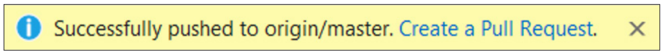
2. Press the **Commit All button**, and Visual Studio will display a message that a commit was created locally.



Commit 037aada4 created locally.

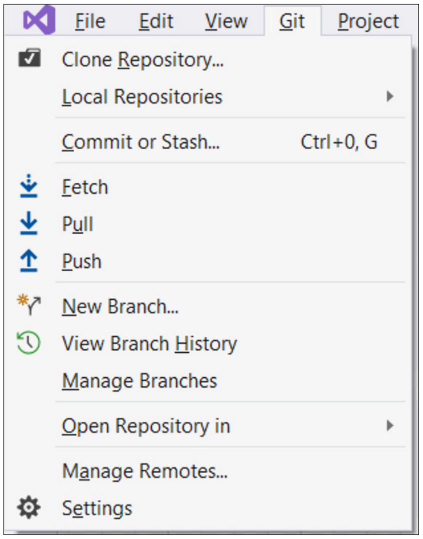
Every commit is given a *unique identifier*, a string of numbers and letters (like `037aada4` in our screenshot).

3. Choose **Push** from the Git menu to push your commit back to the repository. It will show you a message when your push is complete.

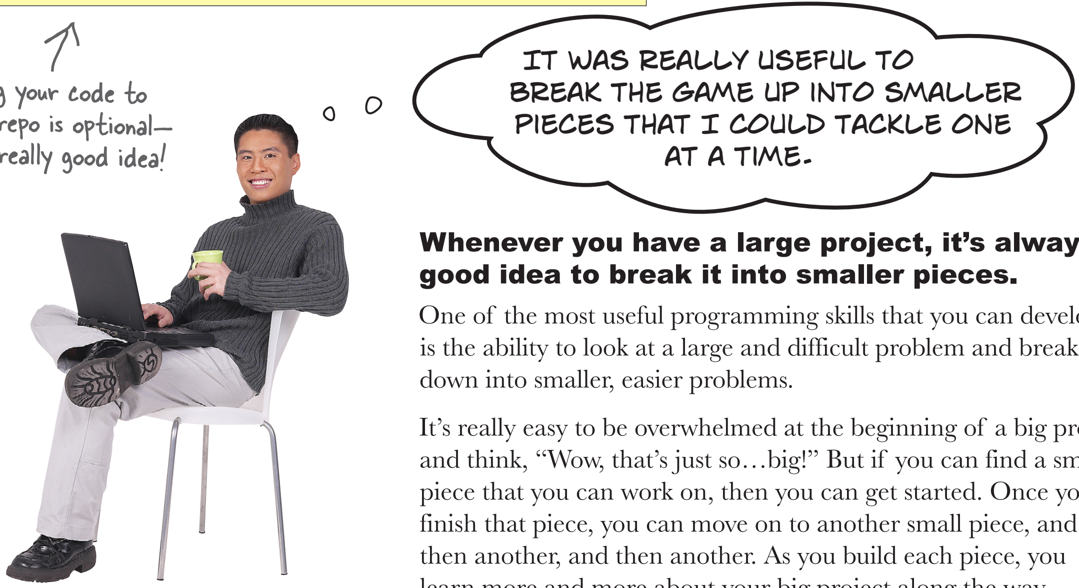


Successfully pushed to origin/master. Create a Pull Request.

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



You can use the commands in the Git menu to create a new commit with your latest code changes and push it to your Git repo.


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

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

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

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

Even better ifs...

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

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

MINI Sharpen your pencil

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

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

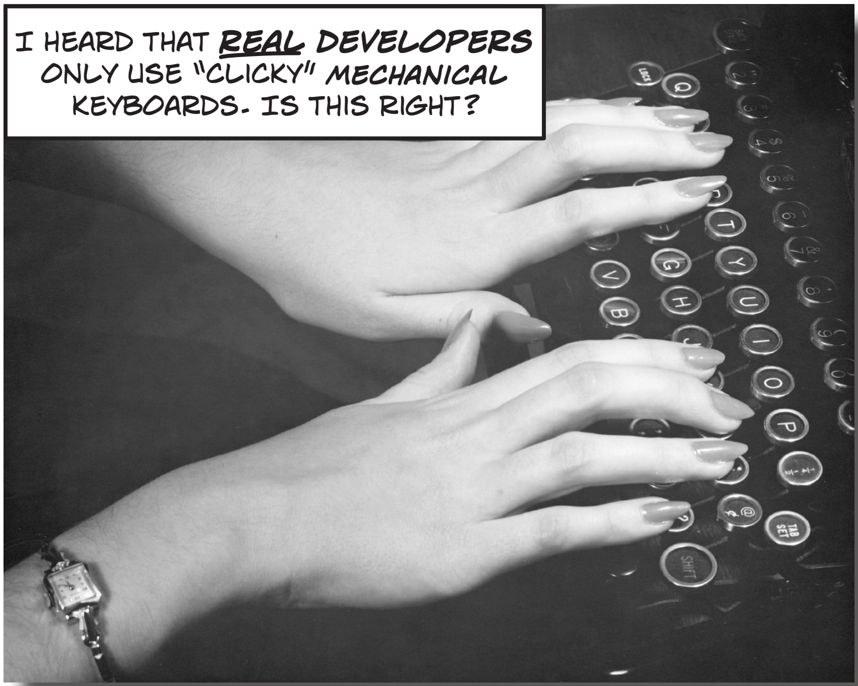
BULLET POINTS

- Visual Studio 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 back up your code and keep track of all changes that you've made.
- You can commit your code to a **remote Git repository**. We use GitHub for the repository with the source code for all of the projects in this book.

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

2 dive into C#

Statements, classes, and code


I HEARD THAT REAL DEVELOPERS ONLY USE "CLICKY" MECHANICAL KEYBOARDS. IS THIS RIGHT?

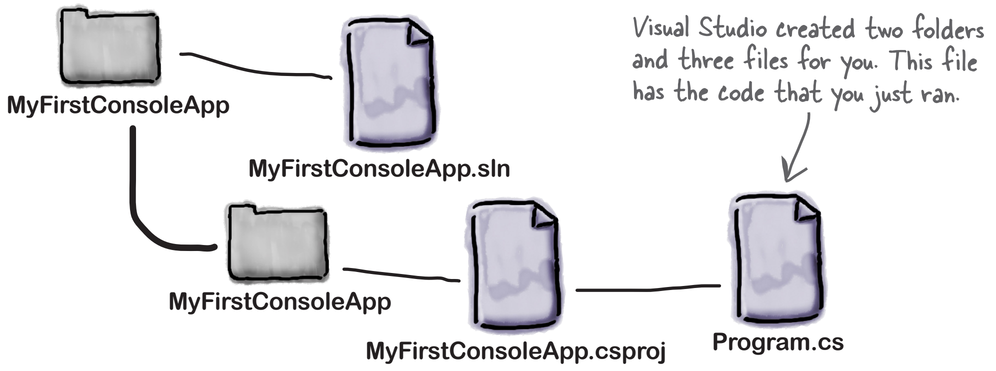
You're not just an IDE user. You're a developer.

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.

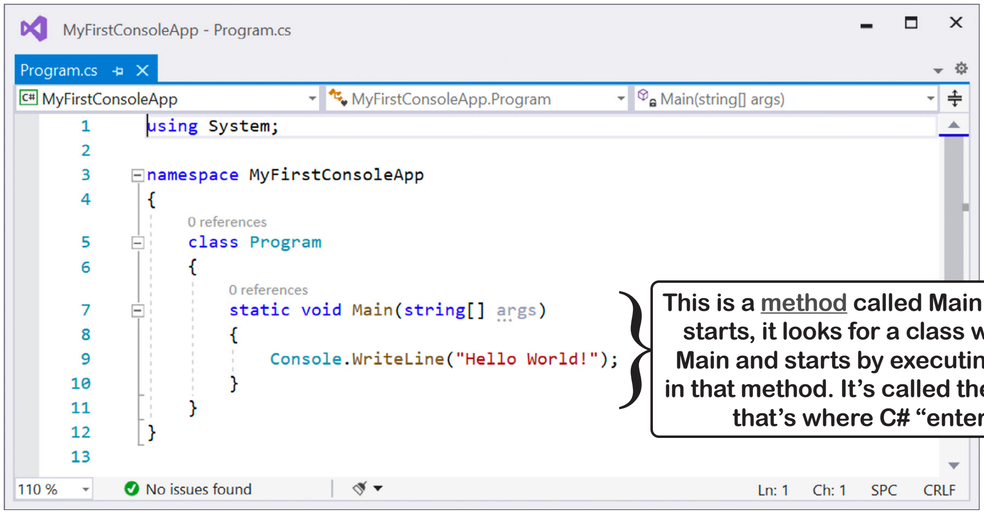
(And for the record, you can be a **real developer** no matter what kind of keyboard you prefer. The only thing you need to do is **write code**!)

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 project and named it MyFirstConsoleApp. When you did that, Visual Studio created two folders and three files.


MyFirstConsoleApp
MyFirstConsoleApp.sln
MyFirstConsoleApp
MyFirstConsoleApp.csproj
Program.cs
Visual Studio created two folders and three files for you. This file has the code that you just ran.

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


MyFirstConsoleApp - Program.cs
Program.cs
MyFirstConsoleApp
MyFirstConsoleApp.Program
Main(string[] args)
0 references
0 references
110 %
No issues found
Ln: 1
Ch: 1
SPC
CRLF

```
using System;

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

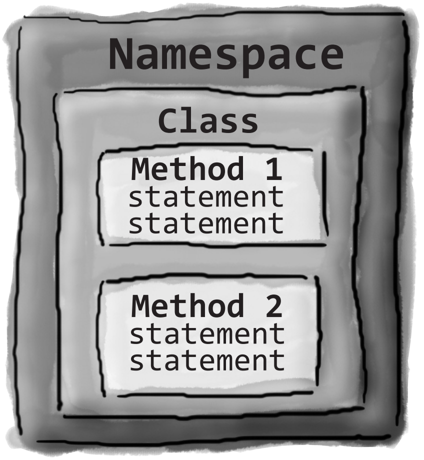
This is a screenshot of Visual Studio for Windows. If you're using macOS the screen will look a little different, but the code will be the same.

This is a method called Main. When a console app starts, it looks for a class with a method called Main and starts by executing the first statement in that method. It's called the entry point because that's where C# "enters" the program.

- ★ 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. 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

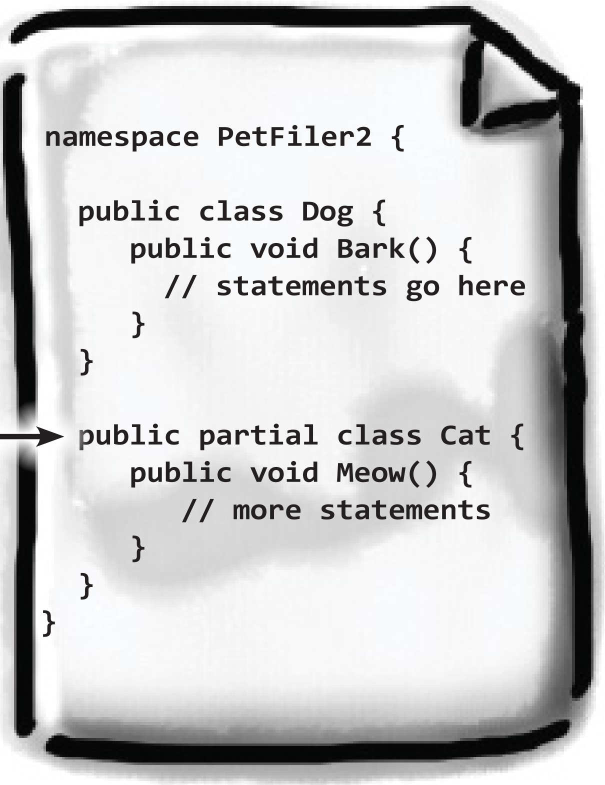
Q: I understand what *Program.cs* does—that's where the code for my program lives. But does my program need the other two files and folders?

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

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***.

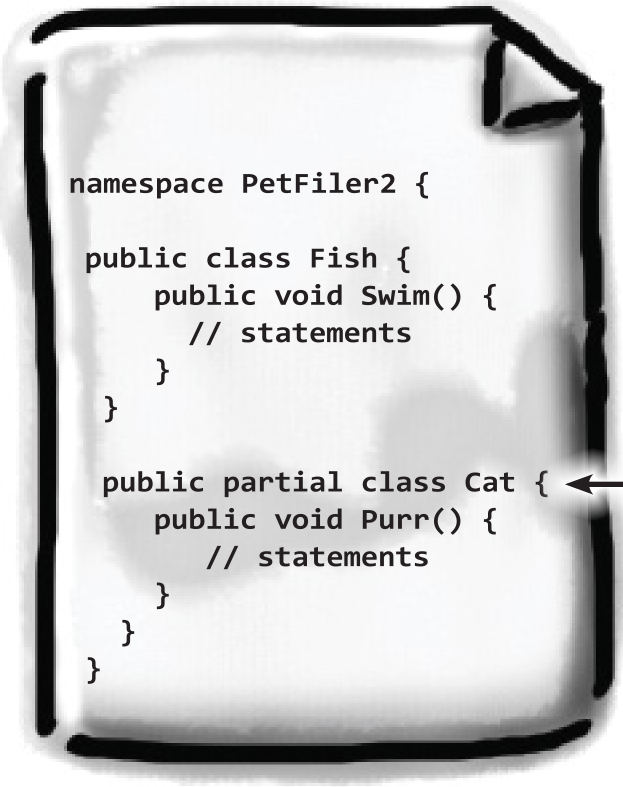
SomeClasses.cs



```
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.

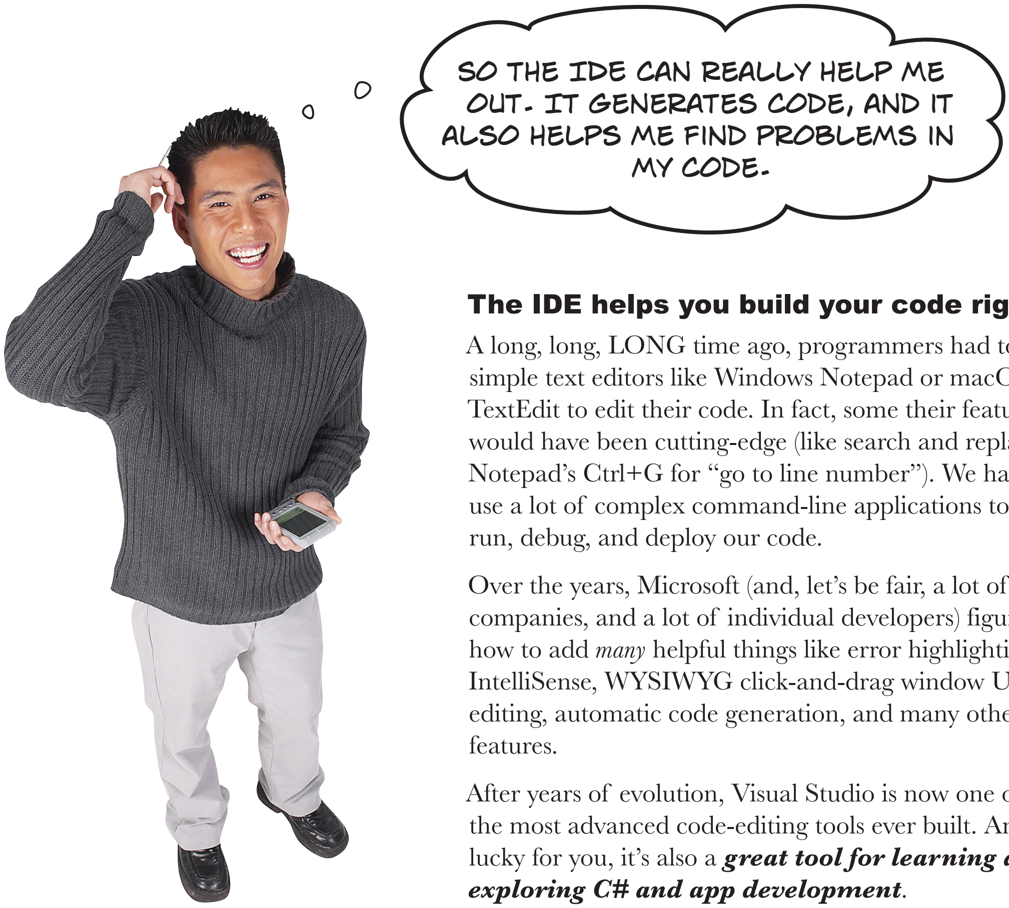
MoreClasses.cs



```
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.


SO THE IDE CAN REALLY HELP ME OUT. IT GENERATES CODE, AND IT ALSO HELPS ME FIND PROBLEMS IN MY CODE.

The IDE helps you build your code right.

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

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

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

there are no Dumb Questions

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

A: "Hello World" is a program that does one thing: it outputs the phrase "Hello World" to show that you can actually get something working. It's often the first program you write in a new language—and for a lot of us, the first piece of code we write in any 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—some folks say "mustaches," but we won't be using that term) 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 with 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.

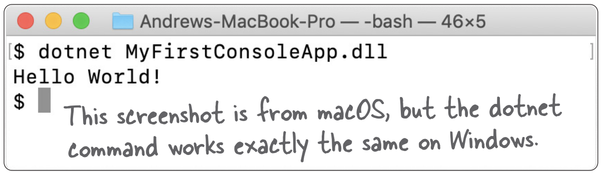
So when you run your program, how does it know which statement to start with? That's where the entry point comes in. Your code won't build unless there is **exactly one method called** Main. It's called the entry point because the program starts running—we say that it *enters* the code—with the first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

Q: So my .NET Core console apps really run on other operating systems?

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

You can try this out right now. You'll need .NET Core. The Visual Studio installer ***automatically installs .NET Core***, but you can also download it here: https://dotnet.microsoft.com/download.

Once it's installed, find your project folder by right-clicking on the MyFirstConsoleApp project in the IDE and choosing *Open Folder in File Explorer* (Windows) or *Reveal in Finder* (macOS). Go to the subdirectory under bin/Debug/, and copy all of the files to the computer you want to run it on. Then you can run it—and this will work on ***any*** **Windows, Mac, or Linux box** with .NET Core installed:



```
$ dotnet MyFirstConsoleApp.dll
Hello World!
$
```

This screenshot is from macOS, but the dotnet command works exactly the same on Windows.

Q: I can usually run programs by double-clicking on them, but I can't double-click on that *.dll* file. Can I create a Windows executable or macOS app that I can run directly?

A: Yes. You can use `dotnet` to publish **executable binaries** for different platforms. Open Command Prompt or Terminal, go to the folder with either your *.sln* or *.csproj* file, and run this command to generate a Windows executable—and 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 **self-contained macOS app**:

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

or specify `linux-x64` to publish a Linux app. That parameter is called a **runtime identifier** (or RID)—you can find 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 in this chapter and throughout the book. For example:

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

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

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

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

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. 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.

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.

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`.

You need to assign values to variables before you use them

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

Do this!

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

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

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

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

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

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

A few useful types

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

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

Generate a new method to work with variables

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

Do this!

① **Add a method to your new MyFirstConsoleApp project.**

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

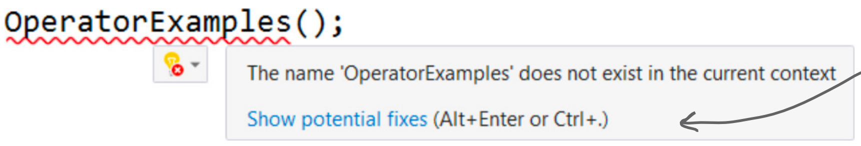
```
Console.WriteLine("Hello World!");
```

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:



```
OperatorExamples();
```

The name 'OperatorExamples' does not exist in the current context

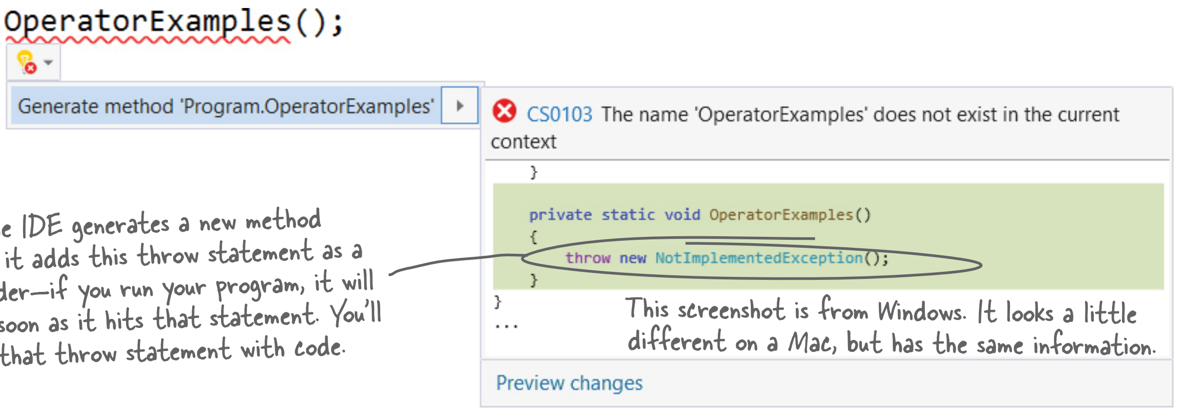
Show potential fixes (Alt+Enter or Ctrl+.)

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

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

③ **Generate the OperatorExamples method.**

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



```
OperatorExamples();
```

Generate method 'Program.OperatorExamples'

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

```
    }

    private static void OperatorExamples()
    {
        throw new NotImplementedException();
    }
}
...
```

Preview changes

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

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

The IDE has a solution: it will generate a method called OperatorExamples in your 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. 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.

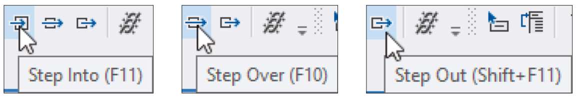
Line1: ______________________________

Line2: ______________________________

Line3: ______________________________

Use the debugger to watch your variables change

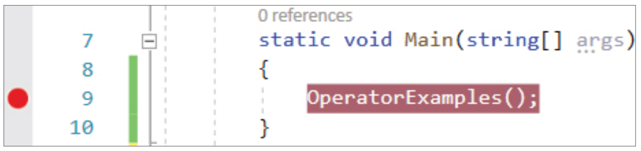
When you ran your program earlier, it was executing in the **debugger**—and that's an incredibly useful tool for understanding how your programs work. You can use **breakpoints** to pause your program when it hits certain statements and add **watches** to look at the value of your variables. Let's use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



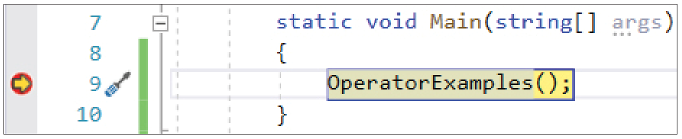
```
0 references
static void Main(string[] args)
{
    OperatorExamples();
}
```

The debugging shortcut keys for Mac are Step Over (⇧⌘O), Step In (⇧⌘I), and Step Out (⇧⌘U). The screens will look a little different, but the debugger operates exactly the same, as you saw in Chapter 1 in the *Mac Learner's Guide*.

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.

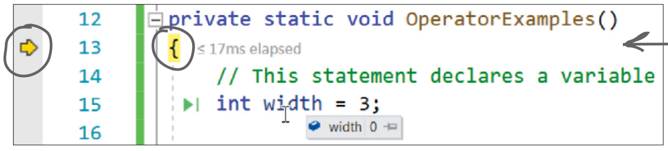


```
static void Main(string[] args)
{
    OperatorExamples();
}
```

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.



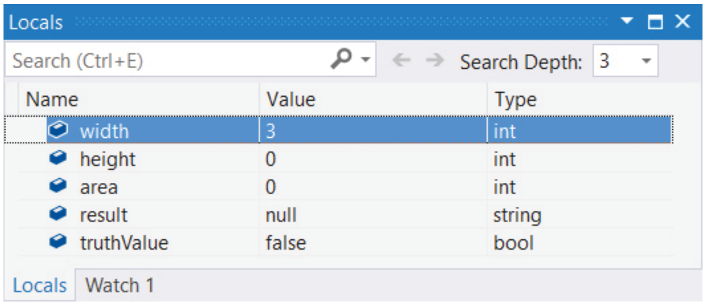
```
private static void OperatorExamples()
{  ≤ 17ms elapsed
    // This statement declares a variable
    int width = 3;
```

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 the values of your variables.**

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

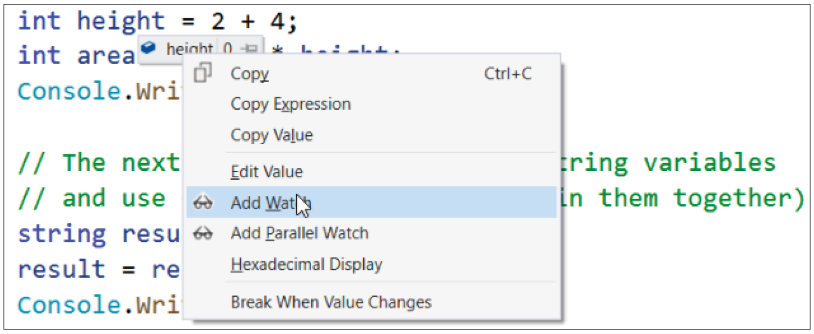


Name	Value	Type
width	3	int
height	0	int
area	0	int
result	null	string
truthValue	false	bool

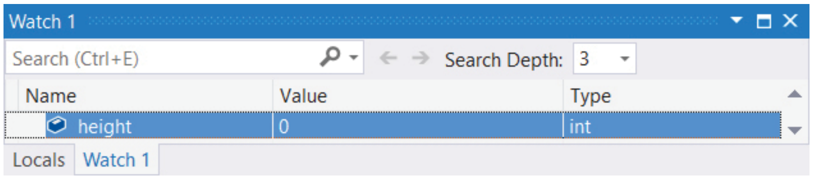
The Locals and Watch windows in Visual Studio for Mac look a little different than they do on Windows, but they contain the same information. You add watches the same way in both Windows and Mac versions of Visual Studio..

5 **Add a watch for the height variable.**

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


```
int height = 2 + 4;
int area
Console.Wri

// The next                    ring variables
// and use                     in them together)
string resu
result = re
Console.Wri
```

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



Name	Value	Type
height	0	int

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

6 **Step through the rest of the method.**

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

Use operators to work with variables

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

Equality Operators

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

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

Relational Operators

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

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

Logical Operators

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

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

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

Don't confuse the two equals sign operators!

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

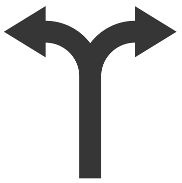
Use operators to compare two int variables

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

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

These are the ones you'll use most often.

"if" statements make decisions



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

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

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

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

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

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

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

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

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

Loops perform an action over and over

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

while loops keep looping statements while a condition is true

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

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

do/while loops run the statements then check the condition

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

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

for loops run a statement after each loop

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

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

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

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

The conditional test always runs at the beginning of each iteration, and the iterator always runs at the end of the iteration.

For Loops Up Close

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

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

}
```

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

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

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

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

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

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

Sharpen your pencil

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

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

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

```
// Loop #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 #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.

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

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

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

Sharpen your pencil Solution

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

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

Loop #1 executes once.

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

```
// Loop #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.

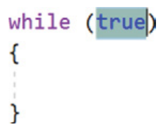
Take the time to <u>really figure out</u> how loop #3 works. Here's a perfect opportunity to try out the debugger on your own! Set a breakpoint on `q = p - q;` and <u>use the Locals window</u> to watch how the values of `p` and `q` change as you step through the loop.

Use code snippets to help write loops

Do this!

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

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



```
while (true)
{

}
```

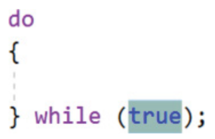
Type **`area < 50`**—the IDE will replace `true` with the text. **Press Enter** to finish the snippet. Then add two statements between the brackets:

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

IDE Tip: Brackets

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

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



```
do
{

} while (true);
```

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

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

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

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

Sharpen your pencil

Let's get some practice working with conditionals and loops. Update the Main method in your console app so it matches the new Main 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)
{
    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 declared with a *type* followed by a *name*.**
  ```
  int weight;
  // the variable's type is int and its name is weight
  ```
- **Most of the time, extra whitespace is fine.**
  So this: `int          j          =          1234          ;`
  Is exactly the same as this: `int j = 1234;`
- **If/else, while, do, and for are all about testing conditions.**
  Every loop we've seen so far keeps running as long as a condition is true.

THERE'S A *FLAW IN YOUR LOGIC!* WHAT HAPPENS 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`**. If it doesn't, then the loop will keep running until you kill the program or turn the computer off!

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

BRAIN POWER

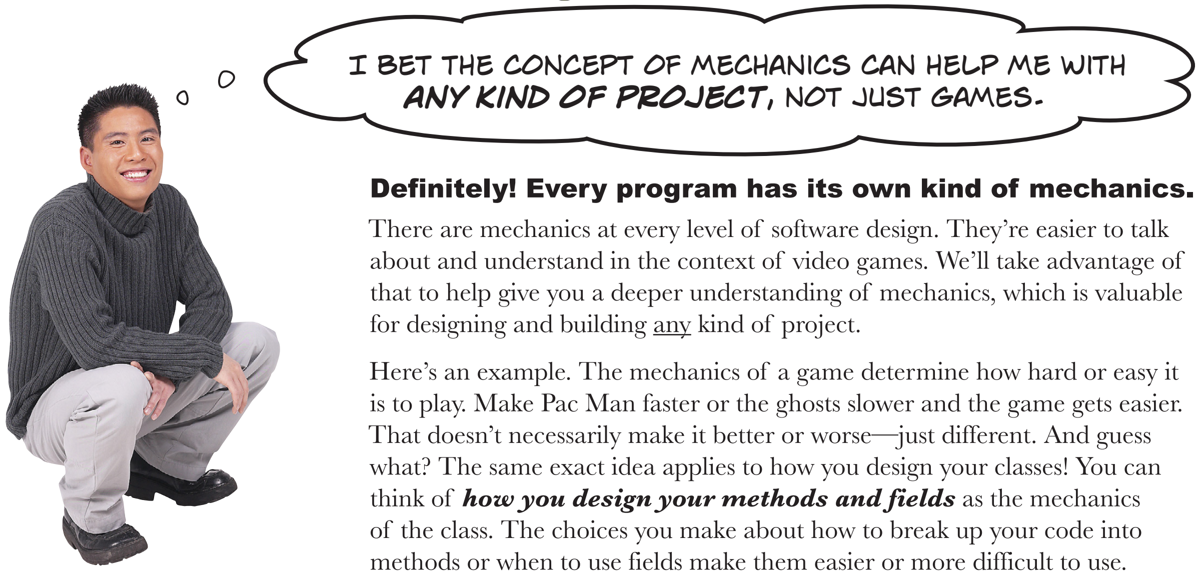
Can you think of a reason that you'd want to write a loop that never stops running?

Game design... and beyond

Mechanics

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

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


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

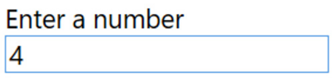
Definitely! Every program has its own kind of mechanics.

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

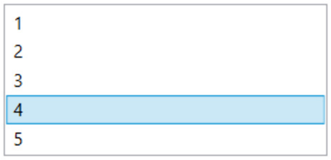
Here's an example. The mechanics of a game determine how hard or easy it is to play. Make Pac Man faster or the ghosts slower and the game gets easier. That doesn't necessarily make it better or worse—just different. And guess what? The same exact idea applies to how you design your classes! You can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

Controls drive the mechanics of your user interfaces

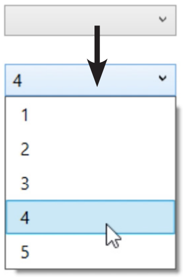
In the last chapter, you built a game using TextBlock and Grid **controls**. But there are a lot of different ways that you can use controls, and the choices you make about what controls to use can really change your app. Does that sound weird? It's actually really similar to the way we make choices 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). The same goes for apps: if you're designing an app where the user needs to enter a number, you can choose from different controls to let them do that—***and that choice affects how your user experiences the app***.



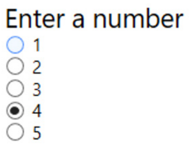
★ A **text box** lets a user enter any text they want. But we need a way to make sure they're only entering numbers and not just any text.



★ 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.



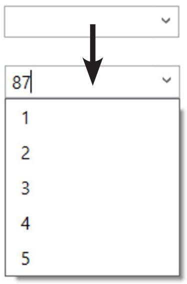
★ A **combo box** combines the behavior of a list box and a text box. It looks like a normal text box, but when the user clicks it a list box pops up underneath it.



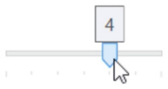
★ **Radio buttons** let you restrict the user's choice. You can use them for numbers if you want, and you can choose how you want to lay them out.

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.

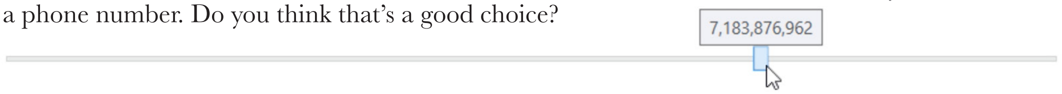
We can borrow the idea of mechanics from video games to understand our options, so we can make great choices for any of our own apps—not just games.



Editable combo 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?

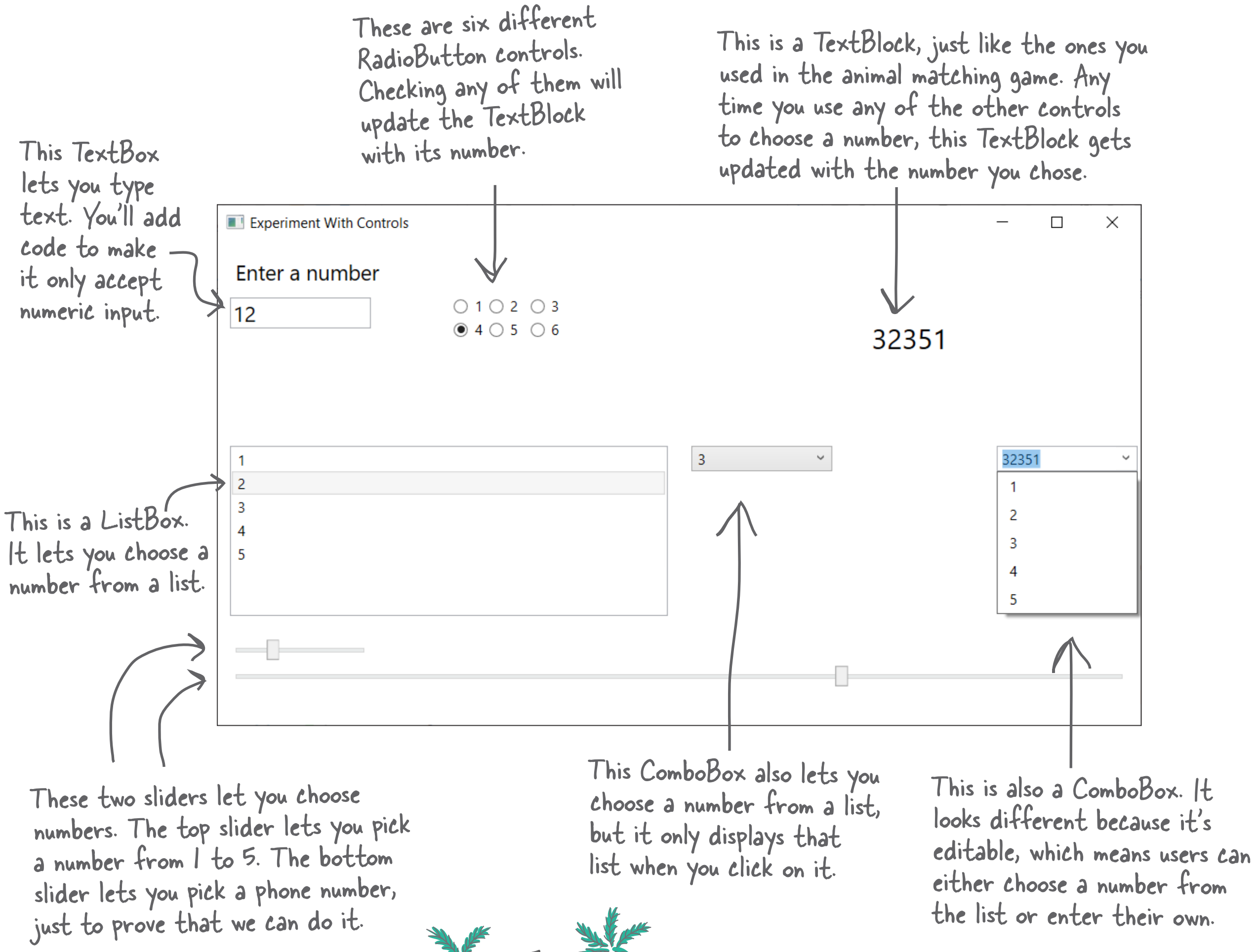


The rest of this chapter contains a project to build a WPF desktop app for Windows. Go to the Visual Studio for Mac Learner's Guide for the corresponding macOS project.

Create a WPF app to experiment with controls

Do this!

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 get 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.


These are six different RadioButton controls. Checking any of them will update the TextBlock with its number.
This is a TextBlock, just like the ones you used in the animal matching game. Any time you use any of the other controls to choose a number, this TextBlock gets updated with the number you chose.
This TextBox lets you type text. You'll add code to make it only accept numeric input.
Experiment With Controls
Enter a number
12
1 2 3
4 5 6
32351
1
2
3
4
5
3
32351
1
2
3
4
5
This is a ListBox. It lets you choose a number from a list.
These two sliders let you choose numbers. The top slider lets you pick a number from 1 to 5. The bottom slider lets you pick a phone number, just to prove that we can do it.
This ComboBox also lets you choose a number from a list, but it only displays that list when you click on it.
This is also a ComboBox. It looks different because it's editable, which means users can either choose a number from the list or enter their own.

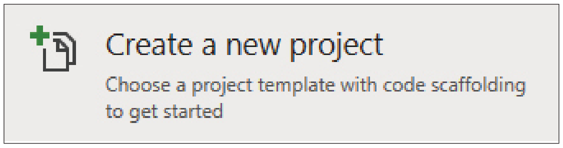
Don't worry about committing the XAML for controls to memory.

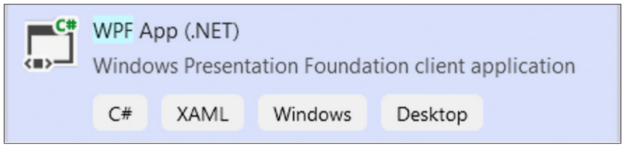
This Do this! and these exercises are all about getting some practice using XAML to build a UI with controls. You can always refer back to it when we use these controls in projects later in the book.

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 with your animal matching game in Chapter 1. Choose "Create a new project" and select WPF App (.NET).


Create a new project
Choose a project template with code scaffolding to get started



Name your project **ExperimentWithControls**.

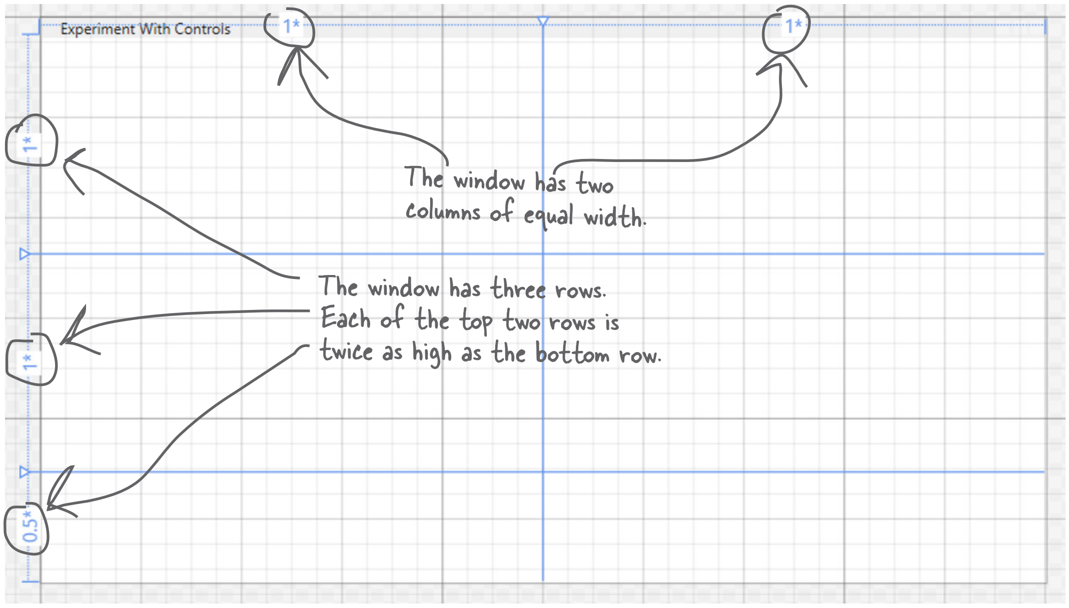
Set the window title

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 and two columns. The first two rows should each be twice the height of the third, and the two columns should be equal width.

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

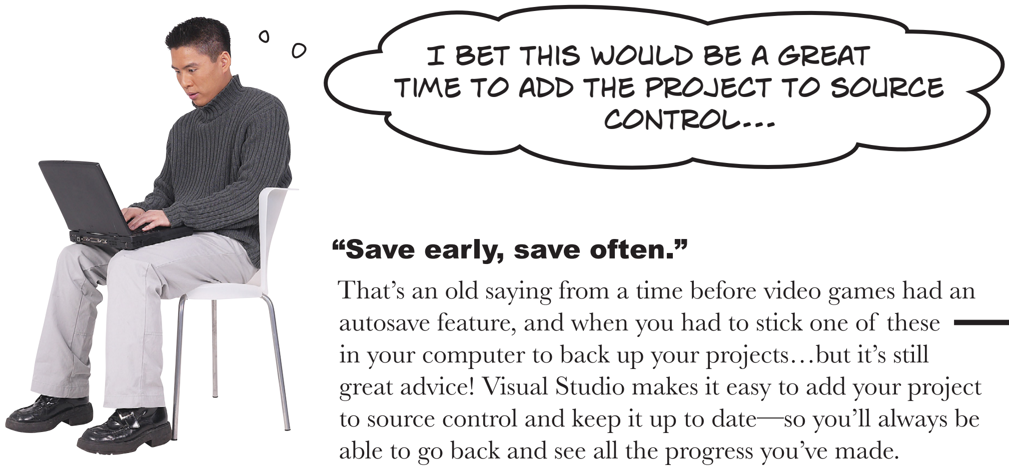

Experiment With Controls
1*
1*
1*
1*
0.5*
The window has two columns of equal width.
The window has three rows. Each of the top two rows is twice as high as the bottom row.

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.).

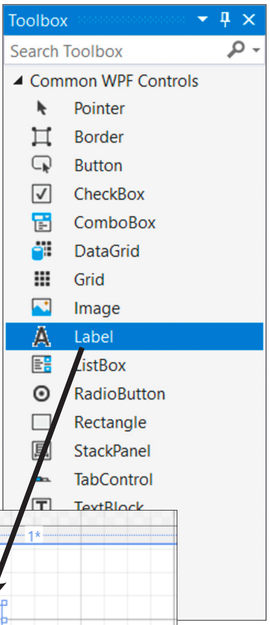

I BET THIS WOULD BE A GREAT TIME TO ADD THE PROJECT TO SOURCE CONTROL...

"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.

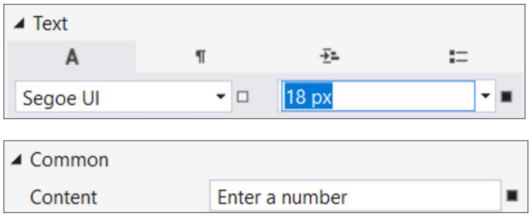
Add a TextBox control to your app

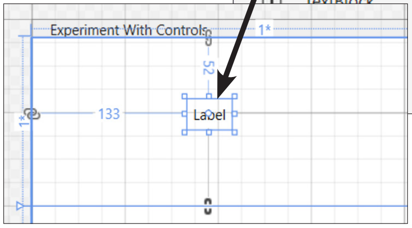
A **TextBox control** gives your user a box to type text into, so let's add one to your app. But we don't just want a TextBox sitting there without a label, so first we'll add a **Label control** (which is a lot like a TextBlock, except it's specifically used to add labels to other controls).



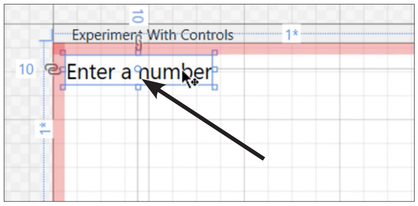
1 **Drag a Label out of the Toolbox into the top-left cell of the grid.**
This is exactly how you added TextBlock controls to your animal matching game in Chapter 1, except this time you're doing it with a Label control. It doesn't matter where in the cell you drag it, as long as it's in the upper-left cell.

2 **Set the text size and content of the Label.**
While the Label control is selected, go to the Properties window, 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`.





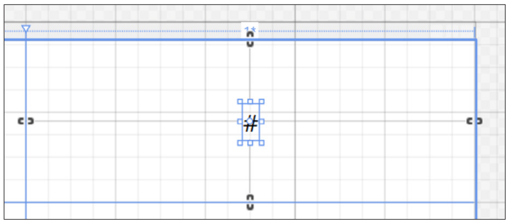
3 **Drag the Label to the upper-left corner of the cell.**
Click on the Label in the designer and drag it to the upper-left corner. When it's 10 pixels away from the left or top cell wall, you'll see gray bars appear and it will snap to a 10px margin.



The XAML for your window should now contain a 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**, and **center** it in the **upper-right cell** of the grid.

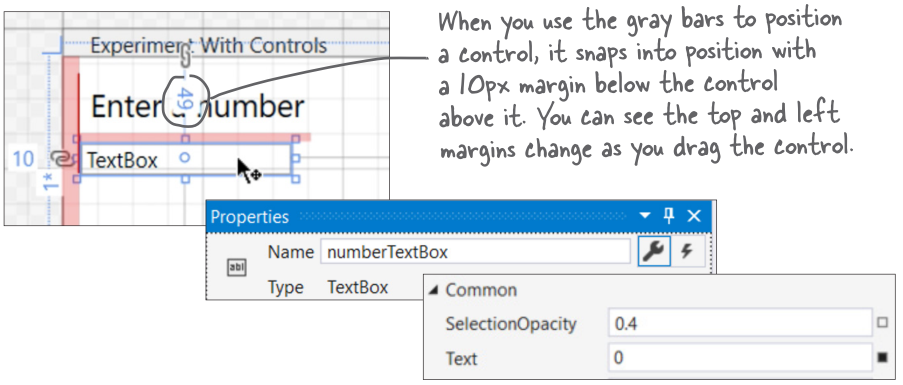


Here's the XAML for the TextBlock that goes in the upper right cell of the grid. You can use the visual designer or type in the XAML by hand. Just make sure your TextBlock has exactly the same properties as this solution—but like earlier, it's OK if your properties are in a different order.

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

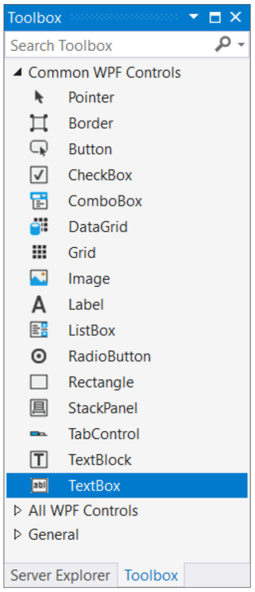
4 Drag a TextBox into the top-left cell of the grid.

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 below the Label—the same gray bars will appear to position it just underneath the Label with a 10px left margin. Set its name to **numberTextBox**, font size to **18px**, and text to **0**.

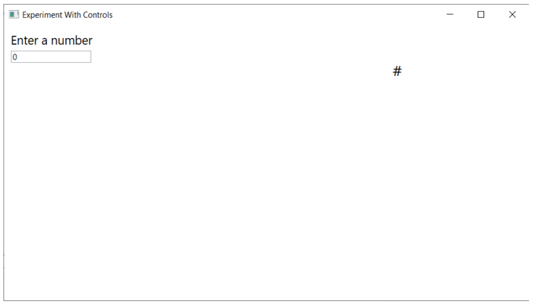


When you use the gray bars to position a control, it snaps into position with a 10px margin below the control above it. You can see the top and left margins change as you drag the control.

Properties
Name numberTextBox
Type TextBox
Common
SelectionOpacity 0.4
Text 0



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 OK 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 matching 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 that you added to the upper-right cell in the mini-exercise.

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 might add other event handlers—and in some cases (like with TextBlock) doesn't add any event handlers at all.

1 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)
{

}
```

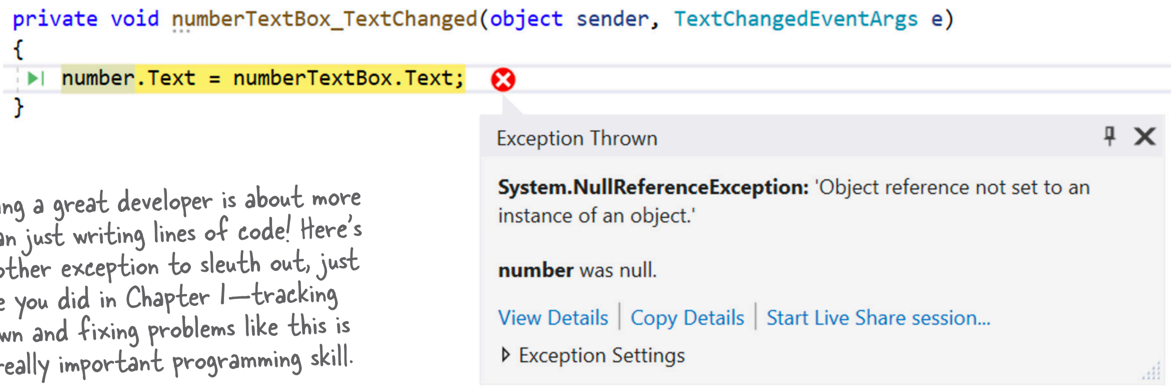
2 Add code to the new TextChanged event handler.

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.

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



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

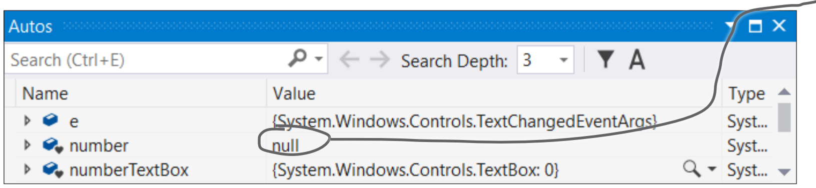
Exception Thrown

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

number was null.

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.

Take a look at the bottom of the IDE. It has an Autos window that shows you any defined variables. If you don't see, choose *Debug* >> *Windows* >> *Autos* from the menu.



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

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

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 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***.

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" ... />
```

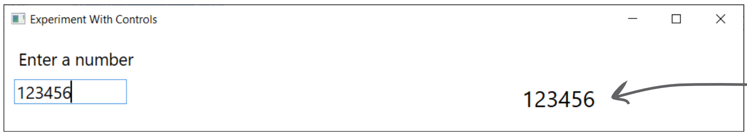
Select the TextBlock tag in the XAML editor move it above the TextBox so it gets initialized first.

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

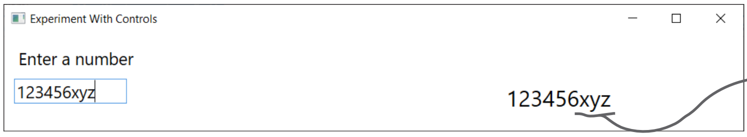
3 Run your app and try out the TextBox.

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 matching 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!

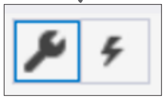


There has to be a way to allow the user to enter only numbers! How do you think we'll do that?

Add an event handler that only allows number input

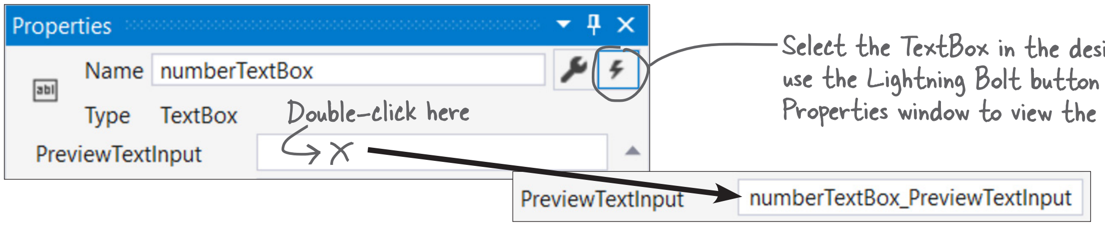
When you added the MouseDown event to your TextBlock in Chapter 1, you used the buttons in the upper-right corner of the Properties window to switch between properties and events. Now you'll do the same thing, except this time you'll use the **PreviewTextInput event** to only accept input that's made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

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.



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. Scroll down and **double-click inside the box next to PreviewTextInput** to make the IDE generate an event handler method.

Do this!



Select the TextBox in the designer, then use the Lightning Bolt button in the Properties window to view the events.

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:

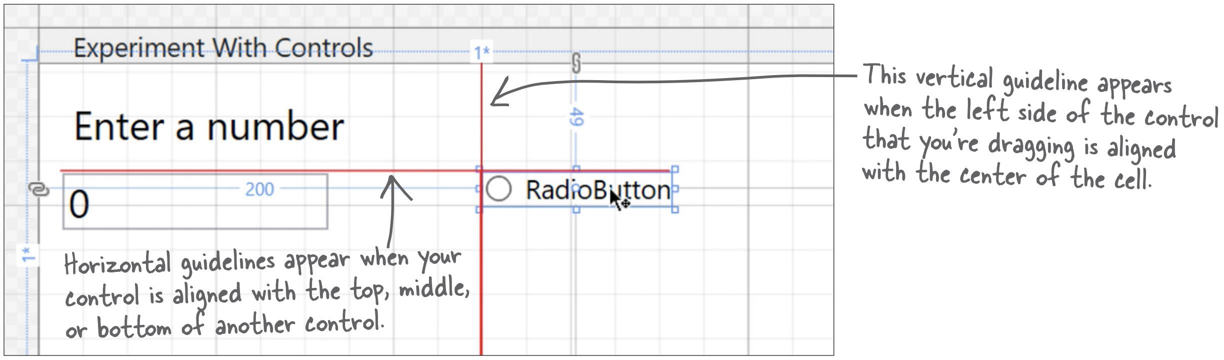
```
<TextBox x:Name="numberTextBox" FontSize="18" Margin="10,49,0,0" Text="0" Width="120"
         HorizontalAlignment="Left" TextWrapping="Wrap" VerticalAlignment="Top"
         TextChanged="numberTextBox_TextChanged"
         PreviewTextInput="numberTextBox_PreviewTextInput" />
```

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.

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.

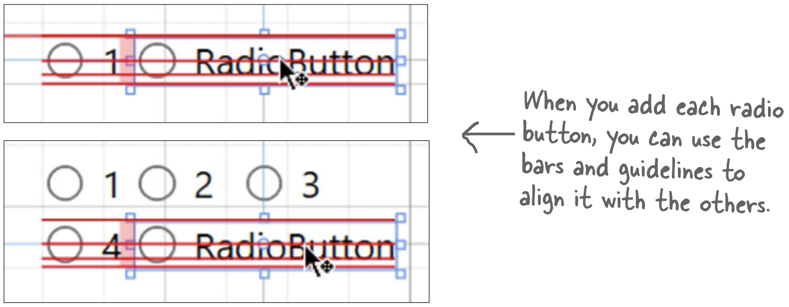
Add radio buttons to the upper-left cell next to the TextBox

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.


Experiment With Controls
Enter a number
0
200
RadioButton
49
1*
This vertical guideline appears when the left side of the control that you're dragging is aligned with the center of the cell.
Horizontal guidelines appear when your control is aligned with the top, middle, or bottom of another control.

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

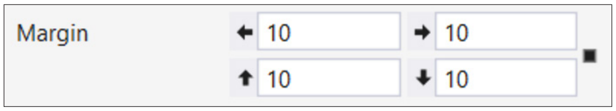
Next, add five more RadioButton controls, align them, and set their Content properties. But this time, <u>don't drag them out of the Toolbox</u>. Instead, **click on RadioButton in the Toolbox, then click inside the cell.** *(The reason you're doing that is if you have a RadioButton selected and then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


1 RadioButton
1 2 3
4 RadioButton
When you add each radio button, you can use the bars and guidelines to align it with the others.

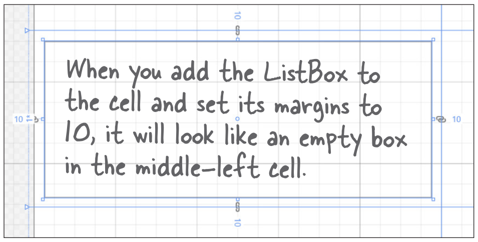
Still seeing event handlers and not properties in the Properties window? Use the Wrench button to display properties again—and if you used the search box, make sure you clear it.

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. In the Layout section, set all of its margins to 10.

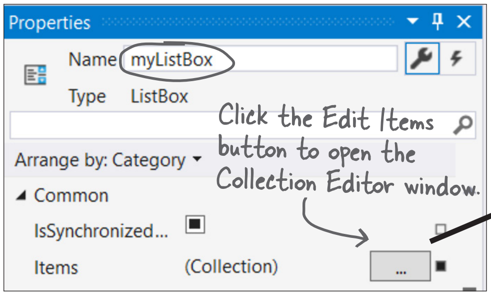


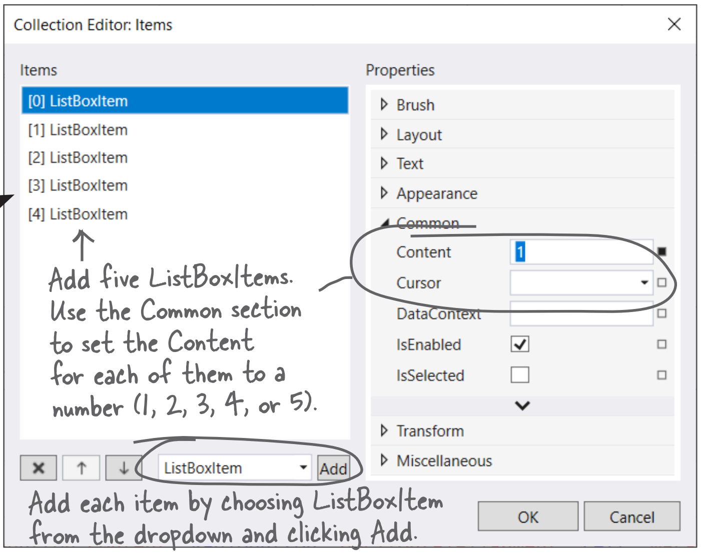
Margin	← 10	→ 10
	↑ 10	↓ 10


When you add the ListBox to the cell and set its margins to 10, it will look like an empty box in the middle-left cell.

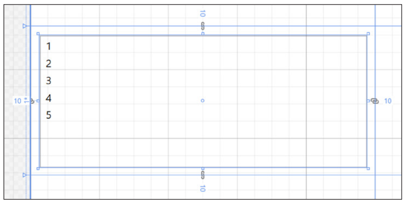
Name your ListBox myListBox and add ListBoxItems to it

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.



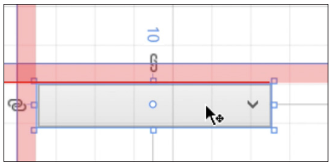


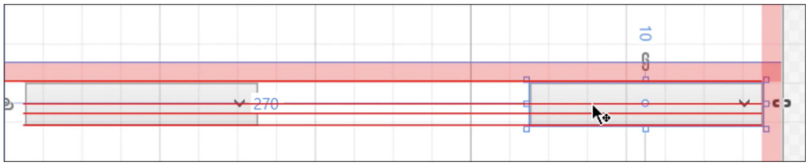
Your ListBox should now look like this:



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

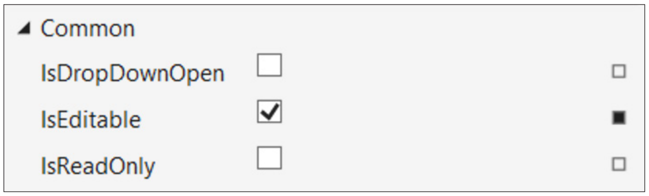
Click on ComboBox in the Toolbox, then click inside the middle-right cell to **add a ComboBox and name it readOnlyComboBox**. Drag it to the upper-left corner and use the gray bars to give it left and top margins of 10. Then **add another ComboBox named editableComboBox** to the same cell and align it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

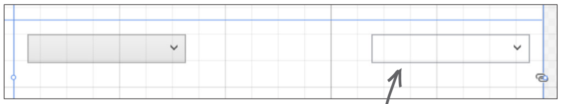




Use the Collection Editor window to **add the same ListBoxItems** with numbers 1, 2, 3, 4, and 5 to ***both*** ComboBoxes—so you'll need to do it for the first ComboBox, then the second ComboBox.

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 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 these lines just above the closing `</Grid>` tag. Just like with any other XAML you've seen so far, it's OK 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"/>
```

The IDE added the margin and alignment properties to each RadioButton control when you dragged it into place.

```
<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>
```

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.

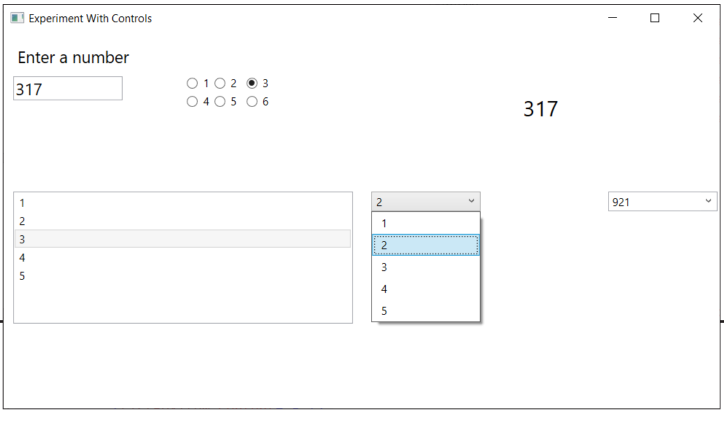
```
<ComboBox x:Name="readOnlyComboBox" Grid.Column="1" Margin="10,10,0,0" Grid.Row="1"
          HorizontalAlignment="Left" VerticalAlignment="Top" Width="120">
    <ListBoxItem Content="1"/>
    <ListBoxItem Content="2"/>
    <ListBoxItem Content="3"/>
    <ListBoxItem Content="4"/>
    <ListBoxItem Content="5"/>
</ComboBox>
```

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.

```
<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>
```

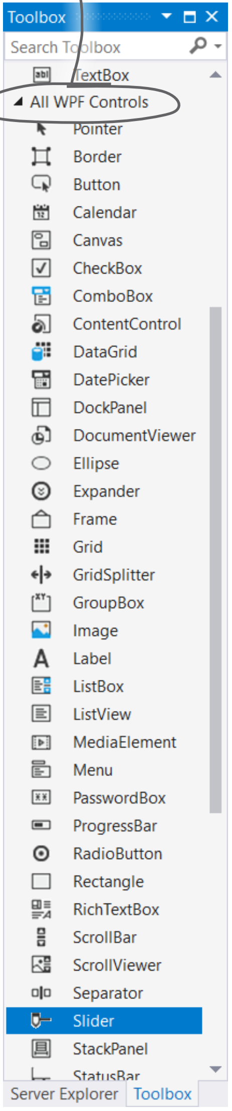
When you run your program, it should look like this. You can use all of the controls, but only the TextBox actually updates the value at the upper right.



Add sliders to the bottom row of the grid

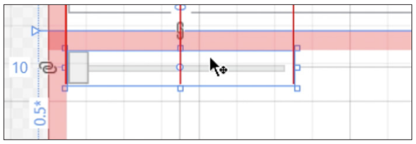
Let's add two sliders to the bottom row and then hook up their event handlers so they update the TextBlock in the upper-right corner.

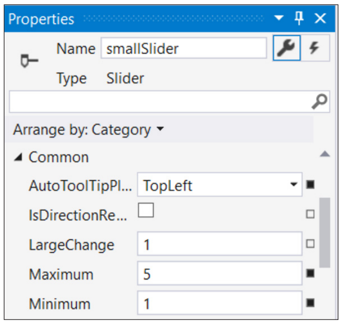
To find the Slider control in the Toolbox, you'll need to expand the "All WPF Controls" section and scroll almost all the way to the bottom.



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-left corner of the cell and use the gray bars to give it left and top margins of 10.





Properties

Name smallSlider

Type Slider

Arrange by: Category

Common

AutoToolTipPl...	TopLeft
IsDirectionRe...	☐
LargeChange	1
Maximum	5
Minimum	1

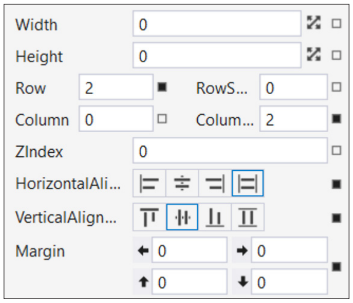
Use the Common section of the Properties window to set AutoToolTipPlacement to **`TopLeft`**, Maximum to **`5`**, and Minimum to **`1`**. Give it the name **`smallSlider`**. Then double-click on the slider to add this event handler:

```
private void smallSlider_ValueChanged(
        object sender, RoutedPropertyChangedEventArgs<double> e)
{
    number.Text = smallSlider.Value.ToString("0");
}
```

The value of the Slider control is a 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.



Width	0		
Height	0		
Row	2	RowS...	0
Column	0	Colum...	2
ZIndex	0		
HorizontalAli...			
VerticalAlign...			
Margin	← 0	→ 0	
	↑ 0	↓ 0	

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 to **`1111111111`**, Maximum to **`9999999999`**, and Value to **`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 zeros and hyphens cause the method to format any 10-digit number as a US phone number.

Add C# code to make the rest of the controls work

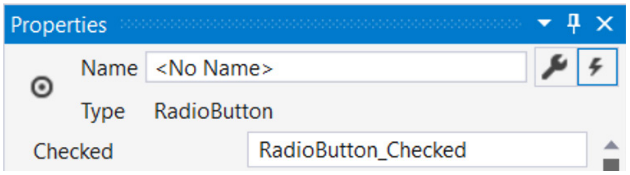
You want each of the controls in your app to do the same thing: update the TextBlock in the upper-right cell with a number, so when you check one of the radio buttons or pick an item from a ListBox or ComboBox, the TextBlock is updated with whatever value you chose.

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 the 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.



Name <No Name>
Type RadioButton
Checked RadioButton_Checked

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 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.

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:

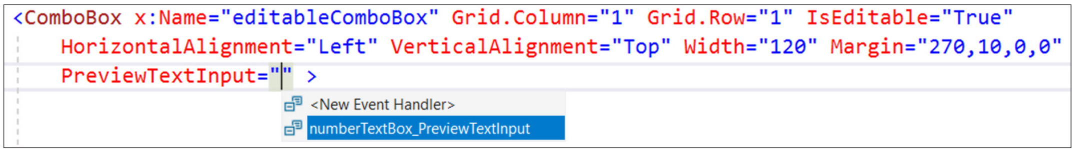
```
private void readOnlyComboBox_SelectionChanged(
        object sender, SelectionChangedEventArgs e)
{
    if (readOnlyComboBox.SelectedItem is ListBoxItem listBoxItem)
        number.Text = listBoxItem.Content.ToString();
}
```

You can also use the Properties window to add a SelectionChanged event. If you accidentally do this, you can hit "undo" (but make sure you do it in both files).

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. 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 can **reuse the same event handler** that you already added for the TextBox.

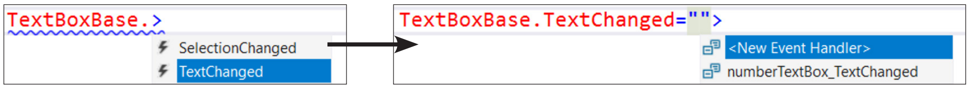
Go to the XAML for the editable ComboBox, put your cursor just before the closing caret **>** and **start typing *PreviewTextInput***. An IntelliSense window will pop up to help you complete the event name. Then **add an equals sign**—as soon as you do, the IDE will prompt you to either choose a new 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="" >
```

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 to add a new tag below `ComboBox`. 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.>
```

```
TextBoxBase.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!

THERE ARE SO MANY DIFFERENT WAYS FOR USERS TO CHOOSE NUMBERS! THAT GIVES ME LOTS OF OPTIONS WHEN I'M DESIGNING MY APPS.

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 time.
- The Visual Studio debugger lets you **pause your app** and inspect the values 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 #1

Explore C# with Unity

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

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

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

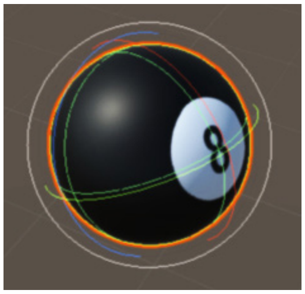
In this first lab, you'll get up and running with Unity. You'll get oriented with the Unity editor, and you'll start creating and manipulating 3D shapes.

Unity is a powerful tool for game design

Welcome to the world of Unity, a complete system for designing professional-quality games—both two-dimensional (2D) and three-dimensional (3D)—as well as simulations, tools, and projects. Unity includes many powerful things, including...

A cross-platform game engine

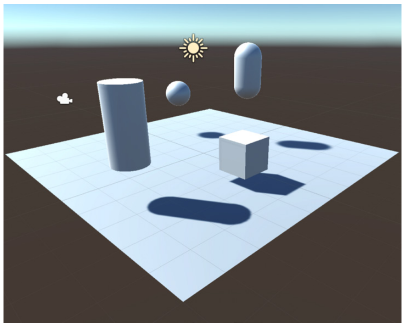
A **game engine** displays the graphics, keeps track of the 2D or 3D characters, detects when they hit each other, makes them act like real-world physical objects, and much, much more. Unity will do all of these things for the 3D games you build throughout this book.



A powerful 2D and 3D scene editor

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

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



An ecosystem for game creation

Beyond being an enormously powerful tool for creating games, Unity also features an ecosystem to help you build and learn. The Learn Unity page (https://unity.com/learn) has valuable self-guided learning resources, and the Unity forums (https://forum.unity.com) help you connect with other game designers and ask questions. The Unity Asset Store (https://assetstore.unity.com) provides free and paid assets like characters, shapes, and effects that you can use in your Unity projects.

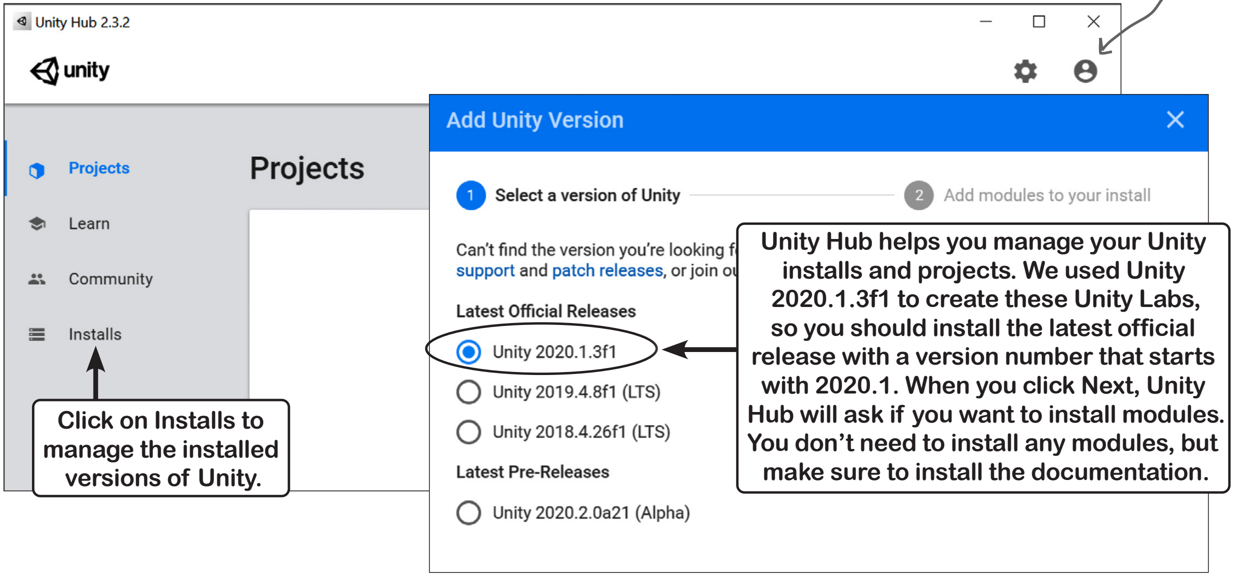
Our Unity Labs will focus on using Unity as a tool to explore C#, and practicing with the C# tools and ideas that you've learned throughout the book.

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

Download Unity Hub

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

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



Click on Installs to manage the installed versions of Unity.

Unity Hub helps you manage your Unity installs and projects. We used Unity 2020.1.3f1 to create these Unity Labs, so you should install the latest official release with a version number that starts with 2020.1. When you click Next, Unity Hub will ask if you want to install modules. You don't need to install any modules, but make sure to install the documentation.

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 2020.1***—that's the same version we used to take the screenshots in these labs. Once it's installed, make sure that it's set as the preferred version.

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

You can learn more about installing Unity Hub on Windows, macOS, and Linux here: https://docs.unity3d.com/2020.1/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.

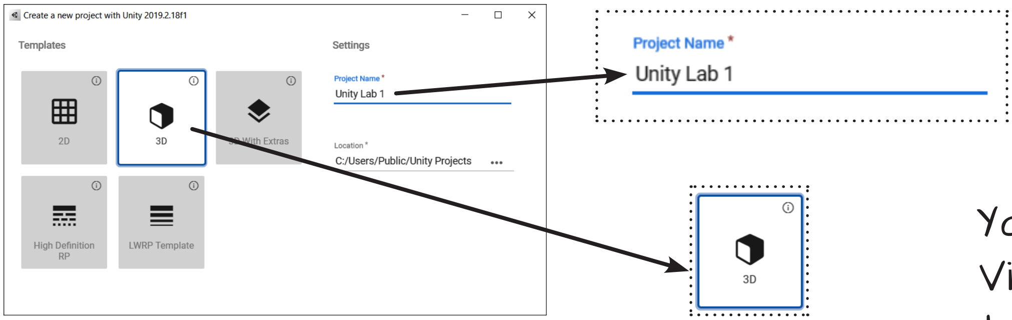
Watch it!

Unity Hub may look a little different.

The screenshots in this book were taken with Unity 2020.1 (Personal Edition) and Unity Hub 2.3.2. 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 what you see won't quite match what's shown 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).


Create a new project with Unity 2019.2.18f1

Project Name * Unity Lab 1

Location * C:/Users/Public/Unity Projects

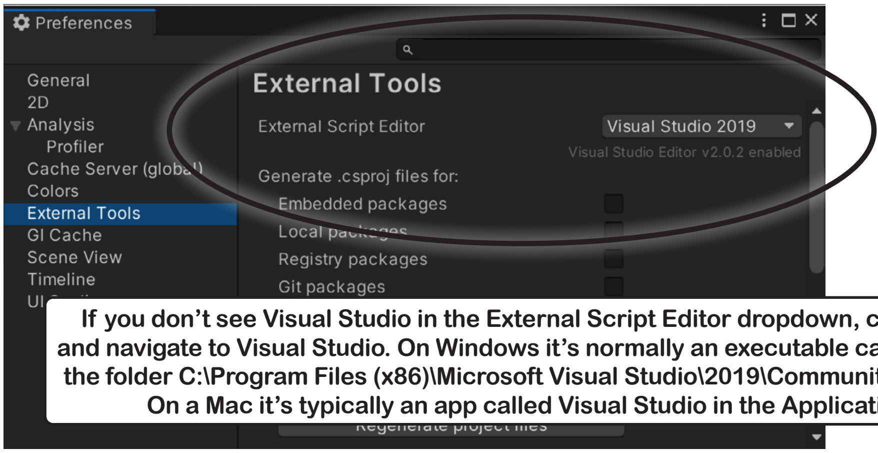
Click Create Project to create the new folder with the Unity project. When you create a new project, Unity generates a lot of files (just like Visual Studio does when it creates new projects for you). It could take Unity a minute or two to create all of the files for your new project.

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

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.

In some older versions of Unity, you may see an ***Editor Attaching*** *checkbox—if so, make sure that it's checked (that will let you debug your Unity code in the IDE).*


External Tools

External Script Editor: Visual Studio 2019

Visual Studio Editor v2.0.2 enabled

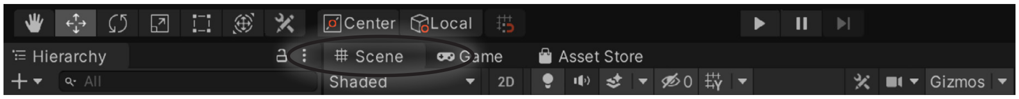
If you don't see Visual Studio in the External Script Editor dropdown, choose Browse... and navigate to Visual Studio. On Windows it's normally an executable called *devenv.exe* in the folder C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\Common7\IDE\. On a Mac it's typically an app called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

OK! 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. Like in 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:

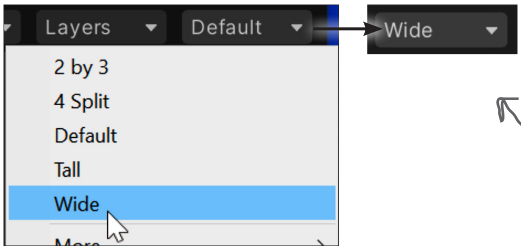


Try docking it inside or next to other panels, then drag it to the middle of the editor to make it a floating window.

Choose the Wide layout to match our screenshots

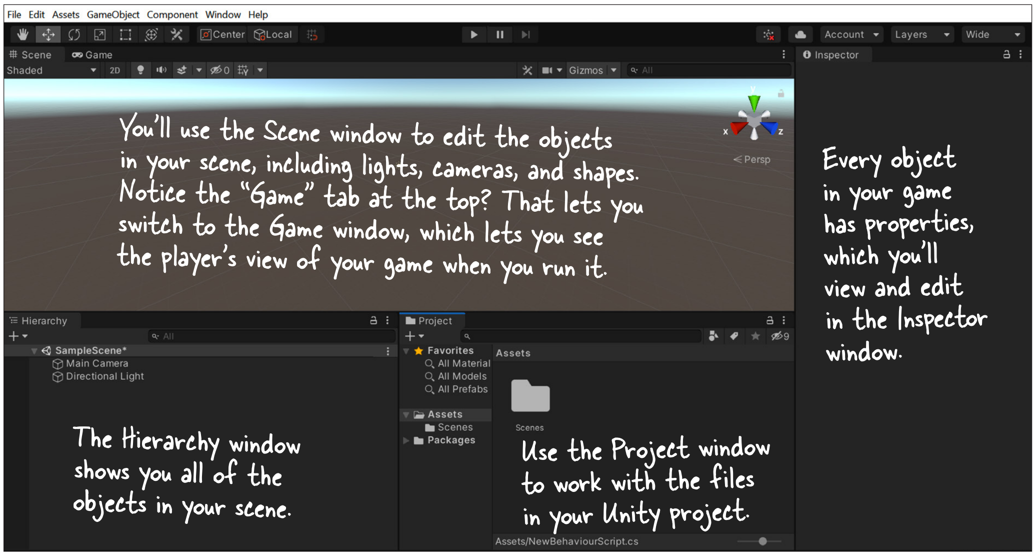
The **Scene view** is your main interactive view of the world that you're creating. You use it to position 3D shapes, cameras, lights, and all of the other objects in your game.

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.

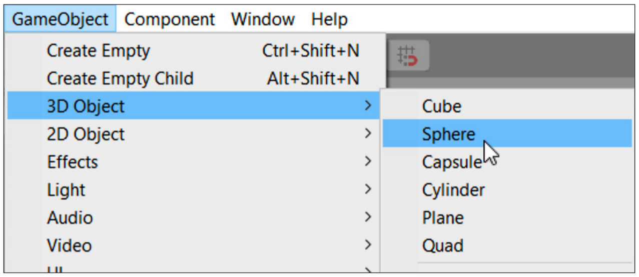
Here's what your Unity editor should look like in the Wide layout:


File Edit Assets GameObject Component Window Help
Center
Local
Account
Layers
Wide
Scene
Game
Inspector
Shaded
2D
Gizmos
Persp
Hierarchy
SampleScene*
Main Camera
Directional Light
Project
Favorites
All Material
All Models
All Prefabs
Assets
Scenes
Packages
Assets
Scenes
Assets/NewBehaviourScript.cs
You'll use the Scene window to edit the objects in your scene, including lights, cameras, and shapes. Notice the "Game" tab at the top? That lets you switch to the Game window, which lets you see the player's view of your game when you run it.
Every object in your game has properties, which you'll view and edit in the Inspector window.
The Hierarchy window shows you all of the objects in your scene.
Use the Project window to work with the files in your Unity project.

Your scene is a 3D environment

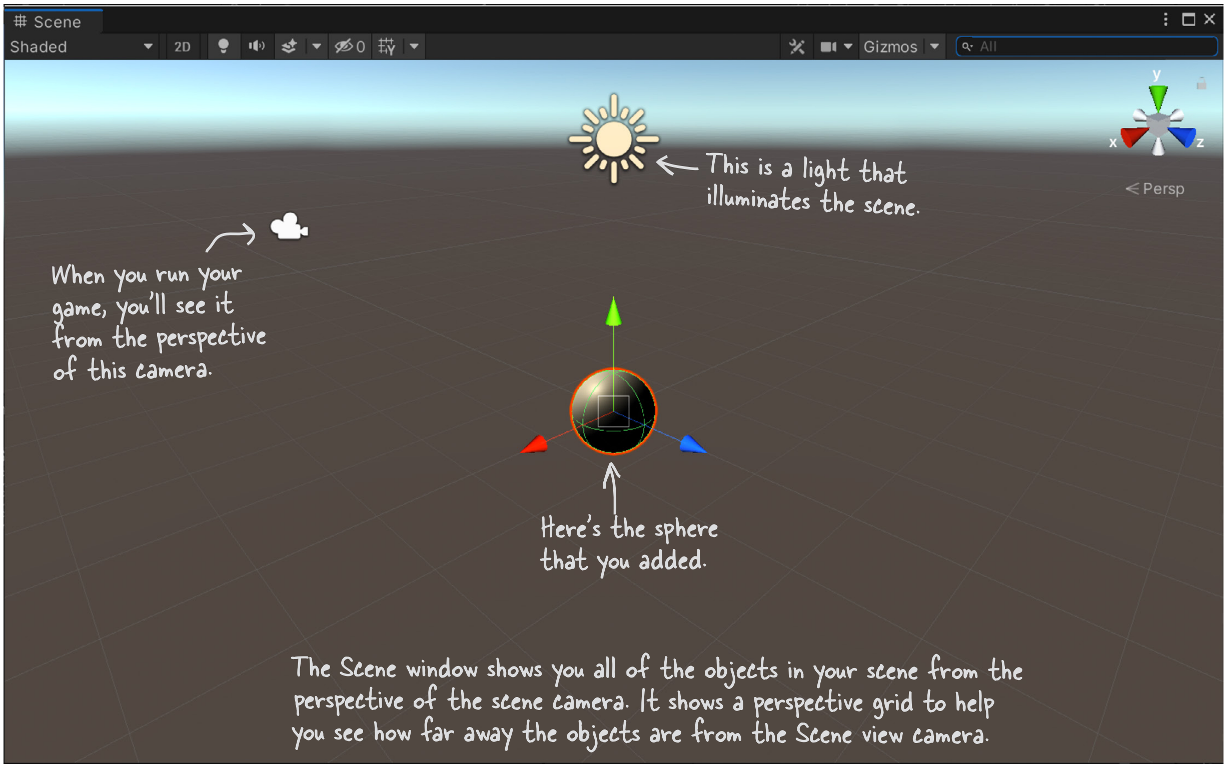
As soon as you start the editor, you're editing a **scene**. You can think of scenes as levels in your Unity games. Every game in Unity is made up of one or more scenes. Each scene contains a separate 3D environment, with its own set of lights, shapes, and other 3D objects. When you created your project,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

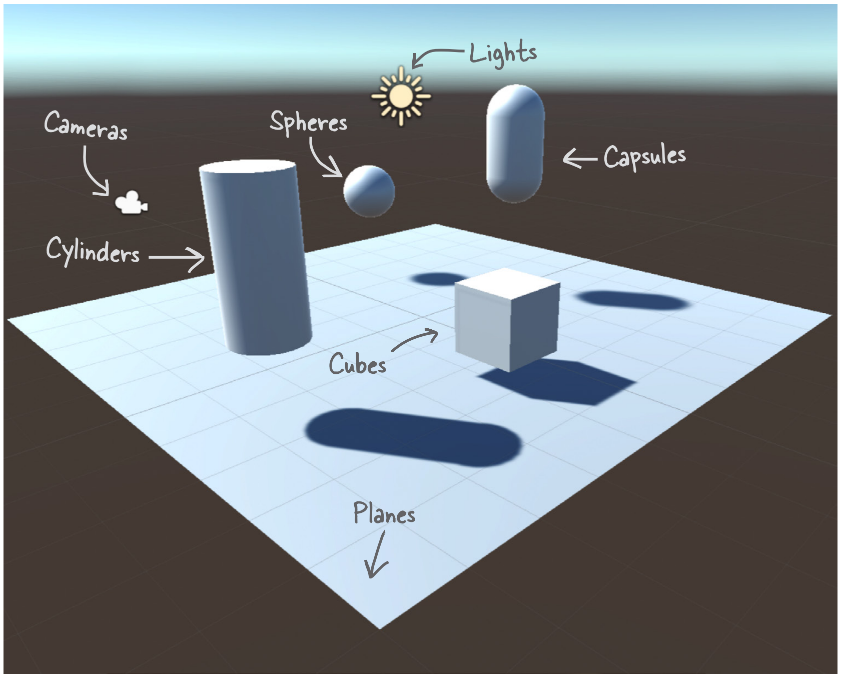

Scene
Shaded
2D
0
Gizmos
All
y
x
z
< Persp
This is a light that illuminates the scene.
When you run your game, you'll see it from the perspective of this camera.
Here's the sphere that you added.

The Scene window shows you all of the objects in your scene from the perspective of the scene camera. It shows a perspective grid to help you see how far away the objects are from the Scene view camera.

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:



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.

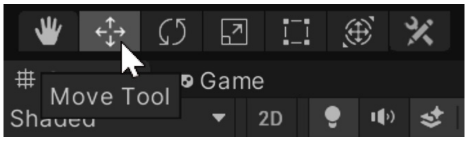
Each GameObject contains a number of **components** that provide its shape, set its position, and give it all of its behavior. For example:

- *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.*

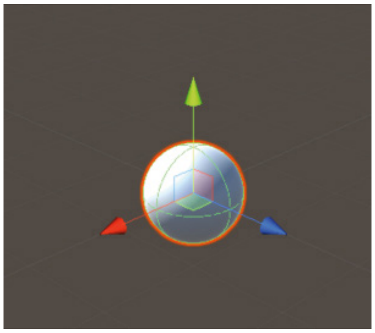
Use the Move Gizmo to move your GameObjects

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.



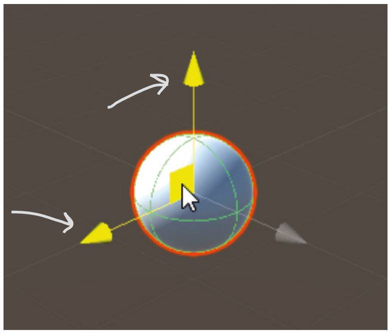
The buttons on the left side of the toolbar let you choose Transform Tools like the Move tool, which displays the Move Gizmo as arrows and a cube on top of the GameObject that's currently selected.

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.



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. 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.

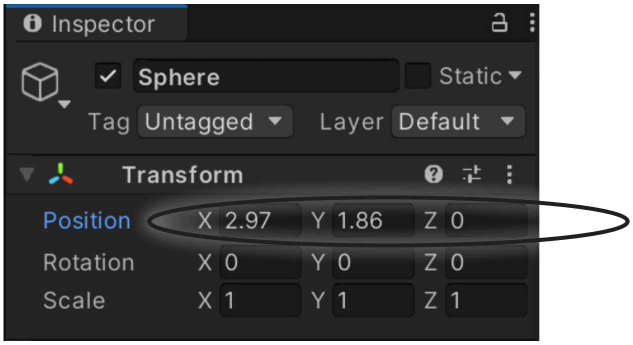
The Inspector shows your GameObject's components

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. When you reset the layout to Wide, the Hierarchy window is in the lower-left corner of the Unity editor.



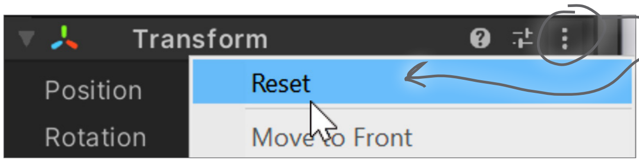
Position	X 2.97	Y 1.86	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

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

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

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

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



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

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

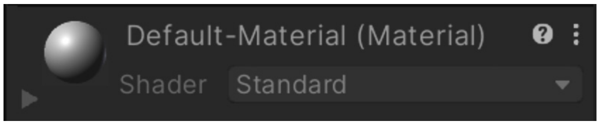
You can learn more about the tools and how to use them to position GameObjects in the Unity Manual. Click Help >> Unity Manual and search for the "Positioning GameObjects" page.

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

Add a material to your Sphere GameObject

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

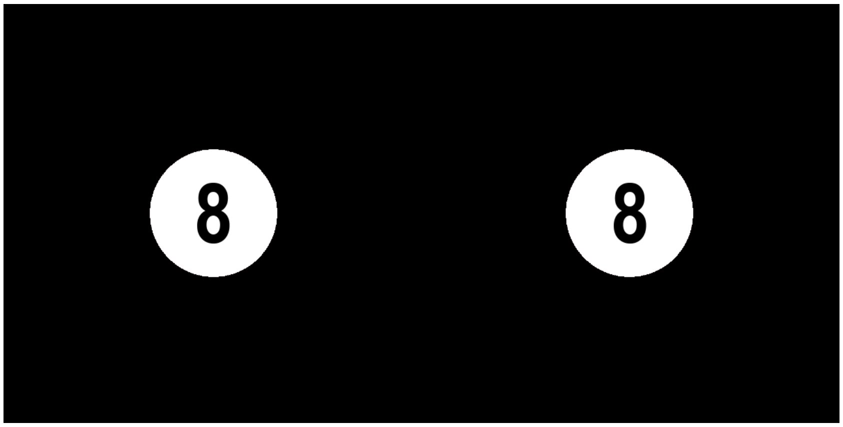
1. **Select the sphere.**
When the sphere is selected, you can see its material as a component in the Inspector window:



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

2. **Go to our Billiard Ball Textures page on GitHub.**
Go to https://github.com/head-first-csharp/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. It's an ordinary 1200 × 600 PNG image file that you can open in your favorite image viewer.

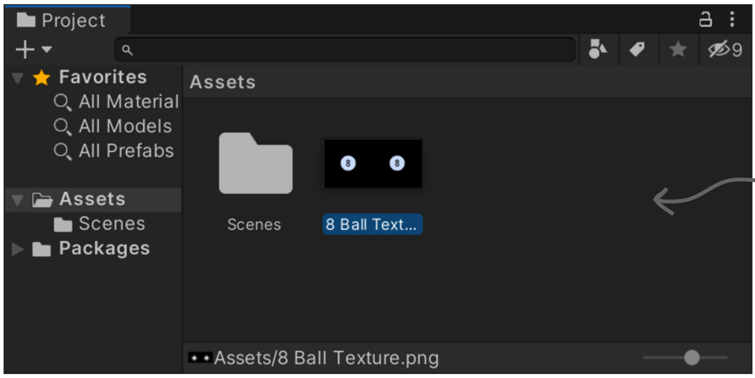


We designed this image file so that it looks like an 8 ball when Unity "wraps" it around a sphere.

Download the file into a folder on your computer.

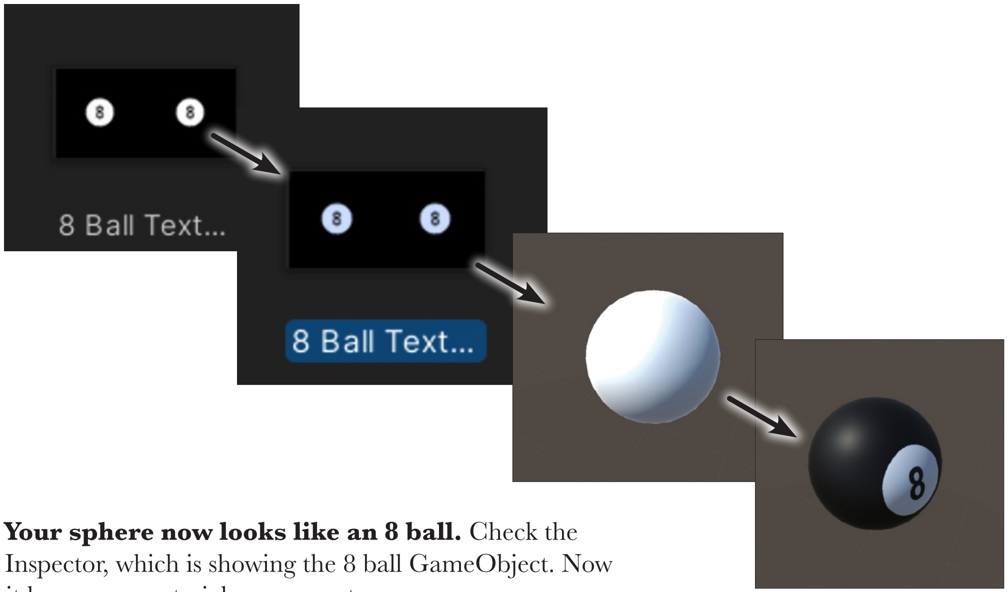
(You might need to right-click on the Download button to save the file, or click Download to open it and then save it, depending on your browser.)

④ **Import the 8 Ball Texture image into your Unity project.**
Right-click on the Assets folder in the Project window, choose **Import New Asset...** and import the texture file. You should now see it when you click on the Assets folder in the Project window.

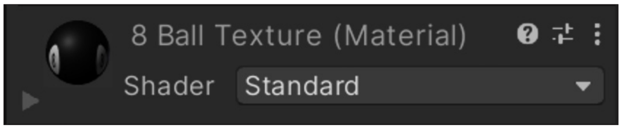


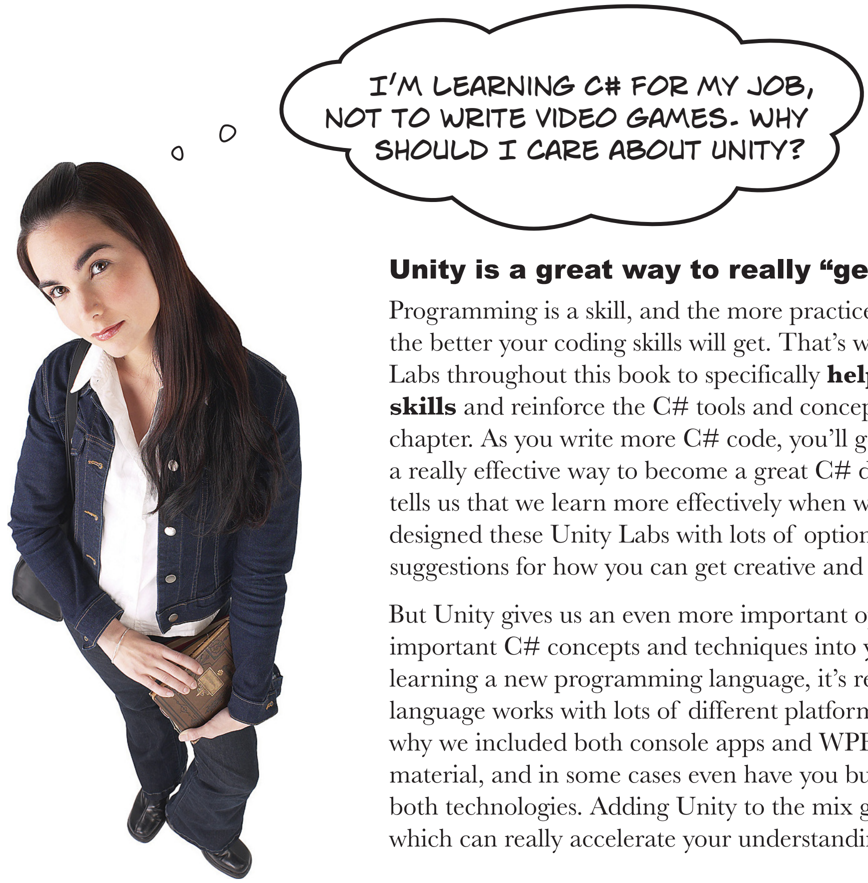
You right-clicked inside the Assets folder in the Project window to import the new asset, so Unity imported the texture into that folder.

⑤ **Add the texture to your sphere.**
Now you just need to take that texture and "wrap" it around your sphere. Click on 8 Ball Texture in the Project window to select it. Once it's selected, **drag it onto your sphere**.



Your sphere now looks like an 8 ball. Check the Inspector, which is showing the 8 ball GameObject. Now it has a new material component:




I'M LEARNING C# FOR MY JOB, NOT TO WRITE VIDEO GAMES. WHY SHOULD I CARE ABOUT UNITY?

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

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

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

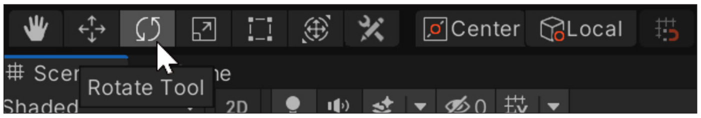
The **GitHub for Unity extension** (https://unity.github.com) lets you save your Unity projects in GitHub. Here's how:

- **To install GitHub for Unity:** Go to https://assetstore.unity.com and add GitHub for Unity to your assets. Go back to Unity, **choose Package Manager** from the Window menu, select "GitHub for Unity" from "My Assets," and import it. You'll need to import GitHub into each new Unity project.
- **To push your changes to a GitHub repo:** Choose GitHub from the Window menu. Each Unity project is stored in a separate repository in your GitHub account, so **click the Initialize button** to initialize a new *local* repo (you'll be prompted to log into GitHub), then **click the Publish button** to create a new repo in your GitHub account for your project. Any time you want to push your changes to GitHub, **go to the Changes tab** in the GitHub window, **click All**, enter a **commit summary** (any text will do), and **click Commit at** the bottom of the GitHub window. Then click **Push (1)** at the top of the GitHub window to push your changes back to GitHub.

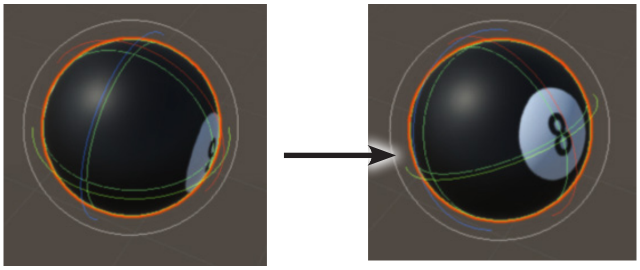
You can also back up and share your Unity projects with **Unity Collaborate**, which lets you publish your projects to their cloud storage. Your Unity Personal account comes with 1 GB of cloud storage for free, which is enough for all of the Unity Lab projects in this book. Unity will even keep track of your project history (which doesn't count 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.



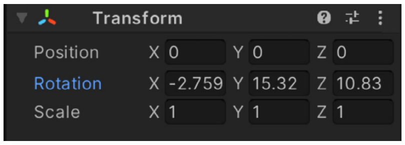
1 **Click on the sphere.** Unity will display a wireframe sphere Rotate Gizmo with red, blue, and green circles. Click the red circle and drag it to rotate the sphere around the X axis.



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

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

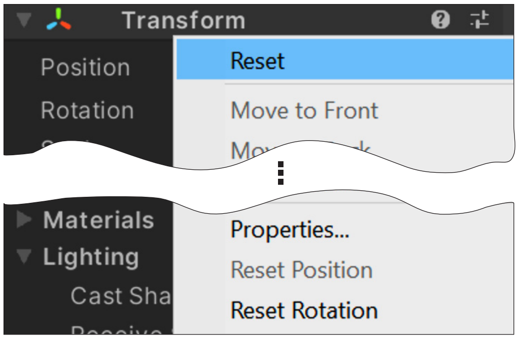
2 **Click and drag the green and blue circles to rotate around the Y and Z axes.** The outer white circle rotates the sphere along the axis coming out of the Scene view camera. Watch the Rotation numbers change in the Inspector window.



Transform

Position	X 0	Y 0	Z 0
Rotation	X -2.759	Y 15.32	Z 10.83
Scale	X 1	Y 1	Z 1

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



Click the three dots (or right-click anywhere in the header of the Transform panel) to bring up the context menu. The Reset option at the top of the menu resets the component to its default values.

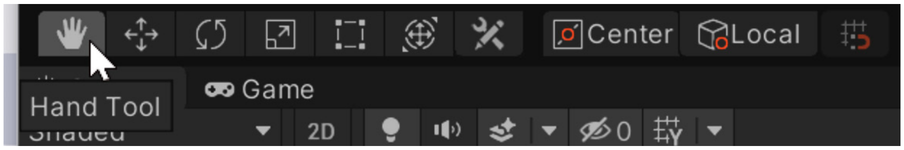
Use these options from further down in the context menu to reset the position and rotation of a GameObject.

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

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

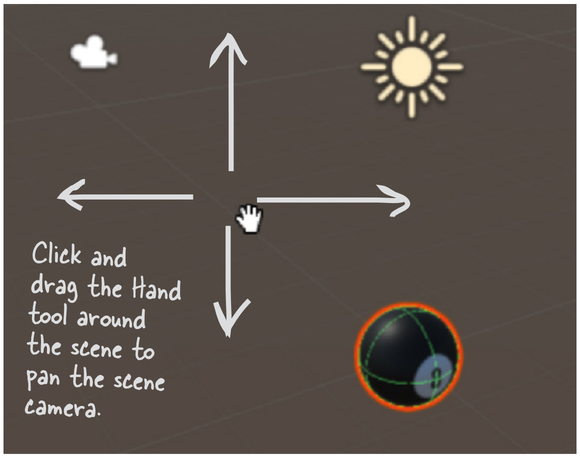
Use the mouse scroll wheel or scroll feature on your trackpad to zoom in and out, and toggle between the Move and Rotate Gizmos. Notice that the sphere changes size, but the Gizmos don't. The Scene window in the editor shows you the view from a virtual **camera**, and the scroll feature zooms that camera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

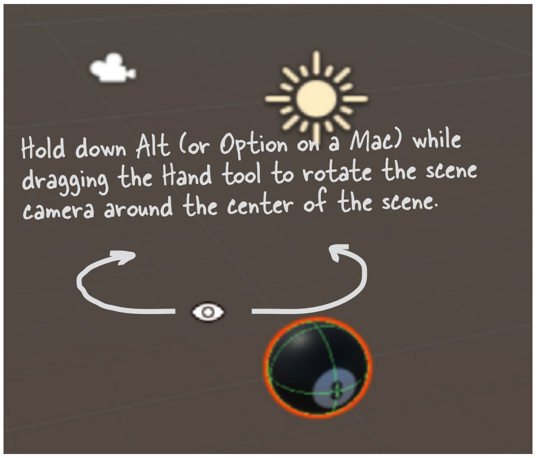
Press Q to select the **Hand tool**, or choose it from the toolbar. Your cursor will change to a hand.



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

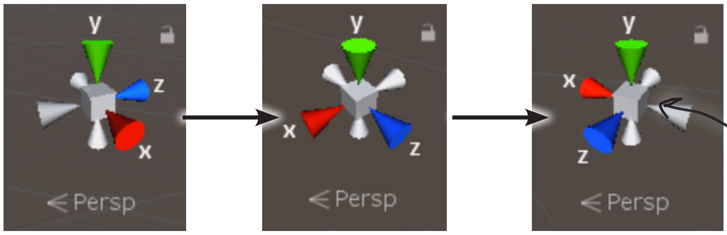
The Hand tool pans around the scene by changing the position and rotation of the scene camera. When the Hand tool is selected, you can click anywhere in the scene to pan.


Click and drag the Hand tool around the scene to pan the scene camera.


Hold down Alt (or Option on a Mac) while dragging the Hand tool to rotate the scene camera around the center of the scene.

When the Hand tool is selected, you can ***pan*** the scene camera by **clicking and dragging**, and you can ***rotate*** it by holding **down Alt (or Option) and dragging**. Use the **mouse scroll wheel** to zoom. Holding down the **right mouse button** lets you ***fly through the scene*** using the W-A-S-D keys.

When you rotate the scene camera, keep an eye on the **Scene Gizmo** in the upper-right corner of the Scene window. The Scene Gizmo always displays the camera's orientation—check it out as you use the Hand tool to move the Scene view camera. Click on the X, Y, and Z cones to snap the camera to an axis.



Click any of the cones in the Scene Gizmo to snap the camera to an axis. Drag them around to rotate the camera.

The Unity Manual has great tips on navigating scenes: https://docs.unity3d.com/Manual/SceneViewNavigation.html.

there are no Dumb Questions

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

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

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

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

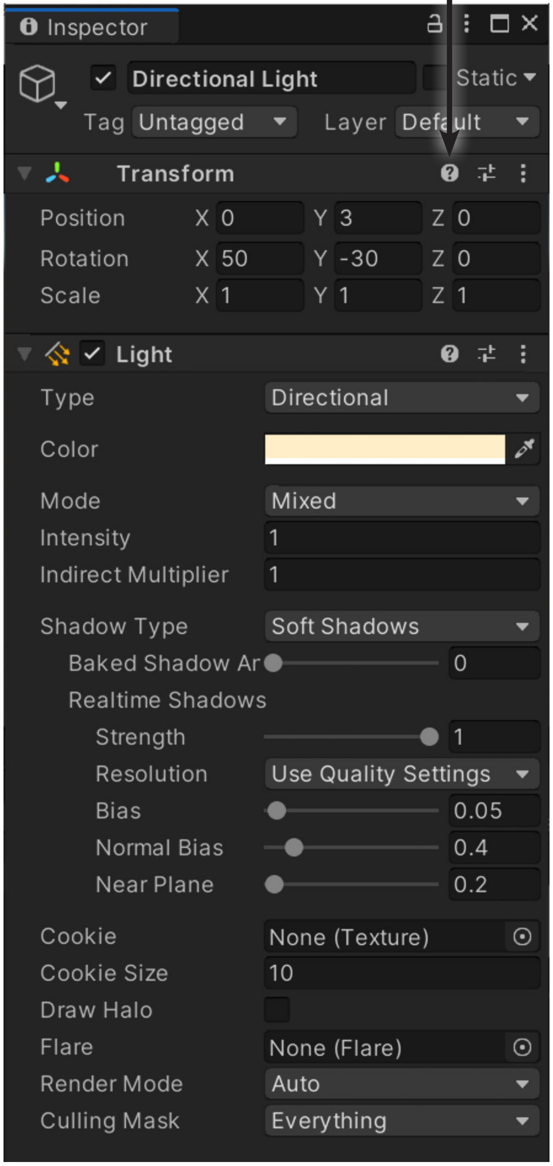
Q: If I add a Light component to any GameObject, does it become a light?

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

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

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

You can click on the Help icon for any component to bring up the Unity Manual page for it.



Directional Light ✓ Static
Tag Untagged Layer Default

Transform
Position X 0 Y 3 Z 0
Rotation X 50 Y -30 Z 0
Scale X 1 Y 1 Z 1

Light ✓
Type: Directional
Color
Mode: Mixed
Intensity: 1
Indirect Multiplier: 1
Shadow Type: Soft Shadows
Baked Shadow Ar: 0
Realtime Shadows
Strength: 1
Resolution: Use Quality Settings
Bias: 0.05
Normal Bias: 0.4
Near Plane: 0.2
Cookie: None (Texture)
Cookie Size: 10
Draw Halo
Flare: None (Flare)
Render Mode: Auto
Culling Mask: Everything

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

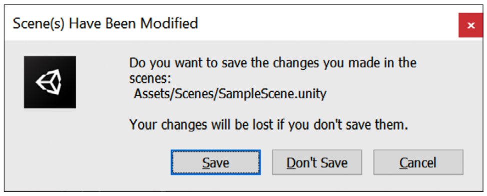
Get creative!

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 where you downloaded *8 Ball Texture.png* from.
- ★ Try adding other shapes by choosing Cube, Cylinder, or Capsule from the GameObject >> 3D Object menu.
- ★ Experiment with using different images as textures. See what happens to photos of people or scenery when you use them to create textures and add them to different shapes.
- ★ Can you create an interesting 3D scene out of shapes, textures, and lights?

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

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.


Scene(s) Have Been Modified

Do you want to save the changes you made in the scenes:
Assets/Scenes/SampleScene.unity

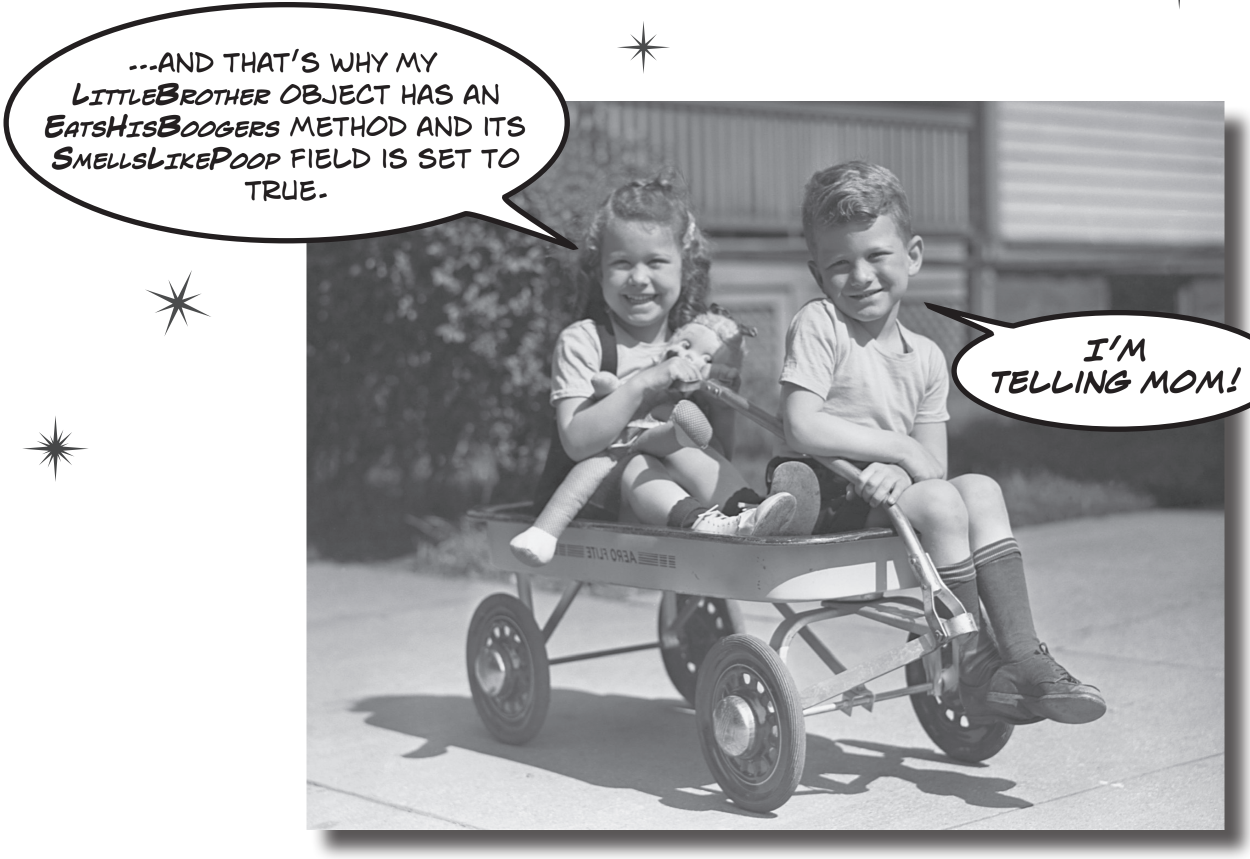
Your changes will be lost if you don't save them.

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 C# scripts and textures.
- The **Hierarchy window** shows all of the GameObjects in the scene.
- **GitHub for Unity** (https://unity.github.com) makes it easy to save your Unity projects in GitHub.
- **Unity Collaborate** also lets you back up projects to free cloud storage that comes with a Unity Personal account.

3 objects...get oriented!

Making code make sense

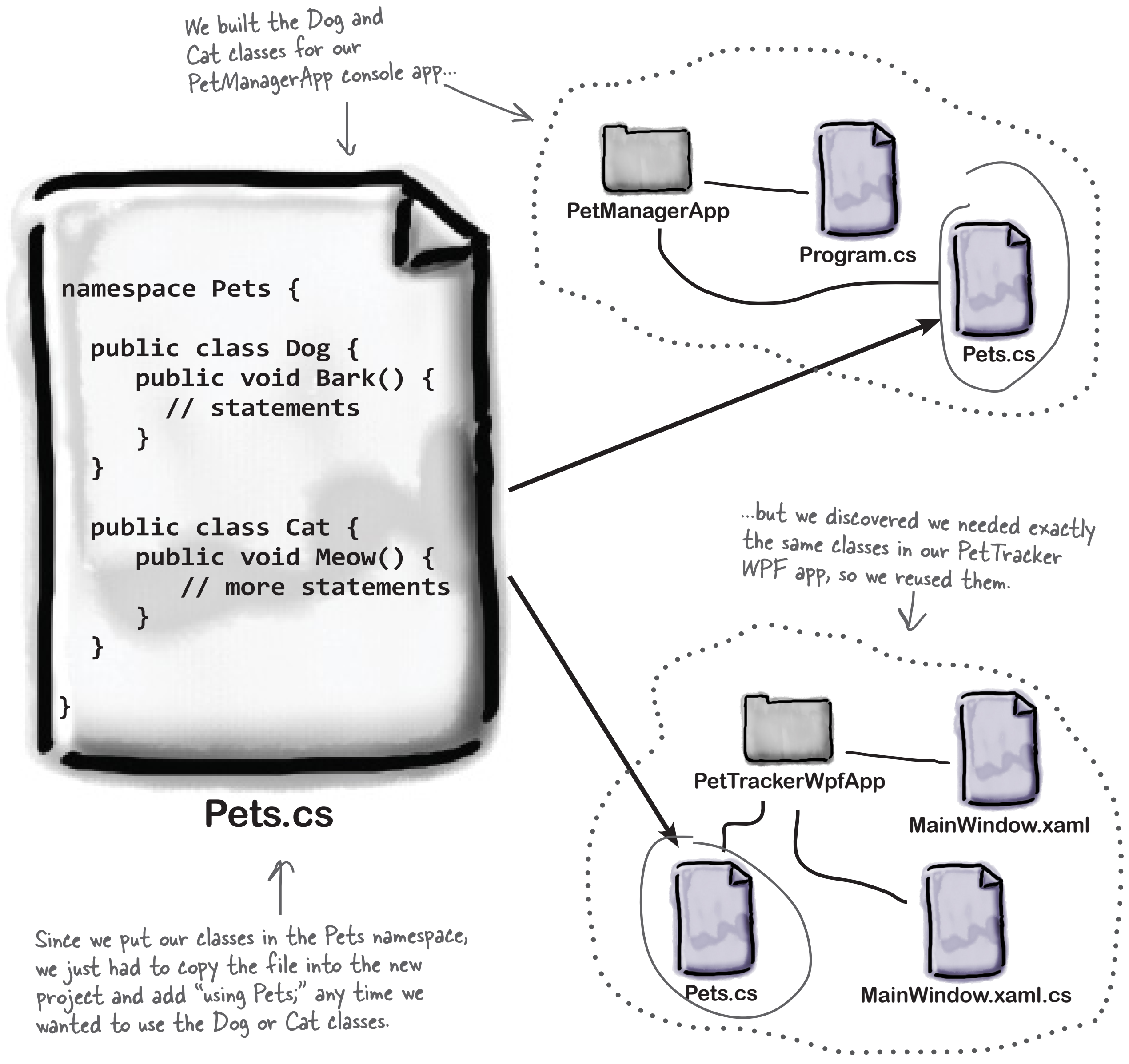

...AND THAT'S WHY MY LITTLEBROTHER OBJECT HAS AN EATSHISBOOGERS METHOD AND ITS SMELLSLIKEPOOP FIELD IS SET TO TRUE.
I'M TELLING MOM!
AERO FLITE

Every program you write solves a problem.

When you're building a program, it's always a good idea to start by thinking about what problem your program's supposed to solve. That's why objects are really useful. They let you structure your code based on the problem it's solving so that you can spend your time thinking about the problem you need to work on rather than getting bogged down in the mechanics of writing code. When you use objects right—and really put some thought into how you design them—you end up with code that's intuitive to write, and easy to read and change.

If code is useful, it gets reused

Developers have been reusing code since the earliest days of programming, and it's not hard to see why. If you've written a class for one program, and you have another program that needs code that does exactly the same thing, then it makes sense to **reuse** the same class in your new program.



```
namespace Pets {

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

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

}
```

We built the Dog and Cat classes for our PetManagerApp console app...
PetManagerApp
Program.cs
Pets.cs
...but we discovered we needed exactly the same classes in our PetTracker WPF app, so we reused them.
PetTrackerWpfApp
MainWindow.xaml
Pets.cs
MainWindow.xaml.cs
Pets.cs
Since we put our classes in the Pets namespace, we just had to copy the file into the new project and add "using Pets;" any time we wanted to use the Dog or Cat classes.

Some methods take parameters and return a value

You've seen methods that do things, like the SetUpGame method in Chapter 1 that sets up your game. Methods can do more than that: they can use **parameters** to get input, do something with that input, and then generate output with a **return value** that can be used by the statement that called the method.

Parameters start the input → **Method** does something → **Return value** sends output back

Parameters are values that the method uses as input. They're declared as variables that are included in the method declaration (between the parentheses). The return value is a value that's calculated or generated inside the method, and sent back to the statement that called that method. The type of the return value (like *string* or *int*) is called the **return type**. If a method has a return type, then it must use a **`return` statement**.

Here's an example of a method with two int parameters and an int return type:

```
int Multiply(int factor1, int factor2)
{
   int product = factor1 * factor2;
   return product;

}
```

The return type is int, so the method must return an int value.

This method takes two int parameters called factor1 and factor2 as input. They're treated just like int variables.

The return statement passes the value back to the statement that called the method.

The method takes two **parameters** called **`factor1`** and **`factor2`**. It uses the multiplication operator * to calculate the result, which it returns using the **`return`** keyword.

This code calls the Multiply method and stores the result in a variable called **`area`**:

```
int height = 179;
int width = 83;
int area = Multiply(height, width);
```

You can pass values like 3 and 5 to methods, like this: Multiply(3, 5)—but you can also use variables when you call your methods. It's fine if the variable names don't match the parameter names.

Do this! →

Since you're about to create methods that return values, right now is a perfect time to write some code and use the debugger to *really dig into how the* `return` *statement works*.

★ What happens when a method is done executing all of its statements? See for yourself—open up one of the programs you've written so far, place a breakpoint inside a method, then keep stepping through it.

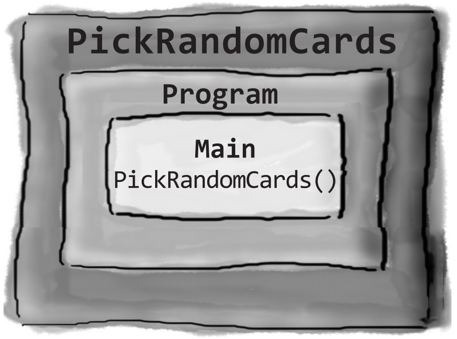
★ When the method runs out of statements, *it **returns** to the statement that called it* and continues executing the next statement after that.

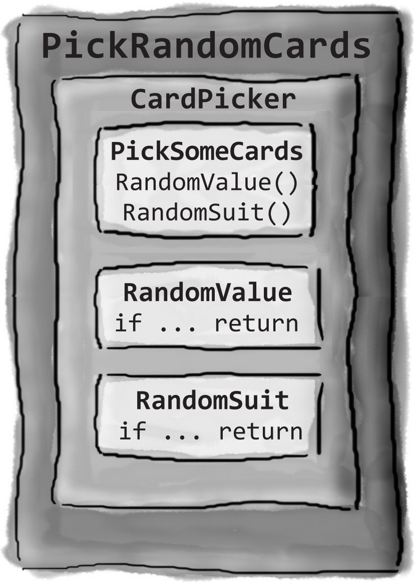
★ A method can also include a **`return`** statement, which causes it to immediately exit without executing any of its other statements. Try adding an extra `return` statement in the middle of a method, then step over it.

Let's build a program that picks some cards

In the first project in this chapter, you're going to build a .NET Core console app called PickRandomCards that lets you pick random playing cards. Here's what its structure will look like:

When you create the console app in Visual Studio, it will add a class called Program in a namespace that matches the project name, with a Main method that has the entry point.





You'll add another class called CardPicker with three methods. The Main method will call the PickSomeCards method in your new class.

Your PickSomeCards method will use string values to represent playing cards. If you want to pick five cards, you'll call it like this:

```
string[] cards = PickSomeCards(5);
```

The `cards` variable has a type that you haven't seen yet. The square brackets `[]` mean that it's an **array of strings**. Arrays let you use a single variable to store multiple values—in this case, strings with playing cards. Here's an example of a string array that the PickSomeCards method might return:

```
{ "10 of Diamonds",
  "6 of Clubs",
  "7 of Spades",
  "Ace of Diamonds",
  "Ace of Hearts" }
```

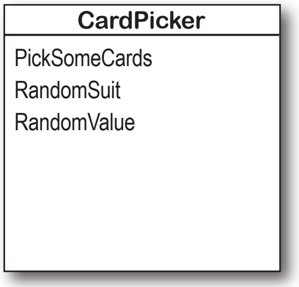
This is an array of five strings. Your card picker app will create arrays like this to represent a number of randomly selected cards.

After your array is generated, you'll use a `foreach` loop to write it to the console.

Create your PickRandomCards console app

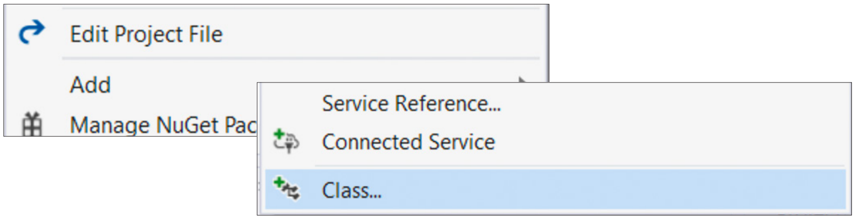
Do this!

Let's use what you've learned so far to create a program that picks a number of random cards. Open Visual Studio and **create a new Console App project called PickRandomCards**. Your program will include a class called CardPicker. Here's a class diagram that shows its name and methods:

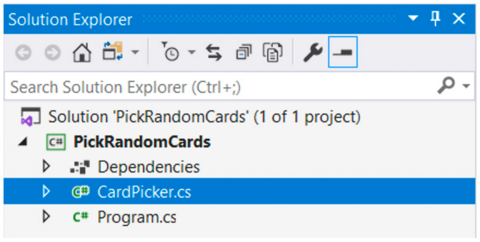


This is a class diagram. It's a rectangle with the class name on top and a list of its methods on the bottom. Your CardPicker class will have three methods named PickSomeCards, RandomSuit, and RandomValue.

Right-click on the PickRandomCards project in the Solution Explorer and **choose Add >> Class…** in Windows (or Add >> New Class… in macOS) from the pop-up menu. Visual Studio will prompt you for a class name—choose *CardPicker.cs*.



Visual Studio will create a brand-new class in your project called CardPicker:


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'PickRandomCards' (1 of 1 project)
PickRandomCards
Dependencies
CardPicker.cs
Program.cs

Your new class is empty—it starts with class `CardPicker` and a pair of curly braces, but there's nothing inside them. **Add a new method called PickSomeCards**. Here's what your class should look like:

```
class CardPicker
{
     public static string[] PickSomeCards(int numberOfCards)
     {

     }
 }
```

Make sure you include the public and static keywords. We'll talk more about them later in the chapter.

If you carefully entered this method declaration exactly as it appears here, you should see a red squiggly underline underneath PickSomeCards. What do you think it means?

Finish your PickSomeCards method

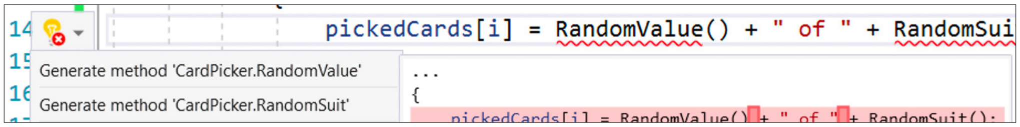
Now do this!

1 ***Your PickSomeCards method needs a return statement, so let's add one.*** Go ahead and fill in the rest of the method—and now that it uses a `return` statement to return a string array value, the error goes away:

```
class CardPicker
{
    public static string[] PickSomeCards(int numberOfCards)
    {
        string[] pickedCards = new string[numberOfCards];
        for (int i = 0; i < numberOfCards; i++)
        {
            pickedCards[i] = RandomValue() + " of " + RandomSuit();
        }
        return pickedCards;
    }
}
```

You made the red squiggly error underlines go away by returning a value with a type that matches the return type of the method.

2 ***Generate the missing methods.*** Your code now has different errors because it doesn't have RandomValue or RandomSuit methods. Generate these methods just like you did in Chapter 1. Use the Quick Actions icon in the left margin of the code editor—when you click it, you'll see options to generate both methods:



```
pickedCards[i] = RandomValue() + " of " + RandomSui
```

Generate method 'CardPicker.RandomValue'
Generate method 'CardPicker.RandomSuit'

```
...
{
    pickedCards[i] = RandomValue() + " of " + RandomSuit();
```

Go ahead and generate them. Your class should now have RandomValue and RandomSuit methods:

```
class CardPicker
{
    public static string[] PickSomeCards(int numberOfCards)
    {
        string[] pickedCards = new string[numberOfCards];
        for (int i = 0; i < numberOfCards; i++)
        {
            pickedCards[i] = RandomValue() + " of " + RandomSuit();
        }
        return pickedCards;
    }

    private static string RandomValue()
    {
        throw new NotImplementedException();
    }

    private static string RandomSuit()
    {
        throw new NotImplementedException();
    }
}
```

You used the IDE to generate these methods. It's OK if they're not in the same order—the order of the methods in a class doesn't matter.

3 ***Use return statements to build out your RandomSuit and RandomValue methods.***
A method can have more than one `return` statement, and when it executes one of those statements it immediately returns—and *does not execute any more statements in the method*.

Here's an example of how you could take advantage of return statements in a program. Let's imagine that you're building a card game, and you need methods to generate random card suits or values. We'll start by creating a random number generator, just like we used in the animal matching game in the first chapter. Add it just below the class declaration:

```
class CardPicker
{
    static Random random = new Random();
```

Now add code to your RandomSuit method that takes advantage of `return` statements to stop executing the method as soon as it finds a match. The random number generator's Next method can take two parameters: `random.Next(1, 5)` returns a number that's at least 1 but less than 5 (in other words, a random number from 1 to 4). Your RandomSuit method will use this to choose a random playing card suit:

```
private static string RandomSuit()
{
    // get a random number from 1 to 4
    int value = random.Next(1, 5);
    // if it's 1 return the string Spades
    if (value == 1) return "Spades";
    // if it's 2 return the string Hearts
    if (value == 2) return "Hearts";
    // if it's 3 return the string Clubs
    if (value == 3) return "Clubs";
    // if we haven't returned yet, return the string Diamonds
    return "Diamonds";
}
```

We added comments to explain exactly what's going on.

Here's a RandomValue method that generates a random value. See if you can figure out how it works:

```
private static string RandomValue()
{
    int value = random.Next(1, 14);
    if (value == 1) return "Ace";
    if (value == 11) return "Jack";
    if (value == 12) return "Queen";
    if (value == 13) return "King";
    return value.ToString();
}
```

Notice how we're returning `value.ToString()` and not just `value`? That's because `value` is an int variable, but the RandomValue method was declared with a string return type, so we need to convert `value` to a string. You can add `.ToString()` to any variable or value to convert it to a string.

The return statement causes your method to stop <u>immediately</u> and go back to the statement that called it.

Your finished CardPicker class

Here's the code for your finished CardPicker class. It should live inside a namespace that matches your project's name:

```
class CardPicker
{
    static Random random = new Random();

    public static string[] PickSomeCards(int numberOfCards)
    {
        string[] pickedCards = new string[numberOfCards];
        for (int i = 0; i < numberOfCards; i++)
        {
            pickedCards[i] = RandomValue() + " of " + RandomSuit();
        }
        return pickedCards;
    }

    private static string RandomValue()
    {
        int value = random.Next(1, 14);
        if (value == 1) return "Ace";
        if (value == 11) return "Jack";
        if (value == 12) return "Queen";
        if (value == 13) return "King";
        return value.ToString();
    }

    private static string RandomSuit()
    {
        // get a random number from 1 to 4
        int value = random.Next(1, 5);
        // if it's 1 return the string Spades
        if (value == 1) return "Spades";
        // if it's 2 return the string Hearts
        if (value == 2) return "Hearts";
        // if it's 3 return the string Clubs
        if (value == 3) return "Clubs";
        // if we haven't returned yet, return Diamonds
        return "Diamonds";
    }
}
```

This is a static field called "random" that we'll use to generate random numbers.

We added these comments to help you understand how the RandomSuit method works. Try adding similar comments to the RandomValue method that explain how it works.

We haven't talked about fields much... yet.

Your CardPicker class has a **field** called **random**. You've seen fields in the animal matching game in Chapter 1, but we still haven't really worked with them much. Don't worry—we'll talk a lot about fields and the `static` keyword later in the chapter.

BRAIN POWER

You used the `public` and `static` keywords when you added PickSomeCards. Visual Studio kept the `static` keyword when it generated the methods, and declared them as `private`, not `public`. What do you think these keywords do?

Exercise

Now that your CardPicker class has a method to pick random cards, you've got everything you need to finish your console app by **filling in the Main method**. You just need a few useful methods to make your console app read a line of input from the user and use it to pick a number of cards.

Useful method #1: Console.Write

You've already seen the Console.WriteLine method. Here's its cousin, Console.Write, which writes text to the console but doesn't add a new line at the end. You'll use it to display a message to the user:

```
Console.Write("Enter the number of cards to pick: ");
```

Useful method #2: Console.ReadLine

The Console.ReadLine method reads a line of text from the input and returns a string. You'll use it to let the user tell you how many cards to pick:

```
string line = Console.ReadLine();
```

Useful method #3: int.TryParse

Your CardPicker.PickSomeCards method takes an int parameter. The line of input you get from the user is a string, so you'll need a way to convert it to an int. You'll use the int.TryParse method for that:

```
if (int.TryParse(line, out int numberOfCards))
{
  // this block is executed if line COULD be converted to an int
  // value that's stored in a new variable called numberOfCards
}
else
{
   // this block is executed if line COULD NOT be converted to an int
}
```

In Chapter 2 you used the int.TryParse method in your TextBox event handler to make it only accept numbers. Take a minute and have another look at how that event handler works.

Put it all together

Your job is to take these three new pieces and put them together in a brand-new Main method for your console app. Modify your *Program.cs* file and replace the "Hello World!" line in the Main method with code that does this:

- ★ Use Console.Write to ask the user for the number of cards to pick.
- ★ Use Console.ReadLine to read a line of input into a string variable called `line`.
- ★ Use int.TryParse to try to convert it to an int variable called `numberOfCards`.
- ★ If the user input ***could be converted*** to an int value, use your CardPicker class to pick the number of cards that the user specified: CardPicker.PickSomeCards(numberOfCards). Use a string[] variable to save the results, then use a `foreach` loop to call Console.WriteLine on each card in the array. Flip back to Chapter 1 to see an example of a `foreach` loop—you'll use it to loop through every element of the array. Here's the first line of the loop: foreach (string card in CardPicker.PickSomeCards(numberOfCards))
- ★ If the user input ***could not be converted***, use Console.WriteLine to write a message to the user indicating that the number was not valid.

While you're working on your program's Main method, take a look at its return type. What do you think is going on there?

Here's the Main method for your console app. It prompts the user for the number of cards to pick, attempts to convert it to an int, and then uses the PickSomeCards method in the CardPicker class to pick that number of cards. PickSomeCards returns each of the picked cards in an array of strings, so it uses a `foreach` loop to write each of them to the console.

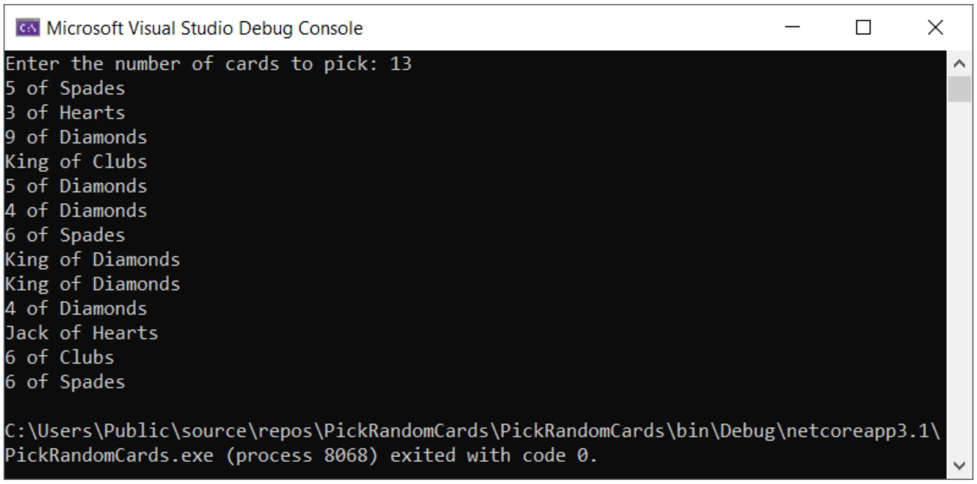
```
static void Main(string[] args)
{
    Console.Write("Enter the number of cards to pick: ");
    string line = Console.ReadLine();
    if (int.TryParse(line, out int numberOfCards))
    {
        foreach (string card in CardPicker.PickSomeCards(numberOfCards))
        {
            Console.WriteLine(card);
        }
    }
    else
    {
        Console.WriteLine("Please enter a valid number.");
    }
}
```

This Main method replaces the one that prints "Hello World!" that Visual Studio created for you in Program.cs.

This foreach loop executes Console.WriteLine(card) for each element in the array returned by PickSomeCards.

Your Main method uses void as the return type to tell C# that it doesn't return a value. A method with a void return type is not required to have a `return` statement.

Here's what it looks like when you run your console app:



```
Enter the number of cards to pick: 13
5 of Spades
3 of Hearts
9 of Diamonds
King of Clubs
5 of Diamonds
4 of Diamonds
6 of Spades
King of Diamonds
King of Diamonds
4 of Diamonds
Jack of Hearts
6 of Clubs
6 of Spades

C:\Users\Public\source\repos\PickRandomCards\PickRandomCards\bin\Debug\netcoreapp3.1\
PickRandomCards.exe (process 8068) exited with code 0.
```

Take the time to <u>really understand</u> how this program works—this is a great opportunity to use the Visual Studio debugger to help you explore your code. Place a breakpoint on the first line of the Main method, then use Step Into (F11) to step through the entire program. Add a watch for the `value` variable, and keep your eye on it as you step through the RandomSuit and RandomValue methods.

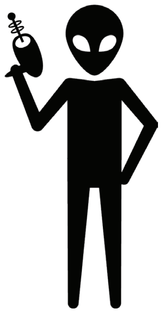
Ana's working on her next game

Meet Ana. She's an indie game developer. Her last game sold thousands of copies, and now she's getting started on her next one.

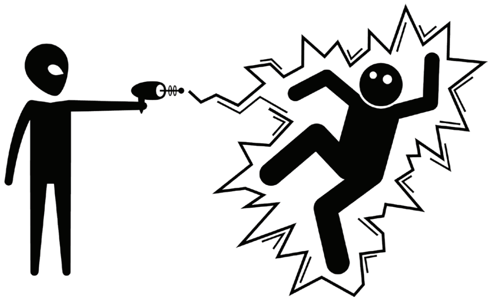
IN MY NEXT GAME, THE PLAYER IS DEFENDING THEIR TOWN FROM ALIEN INVADERS.

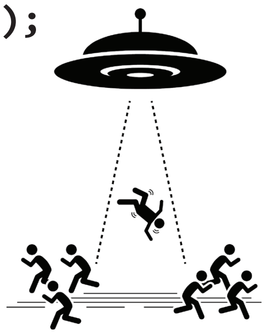
Ana's started working on some **prototypes**. She's been working on the code for the alien enemies that the player has to avoid in one exciting part of the game, where the player needs to escape from their hideout while the aliens search for them. Ana's written several methods that define the enemy behavior: searching the last location the player was spotted, giving up the search after a while if the player wasn't found, and capturing the player if the enemy gets too close.

```
SearchForPlayer();
```



```
if (SpottedPlayer()) {
    CommunicatePlayerLocation();
}
```

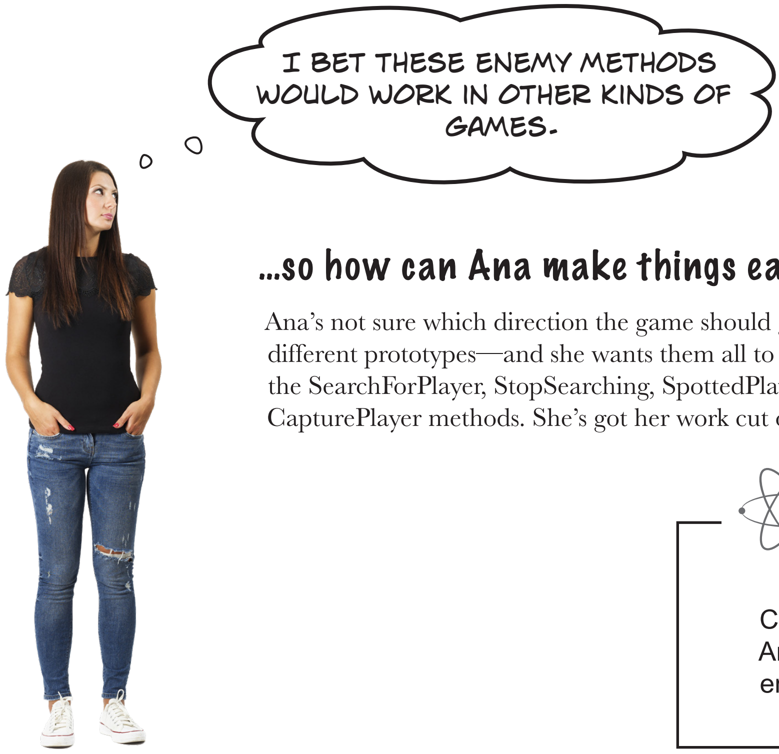




```
CapturePlayer();
```

Ana's game is evolving...

The humans versus aliens idea is pretty good, but Ana's not 100% sure that's the direction she wants to go in. She's also thinking about a nautical game where the player has to evade pirates. Or maybe it's a zombie survival game set on a creepy farm. In all three of those ideas, she thinks the enemies will have different graphics, but their behavior can be driven by the same methods.


I BET THESE ENEMY METHODS WOULD WORK IN OTHER KINDS OF GAMES.

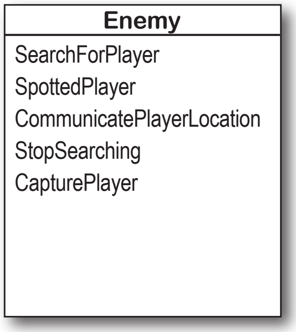
...so how can Ana make things easier for herself?

Ana's not sure which direction the game should go in, so she wants to make a few different prototypes—and she wants them all to have the same code for the enemies, with the SearchForPlayer, StopSearching, SpottedPlayer, CommunicatePlayerLocation, and CapturePlayer methods. She's got her work cut out for her.

BRAIN POWER

Can you think of a good way for Ana to use the same methods for enemies in different prototypes?

I PUT ALL OF THE ENEMY BEHAVIOR METHODS INTO A SINGLE ***ENEMY*** CLASS. CAN I ***REUSE THE CLASS*** IN EACH OF MY THREE DIFFERENT GAME PROTOTYPES?



Enemy
SearchForPlayer
SpottedPlayer
CommunicatePlayerLocation
StopSearching
CapturePlayer

Game design... and beyond

Prototypes

A **prototype** is an early version of your game that you can play, test, learn from, and improve. A prototype can be a really valuable tool to help you make changes early. Prototypes are especially useful because they let you rapidly experiment with a lot of different ideas before you've made permanent decisions.

- The first prototype is often a **paper prototype**, where you lay out the core elements of the game on paper. For example, you can learn a lot about your game by using sticky notes or index cards for the different elements of the game, and drawing out levels or play areas on large pieces of paper to move them around.
- One good thing about building prototypes is that they help you **get from an idea to a working, playable game** very quickly. You learn the most about a game (or any kind of program) when you get working software into the hands of your players (or users).
- Most games will go through **many prototypes**. This is your chance to try out lots of different things and learn from them. If something doesn't go well, think of it as an experiment, not a mistake.
- Prototyping is a **skill**, and just like any other skill, ***you get better at it with practice***. Luckily, building prototypes is also fun, and a great way to get better at writing C# code.

Prototypes aren't just used for games! When you need to build any kind of program, it's often a great idea to build a prototype first to experiment with different ideas.

Build a paper prototype for a classic game

Paper prototypes are really useful for helping you figure out how a game will work before you start building it, which can save you a lot of time. There's a fast way to get started building them—all you need is some paper and a pen or pencil. Start by choosing your favorite classic game. Platform games work especially well, so we chose one of the **most popular, most recognizable** classic video games ever made... but you can choose any game you'd like! Here's what to do next.

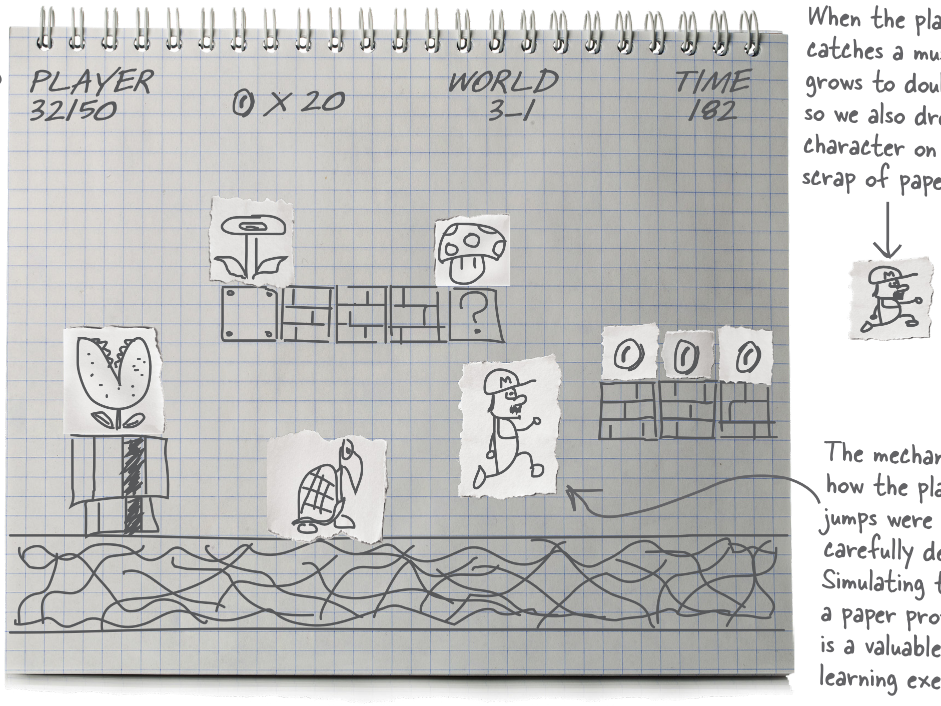
Draw this!

1. **Draw the background on a piece of paper.** Start your prototype by creating the background. In our prototype, the ground, bricks, and pipe don't move, so we drew them on the paper. We also added the score, time, and other text at the top.

2. **Tear small scraps of paper and draw the moving parts.** In our prototype, we drew the characters, the piranha plant, the mushroom, the fire flower, and the coins on separate scraps. If you're not an artist, that's absolutely fine! Just draw stick figures and rough shapes. Nobody else ever has to see this!

3. **"Play" the game.** This is the fun part! Try to simulate player movement. Drag the player around the page. Make the non-player characters move too. It helps to spend a few minutes playing the game, then go back to your prototype and see if you can really reproduce the motion as closely as possible. (It will feel a little weird at first, but that's OK!)

The text at the top of the screen is called the HUD, or head-up display. It's usually drawn on the background in a paper prototype.



When the player catches a mushroom he grows to double his size, so we also drew a small character on a separate scrap of paper.

The ground, bricks, and pipe don't move, so we drew them on the background paper. There's no rule about what goes on the background and what moves around.

The mechanics of how the player jumps were really carefully designed. Simulating them in a paper prototype is a valuable learning exercise.

PAPER PROTOTYPES LOOK LIKE THEY'D BE USEFUL FOR MORE THAN JUST GAMES. I BET I CAN USE THEM IN MY OTHER PROJECTS, TOO.

All of the tools and ideas in "Game design... and beyond" sections are important programming skills that go way beyond just game development—but we've found that they're easier to learn when you try them with games first.

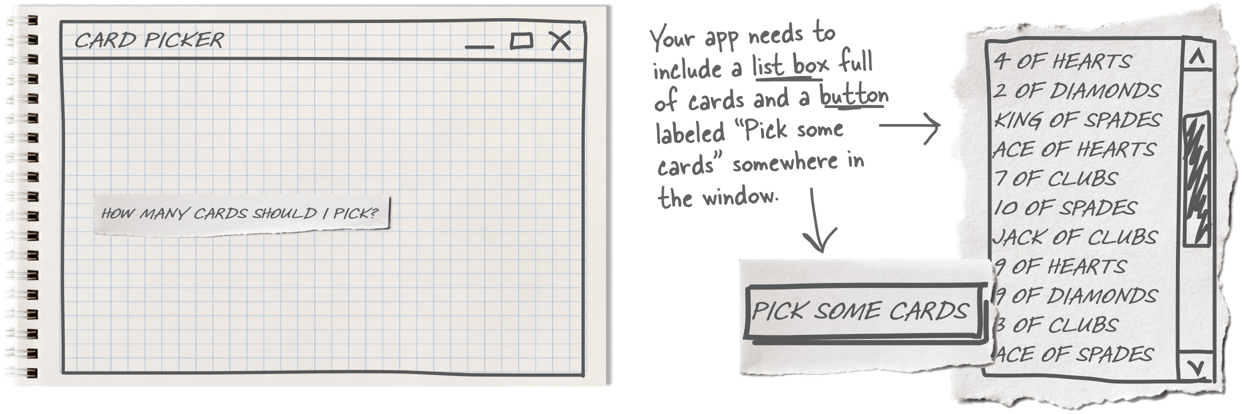
Yes! A paper prototype is a great first step for <u>any</u> project.

If you're building a desktop app, a mobile app, or any other project that has a user interface, building a paper prototype is a great way to get started. Sometimes you need to create a few paper prototypes before you get the hang of it. That's why we started with a paper prototype for a classic game…because that's a great way to learn how to build paper prototypes. **Prototyping is a really valuable skill for <u>any</u> kind of developer**, not just a game developer.

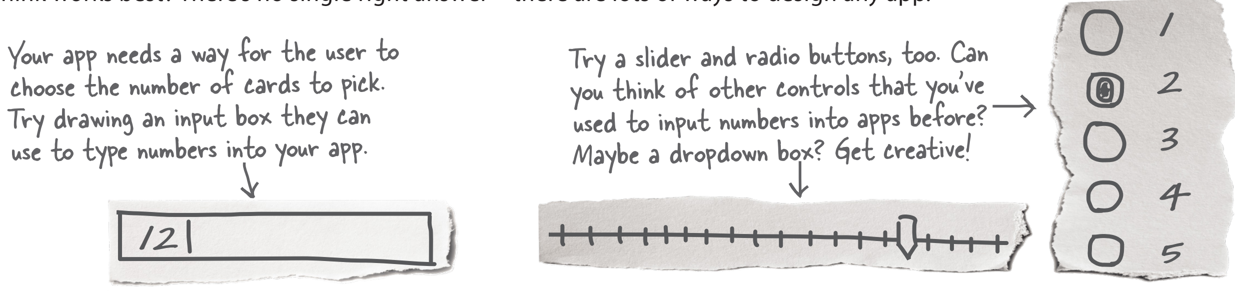
Sharpen your pencil

In the next project, you'll create a WPF app that uses your CardPicker class to generate a set of random cards. In this paper-and-pencil exercise, you'll build a paper prototype of your app to try out various design options.

Start by drawing the window frame on a large piece of paper and a label on a smaller scrap of paper.



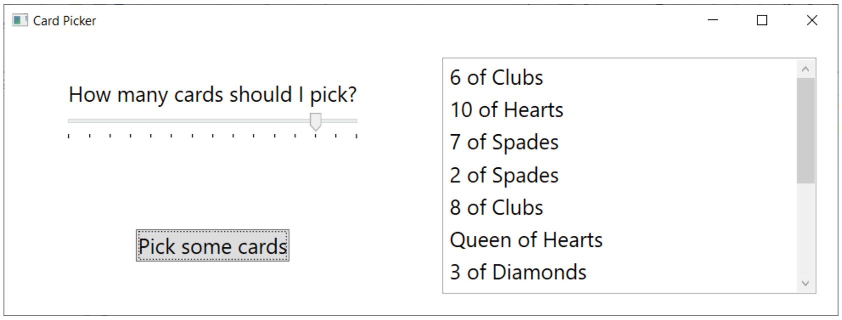
Next, draw a bunch of different types of controls on more small scraps of paper. Drag them around the window and experiment with ways to fit them together. What design do you think works best? There's no single right answer—there are lots of ways to design any app.



Your app needs a way for the user to choose the number of cards to pick. Try drawing an input box they can use to type numbers into your app.
12
Try a slider and radio buttons, too. Can you think of other controls that you've used to input numbers into apps before? Maybe a dropdown box? Get creative!
1
2
3
4
5

Up next: build a WPF version of your card picking app

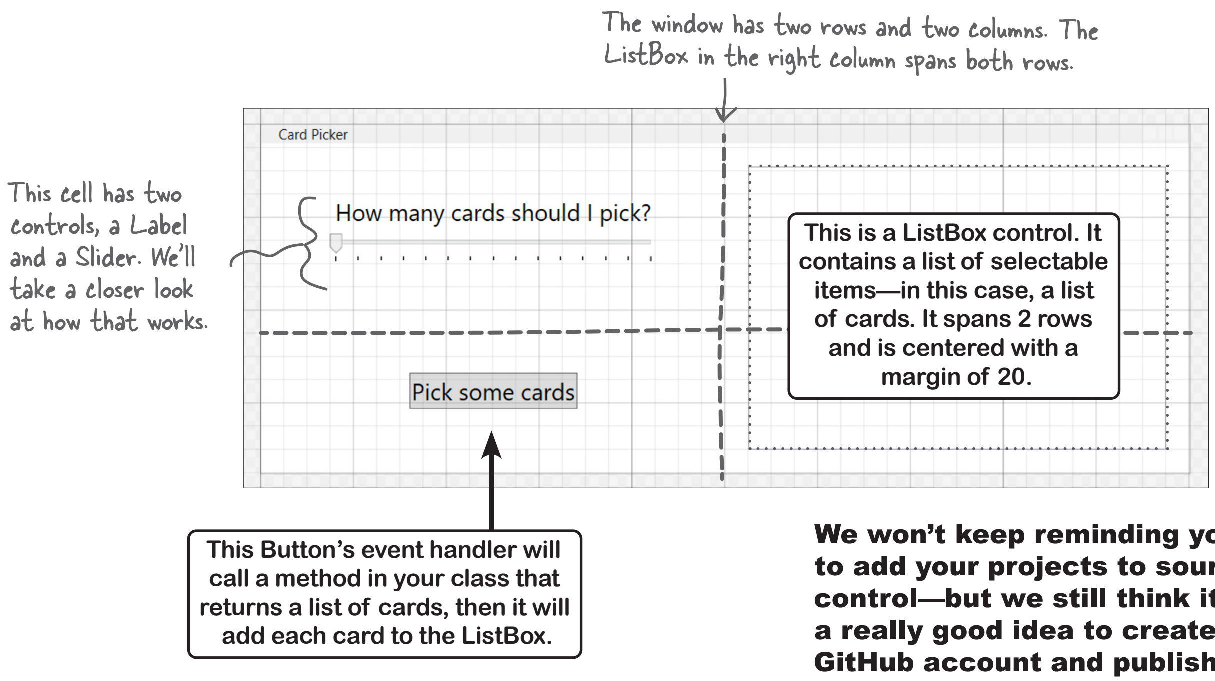
In the next project, you'll build a WPF app called PickACardUI. Here's what it will look like:



We decided to go with a slider to choose the number of cards. That doesn't mean it's the only way to design this app! Did you come up with a different design with your paper prototype? That's OK! There are many ways to design any app, and there's almost never a single right (or wrong) answer.

Your PickACardUI app will let you use a Slider control to choose the number of random cards to pick. When you've selected the number of cards, you'll click a button to pick them and add them to a ListBox.

Here's how the window will be laid out:


The window has two rows and two columns. The ListBox in the right column spans both rows.
Card Picker
How many cards should I pick?
Pick some cards
This cell has two controls, a Label and a Slider. We'll take a closer look at how that works.
This is a ListBox control. It contains a list of selectable items—in this case, a list of cards. It spans 2 rows and is centered with a margin of 20.
This Button's event handler will call a method in your class that returns a list of cards, then it will add each card to the ListBox.

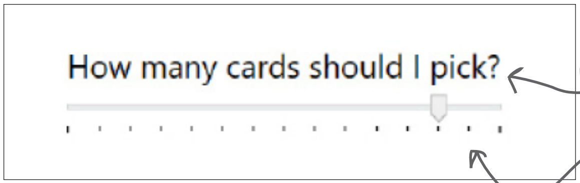
We won't keep reminding you to add your projects to source control—but we still think it's a really good idea to create a GitHub account and publish all of your projects to it!

There are ASP.NET Core versions of all of the WPF projects in this book that feature screenshots from Visual Studio for Mac.

Go to the Visual Studio for Mac Learner's Guide for the Mac version of this project.

A StackPanel is a container that stacks other controls

Your WPF app will use a Grid to lay out its controls, just like you used in your matching game. Before you start writing code, let's take a closer look at the two controls in the upper-left cell of the grid:

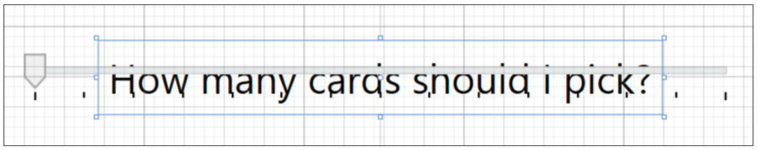

How many cards should I pick?
This is a Label control...
...and this is a Slider control.

So how do we stack them on top of each other like that? We ***could*** try putting them in the same cell in the grid:

```
<Grid>
    <Label HorizontalAlignment="Center" VerticalAlignment="Center" Margin="20"
        Content="How many cards should I pick?" FontSize="20"/>
    <Slider VerticalAlignment="Center" Margin="20"
        Minimum="1" Maximum="15" Foreground="Black"
        IsSnapToTickEnabled="True" TickPlacement="BottomRight" />
</Grid>
```

This is XAML for a Slider control. We'll take a closer look at it when you put your form together.

But that just causes them to overlap each other:

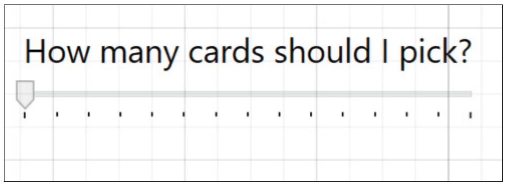


That's where a **StackPanel control** comes in handy. A StackPanel is a container control—like a Grid, its job is to contain other controls and make sure they go in the right place in the window. While the Grid lets you arrange controls in rows and columns, a StackPanel lets you arrange controls ***in a horizontal or vertical stack***.

Let's take the same Label and Slider controls, but this time use a StackPanel to lay them out so the Label is stacked on top of the Slider. Notice that we moved the alignment and margin properties to the StackPanel—we want the panel itself to be centered, with a margin around it:

```
<StackPanel HorizontalAlignment="Center" VerticalAlignment="Center" Margin="20" >
    <Label Content="How many cards should I pick?" FontSize="20" />
    <Slider Minimum="1" Maximum="15" Foreground="Black"
            IsSnapToTickEnabled="True" TickPlacement="BottomRight" />
</StackPanel>
```

The StackPanel will make the controls in the cell look the way we want them to:



So that's how the project will work. Now let's get started building it!

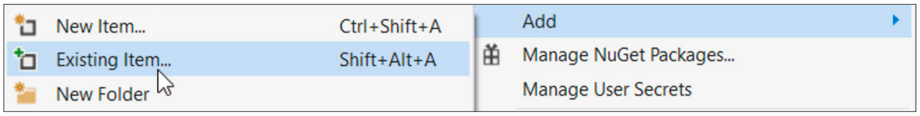
Reuse your CardPicker class in a new WPF app

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

Reuse this!

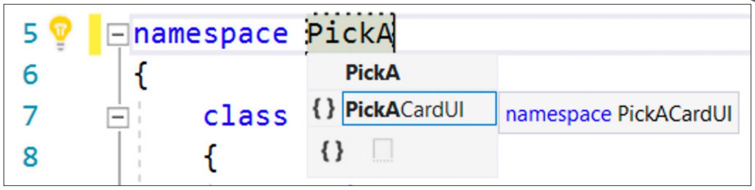
1. **Create a new WPF app called PickACardUI.**
   You'll follow exactly the same steps that you used to create your animal matching game in Chapter 1:
   - Open Visual Studio and create a new project.
   - Select **WPF App** (**.NET**).
   - Name your new app **PickACardUI**. Visual Studio will create the project, adding *MainWindow.xaml* and *MainWindow.xaml.cs* files that have the namespace PickACardUI.

2. **Add the CardPicker class that you created for your Console App project.**
   Right-click on the project name and choose **Add >> Existing Item...** from the menu.



   Navigate to the folder with your console app and select *CardPicker.cs* to add it to your project. Your WPF project should now have a copy of the *CardPicker.cs* file from your console app.

3. **Change the namespace for the CardPicker class.**
   Double-click on *CardPicker.cs* in the Solution Explorer. It still has the namespace from the console app. **Change the namespace** to match your project name. The IntelliSense pop-up will suggest the namespace PickACardUI—**press Tab to accept the suggestion**:



```
namespace PickA
{
    class
    {
```

You're changing the namespace in the CardPicker.cs file to match the namespace that Visual Studio used when it created the files in your new project so you can use your CardPicker class in your new project's code.

   Now your CardPicker class should be in the PickACardUI namespace:

```
namespace PickACardUI
{
    class CardPicker
    {
```

   Congratulations, you've reused your CardPicker class! You should see the class in the Solution Explorer, and you'll be able to use it in the code for your WPF app.

Use a Grid and StackPanel to lay out the main window

Back in Chapter 1 you used a Grid to lay out your animal matching game. Take a few minutes and flip back through the part of the chapter where you laid out the grid, because you're going to do the same thing to lay out your window.

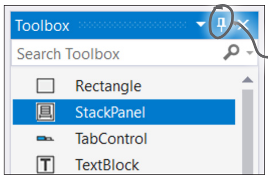
1. **Set up the rows and columns.** Follow the same steps from Chapter 1 to **add two rows and two columns** to your grid. If you get the steps right, you should see these row and column definitions just below the `<Grid>` tag in the XAML:

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

You can use the Visual Studio designer to add two equal rows and two equal columns. If you run into trouble, you can just type the XAML directly into the editor.

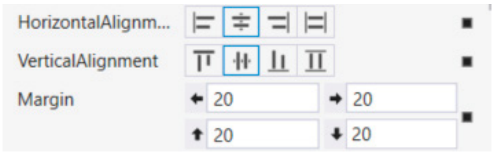
2. **Add the StackPanel.** It's a little difficult to work with an empty StackPanel in the visual XAML designer because it's hard to click on, so we'll do this in the XAML code editor. **Double-click on StackPanel in the Toolbox** to add an empty StackPanel to the grid. You should see:

```
        </Grid.ColumnDefinitions>

        <StackPanel/>

    </Grid>
</Window>
```



It'll be easier to drag controls out of the Toolbox if you use the pushpin in the upper-right corner of the Toolbox panel to pin it to the window.

3. **Set the StackPanel's properties.** When you double-clicked on StackPanel in the Toolbox, it added ***a StackPanel with no properties***. By default it's in the upper-left cell in the grid, so now we just want to set its alignment and margin. **Click on the StackPanel tag *in the XAML editor*** to select it. Once it's selected in the code editor, you'll see its properties in the Properties window. Set the vertical and horizontal alignment to `Center` and all of the margins to `20`.



HorizontalAlignm...	Center	
VerticalAlignment	Center	
Margin	← 20	→ 20
	↑ 20	↓ 20

When you click on the control in the XAML code editor and use the Properties window to edit its properties, you'll see the XAML will get updated immediately.

You should now have a StackPanel like this in your XAML code:

```
<StackPanel HorizontalAlignment="Center" VerticalAlignment="Center" Margin="20" />
```

This means all of the margins are set to 20. You might also see the Margin property set to "20, 20, 20, 20"—it means the same thing.

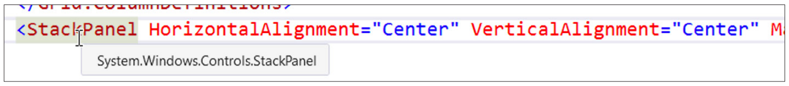
Lay out your Card Picker desktop app's window

Lay out your new card picker app's window so it has the user controls on the left and displays the picked cards on the right. You'll use a **StackPanel** in the upper-left cell. It's a **container**, which means it contains other controls, just like a Grid. But instead of laying the controls out in cells, it stacks them either horizontally or vertically. Once your StackPanel is laid out with a Label and Slider, you'll add ListBox control, just like the one you used in Chapter 2.

Design this!

1 Add a Label and Slider to your StackPanel.

A StackPanel is a container. When a StackPanel doesn't contain any other controls, *you can't see it in the designer*, which makes it hard to drag controls onto it. Luckily, it's just as fast to add controls to it as it is to set its properties. **Click on the StackPanel to select it.**



```
<StackPanel HorizontalAlignment="Center" VerticalAlignment="Center" M
```

System.Windows.Controls.StackPanel

While the StackPanel is selected, **double-click on Label in the Toolbox** to put a new Label control *inside the StackPanel*. The Label will appear in the designer, and a `Label` tag will appear in the XAML code editor.

Next, expand the *All WPF Controls* section in the Toolbox and **double-click on Slider**. Your upper-left cell should now have a StackPanel that contains a Label stacked on top of a Slider.

2 Set the properties for the Label and Slider controls.

Now that your StackPanel has a Label and a Slider, you just need to set their properties:

★ Click on the Label in the designer. Expand the Common section in the Properties window and set its content to `How many cards should I pick?`—then expand the Text section and set its font size to `20px`.

★ Press Escape to deselect the Label, then **click on the Slider in the designer** to select it. Use the Name box at the top of the Properties window to change its name to `numberOfCards`.

★ Expand the Layout section and use the square (■) to reset the width.

★ Expand the Common section and set its Maximum property to `15`, Minimum to `1`, AutoToolTipPlacement to `TopLeft`, and TickPlacement to `BottomRight`. Then click the caret (⌄) to expand the Layout section and expose additional properties, including the IsSnapToTickEnabled property. Set it to `True`.

★ Let's make ticks a little easier to see. Expand the Brush section in the Properties window and **click on the large rectangle to the right of Foreground**—this will let you use the color selector to choose the foreground color for the slider. Click in the R box and set it to `0`, then set G and B to `0` as well. The Foreground box should now be black, and the tick marks under the slider should be black.

The XAML should look like this—if you're having trouble with the designer, just edit the XAML directly:

```
<StackPanel HorizontalAlignment="Center" VerticalAlignment="Center" Margin="20">
    <Label Content="How many cards should I pick?" FontSize="20"/>
    <Slider x:Name="numberOfCards" Minimum="1" Maximum="15" TickPlacement="BottomRight"
     IsSnapToTickEnabled="True" AutoToolTipPlacement="TopLeft" Foreground="Black"/>
</StackPanel>
```

3 Add a Button to the lower-left cell.

Drag a Button out of the toolbox and into the lower-left cell of the grid and set its properties:

- ★ Expand the Common section and set its Content property to `Pick some cards`.
- ★ Expand the Text section and set its font size to `20px`.
- ★ Expand the Layout section. Reset its margins, width, and height. Then set its vertical and horizontal alignment to `Center` (≑ and ⫲).

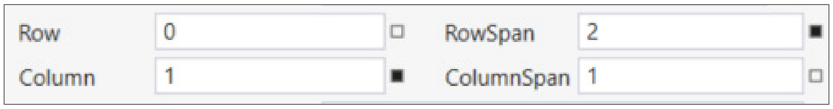
The XAML for your Button control should look like this:

```
<Button Grid.Row="1" Content="Pick some cards" FontSize="20"
        HorizontalAlignment="Center" VerticalAlignment="Center" />
```

4 Add a ListBox that fills the right half of the window by spanning two rows.

Drag a ListBox control into the upper-right cell and set its properties:

- ★ Use the Name box at the top of the Properties window to set the ListBox's name to `listOfCards`.
- ★ Expand the Text section and set its font size to `20px`.
- ★ Expand the Layout section. Set its margins to `20`, just like you did with the StackPanel control. Make sure its width, height, horizontal alignment, and vertical alignment are reset.
- ★ Make sure Row is set to 0 and Column is set to 1. Then **set the RowSpan to 2** so that the ListBox takes up the entire column and stretches across both rows:



Row	0	□	RowSpan	2	■
Column	1	■	ColumnSpan	1	□

The XAML for your ListBox control should look like this:

```
<ListBox x:Name="listOfCards" Grid.Column="1" Grid.RowSpan="2"
         FontSize="20" Margin="20,20,20,20"/>
```

It's OK if this value is just "20" instead of "20, 20, 20, 20"—that means the same thing.

5 Set the window title and size.

When you create a new WPF app, Visual Studio creates a main window that's 450 pixels wide and 800 pixels tall with the title "Main Window." Let's resize it, just like you did with the animal matching game:

- ★ Click on the window's title bar in the designer to select the window.
- ★ Use the Layout section to set the width to **`300`**.
- ★ Use the Common section to set the title to **`Card Picker`**.

Scroll to the top of the XAML editor and look at the last line of the `Window` tag. You should see these properties:

```
Title="Card Picker" Height="300" Width="800"
```

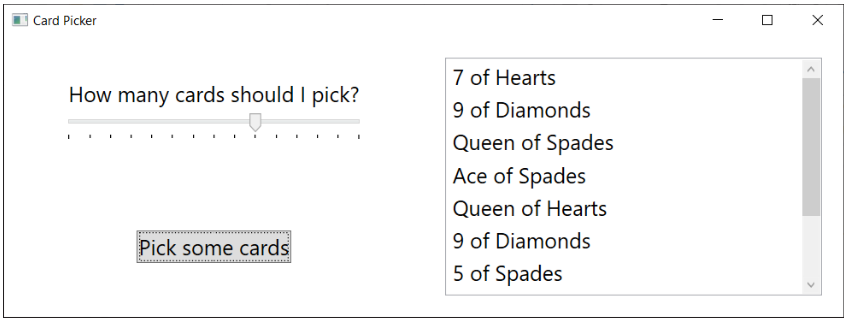
6 Add a Click event handler to your Button control.

The **code-behind**—tthe C# code in *MainWindow.xaml.cs* that's joined to your XAML—consists of a single method. Double-click on the button in the designer—the IDE will add a method called Button_Click and make it the Click event handler, just like it did in Chapter 1. Here's the code for your new method:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    string[] pickedCards = CardPicker.PickSomeCards((int)numberOfCards.Value);
    listOfCards.Items.Clear();
    foreach (string card in pickedCards)
    {
        listOfCards.Items.Add(card);
    }
}
```

The C# code joined to your XAML window that contains the event handlers is called the code-behind.

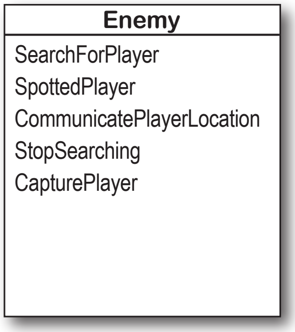
Now run your app. Use the slider to choose the number of random cards to pick, then press the button to add them to the ListBox. ***Nice work!***



7 of Hearts
9 of Diamonds
Queen of Spades
Ace of Spades
Queen of Hearts
9 of Diamonds
5 of Spades

BULLET POINTS

- Classes have methods that contain statements that perform actions. Well-designed classes have sensible method names.
- Some methods have a **return type**. You set a method's return type in its declaration. A method with a declaration that starts with the `int` keyword returns an int value. Here's an example of a statement that returns an int value: `return 37;`
- When a method has a return type, it **must** have a `return` statement that returns a value that matches a return type. So if a method declaration has the string return type then you need a `return` statement that returns a string.
- As soon as a `return` statement in a method executes, your program jumps back to the statement that called the method.
- Not all methods have a return type. A method with a declaration that starts `public void` doesn't return anything at all. You can still use a `return` statement to exit a void method, as in this example: `if (finishedEarly) { return; }`
- Developers often want to **reuse** the same code in multiple programs. Classes can help you make your code more reusable.
- When you **select a control** in the XAML code editor, you can edit its properties in the Properties window.



Ana's prototypes look great…

Ana found out that whether her player was being chased by an alien, a pirate, a zombie, or an evil killer clown, she could use the same methods from her Enemy class to make them work. Her game is starting to shape up.

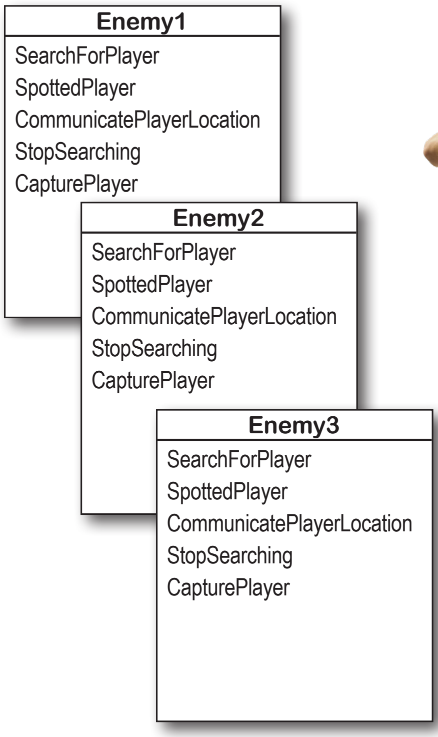
…but what if she wants more than one enemy?

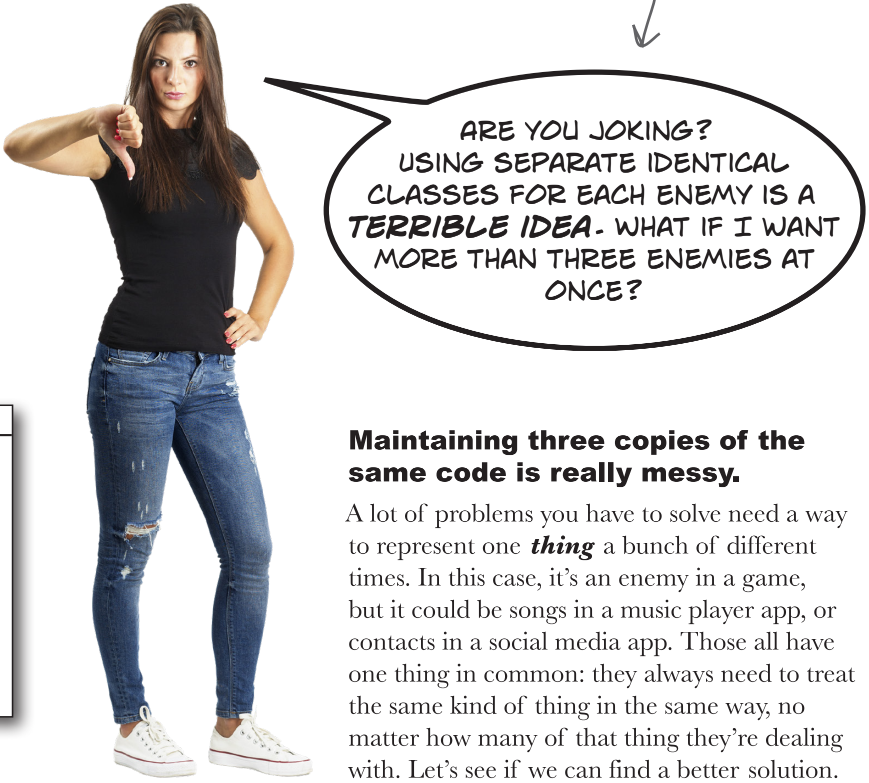
And that's great…until Ana wants more than one enemy, which is all there was in each of her early prototypes. What should she do to add a second or third enemy to her game?

Ana *could* copy the Enemy class code and paste it into two more class files. Then her program could use methods to control three different enemies at once. Technically, we're reusing the code…right?

Hey Ana, what do you think of that idea?

She has a point. What if she wants a level with, say, dozens of zombies? Creating dozens of identical classes just isn't practical.



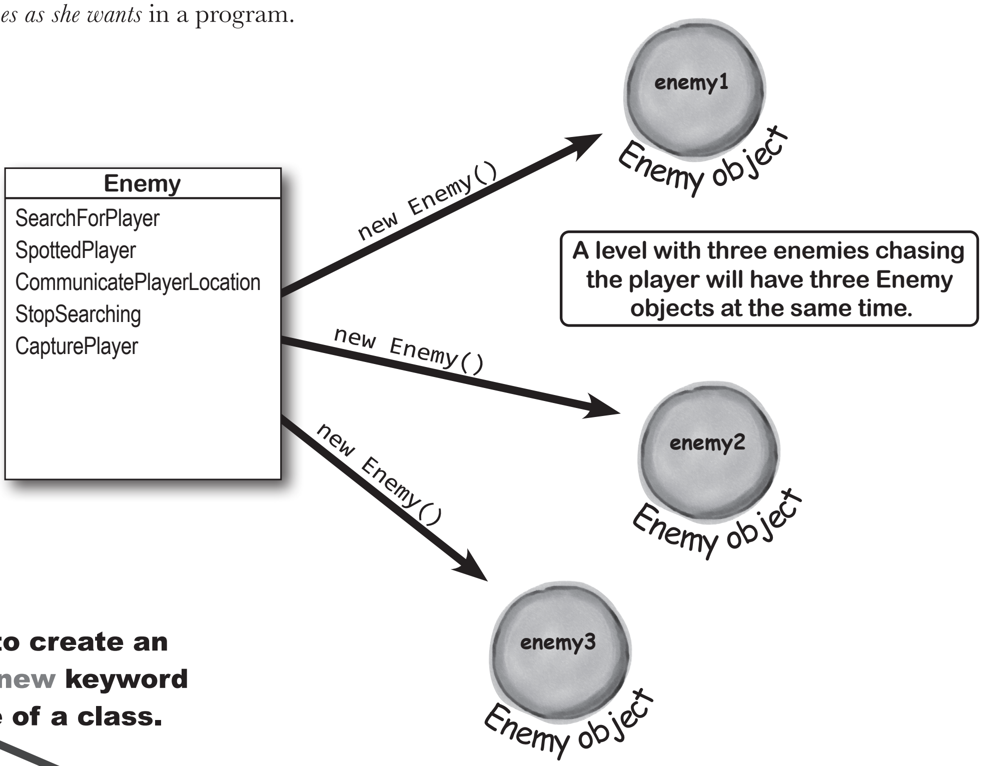

ARE YOU JOKING? USING SEPARATE IDENTICAL CLASSES FOR EACH ENEMY IS A TERRIBLE IDEA. WHAT IF I WANT MORE THAN THREE ENEMIES AT ONCE?

Maintaining three copies of the same code is really messy.

A lot of problems you have to solve need a way to represent one ***thing*** a bunch of different times. In this case, it's an enemy in a game, but it could be songs in a music player app, or contacts in a social media app. Those all have one thing in common: they always need to treat the same kind of thing in the same way, no matter how many of that thing they're dealing with. Let's see if we can find a better solution.

Ana can use objects to solve her problem

Objects are C#'s tool that you use to work with a bunch of similar things. Ana can use objects to program her Enemy class just once, and use it *as many times as she wants* in a program.



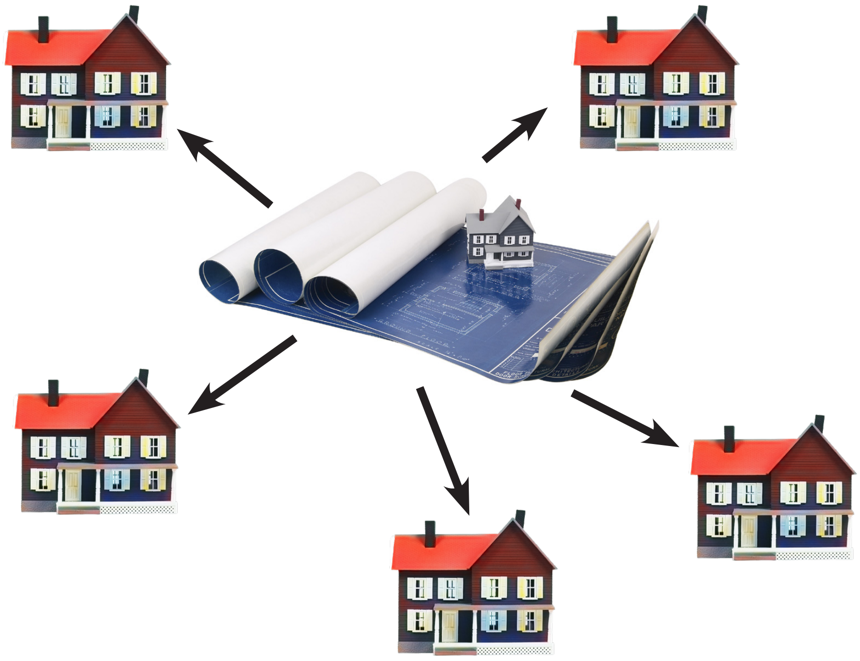
All you need to create an object is the new keyword and the name of a class.

```
Enemy enemy1 = new Enemy();
enemy1.SearchForPlayer();
if (enemy1.SpottedPlayer()) {
   enemy1.CommunicatePlayerLocation();
} else {
   enemy1.StopSearching();
}
```

Now you can use the object! When you create an object from a class, that object has all of the methods from that class.

You use a class to build an object

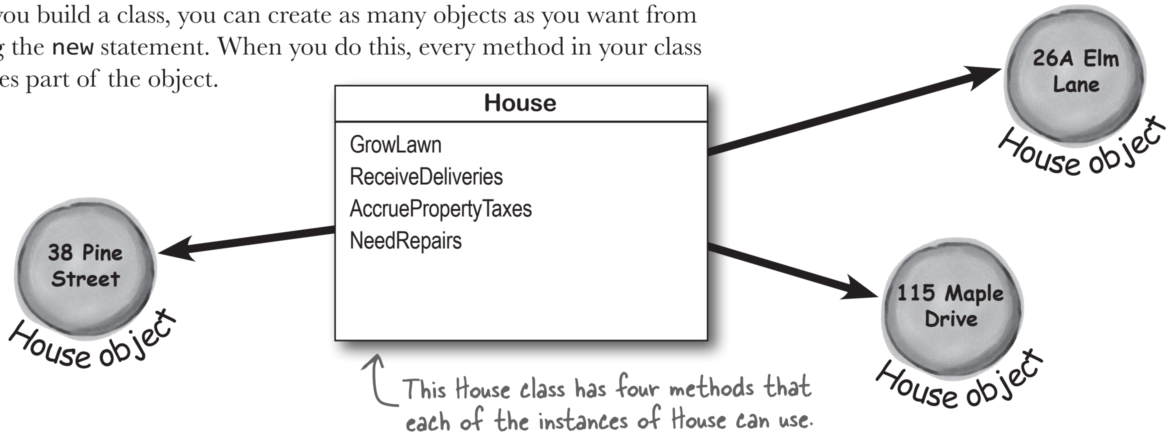
A class is like a blueprint for an object. If you wanted to build five identical houses in a suburban housing development, you wouldn't ask an architect to draw up five identical sets of blueprints. You'd just use one blueprint to build five houses.



A class defines its members, just like a blueprint defines the layout of the house. You can use one blueprint to make any number of houses, and you can use one class to make any number of objects.

An object gets its methods from its class

Once you build a class, you can create as many objects as you want from it using the new statement. When you do this, every method in your class becomes part of the object.



When you create a new object from a class, it's called an instance of that class

You use the **new keyword** to create an object. All you need is a variable to use with it. Use the class as the variable type to declare the variable, so instead of int or bool, you'll use a class like House or Enemy.

in-stance, noun.
an example or one occurrence of something. *The IDE search-and-replace feature finds every* ***instance*** *of a word and changes it to another.*

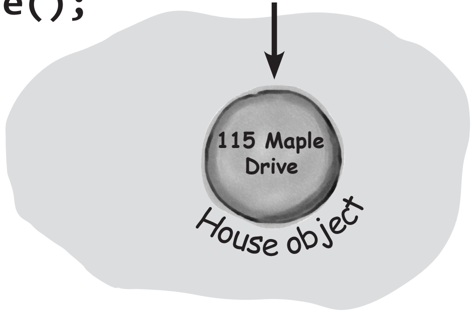
Before: here's a picture of your computer's memory when your program starts.

Your program executes a new statement.

```
House mapleDrive115 = new House();
```

This new statement creates a new House object and assigns it to a variable called mapleDrive115.

After: now it has an instance of the House class in memory.



THAT NEW KEYWORD LOOKS FAMILIAR. I'VE SEEN THIS SOMEWHERE BEFORE, HAVEN'T I?

Yes! You've already created instances in your own code.

Go back to your animal matching program and look for this line of code:

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

You created an instance of the Random class, and then you called its Next method. Now look at your CardPicker class and find the **new** statement. You've been using objects this whole time!

A better solution for Ana…brought to you by objects

Ana used objects to reuse the code in the Enemy class without all that messy copying that would've left duplicate code all over her project. Here's how she did it.

1. Ana created a Level class that stored the enemies in an **Enemy array** called `enemies`, just like you used string arrays to store cards and animal emoji.

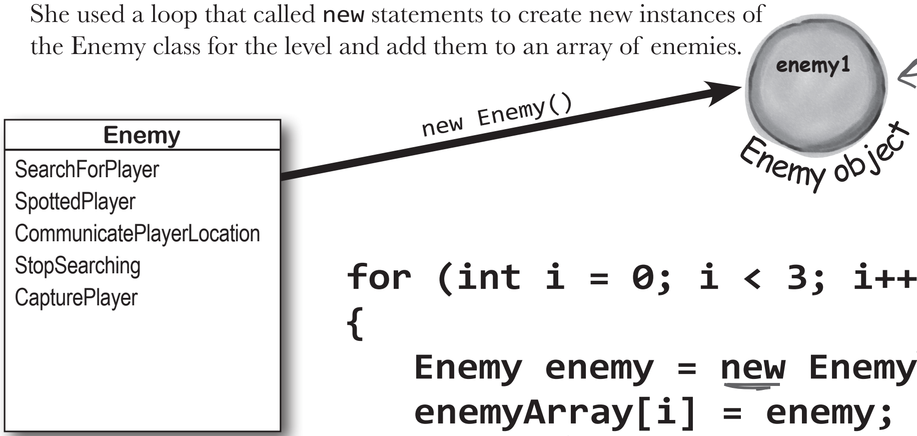
```
public class Level {
    Enemy[] enemyArray = new Enemy[3];
```

Hmm, this array is inside the class, but outside of the methods. What do you think is going on?

Use the name of a class to declare an array of instances of that class.

We're using the new keyword to create an array of Enemy objects, just like you did earlier with strings.

2. She used a loop that called `new` statements to create new instances of the Enemy class for the level and add them to an array of enemies.



The enemy1 object is an instance of the Enemy class.

```
for (int i = 0; i < 3; i++)
{
    Enemy enemy = new Enemy();
    enemyArray[i] = enemy;
}
```

This statement uses the new keyword to create an Enemy object.

This statement adds the newly created Enemy object to the array.

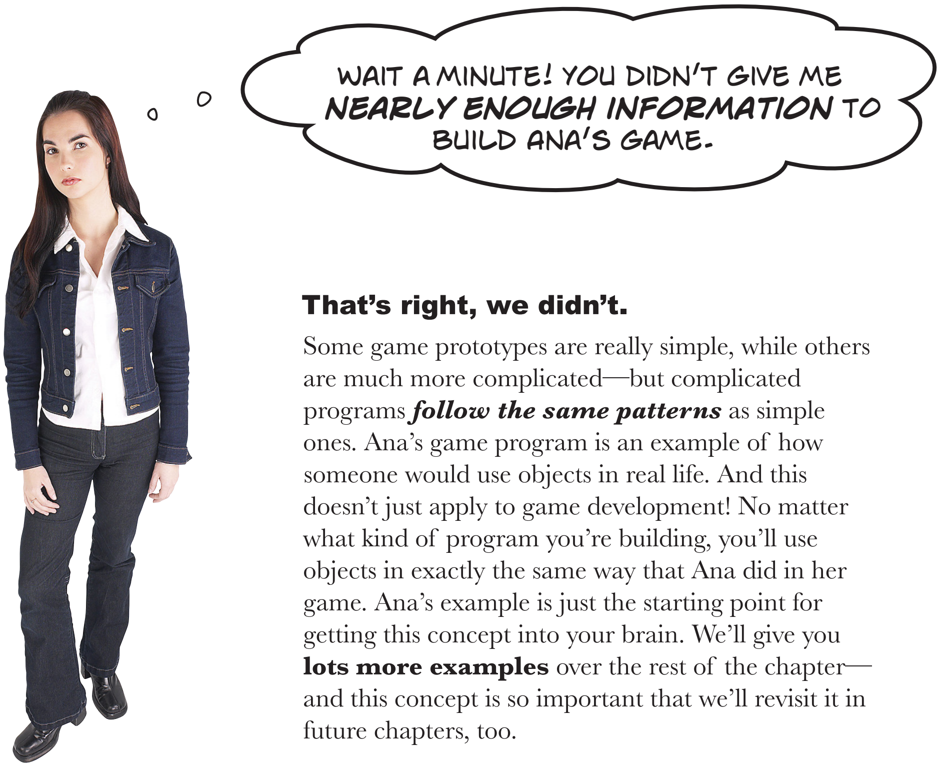
3. She called methods of each Enemy instance during every frame update to implement the enemy behavior.

enemy1
Enemy object
enemy2
Enemy object
enemy3
Enemy object

```
foreach (Enemy enemy in enemyArray)
{
    // code that calls the Enemy methods
}
```

The foreach loop iterates through the array of Enemy objects.

When you create a new instance of a class, it's called instantiating that class.


WAIT A MINUTE! YOU DIDN'T GIVE ME **NEARLY ENOUGH INFORMATION** TO BUILD ANA'S GAME.

That's right, we didn't.

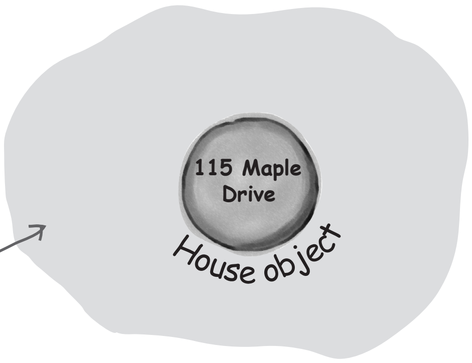
Some game prototypes are really simple, while others are much more complicated—but complicated programs ***follow the same patterns*** as simple ones. Ana's game program is an example of how someone would use objects in real life. And this doesn't just apply to game development! No matter what kind of program you're building, you'll use objects in exactly the same way that Ana did in her game. Ana's example is just the starting point for getting this concept into your brain. We'll give you **lots more examples** over the rest of the chapter—and this concept is so important that we'll revisit it in future chapters, too.

Theory and practice

Speaking of patterns, here's a pattern that you'll see over and over again throughout the book. We'll introduce a concept or idea (like objects) over the course of a few pages, using pictures and short code excerpts to demonstrate the idea. This is your opportunity to take a step back and try to understand what's going on without having to worry about getting a program to work.

```
House mapleDrive115 = new House();
```

When we're introducing a new concept (like objects), keep your eyes open for pictures and code excerpts like these.



Sharpen your pencil

Now that you've got a better idea of how objects work, it's a great time to go back to your CardPicker class and get to know the Random class that you're using.

1. Put your cursor inside of any of the methods, press Enter to start a new statement, then type **`random.`**—as soon as you type the period, Visual Studio will pop up an IntelliSense window that shows its methods. Each method is marked with a cube icon (). We filled in some of the methods. Finish filling in the class diagram for the Random class.

Random
Equals
GetHashCode
GetType
....
....
....
ToString

2. Write code to create a new array of doubles called **`randomDoubles`**, then use a `for` loop to add 20 double values to that array. You should only add random floating-point numbers that are greater than or equal to 0.0, and less than 1.0. Use the IntelliSense pop-up to help you choose the right method from the Random class to use in your code.

```
Random random = 
.................................................................
double[] randomDoubles = new double[20];
.................................................................

.................................................................
{
.................................................................
   double value = 
.................................................................

.................................................................
}
.................................................................
```

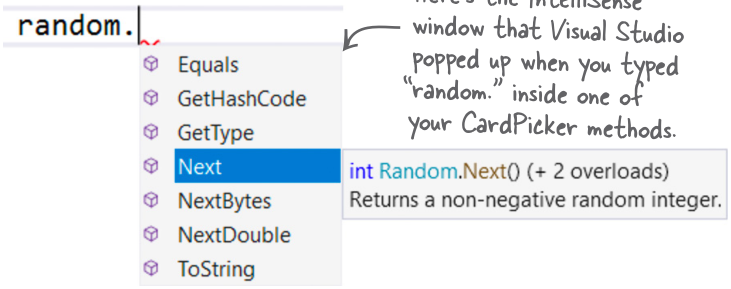
We filled in part of the code, including the curly braces. Your job is to finish those statements and then write the rest of the code.

Sharpen your pencil Solution

Now that you've got a better idea of how objects work, it's a great time to go back to your CardPicker class and get to know the Random class that you're using.

1. Put your cursor inside of any of the methods, press Enter to start a new statement, then type **`random.`**—as soon as you type the period, Visual Studio will pop up an IntelliSense window that shows its methods. Each method is marked with a cube icon (⬡). We filled in some of the methods. Finish filling in the class diagram for the Random class.

Random
Equals
GetHashCode
GetType
Next
NextBytes
NextDouble
ToString



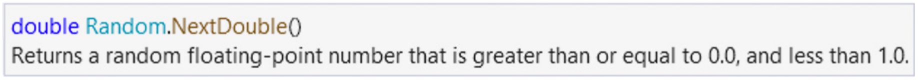
```
random.
```

- Equals
- GetHashCode
- GetType
- Next
- NextBytes
- NextDouble
- ToString

int Random.Next() (+ 2 overloads)
Returns a non-negative random integer.

Here's the IntelliSense window that Visual Studio popped up when you typed "random." inside one of your CardPicker methods.

When you select NextDouble in the IntelliSense window, it shows documentation for the method.



double Random.NextDouble()
Returns a random floating-point number that is greater than or equal to 0.0, and less than 1.0.

2. Write code to create a new array of doubles called **`randomDoubles`**, then use a `for` loop to add 20 double values to that array. You should only add random floating-point numbers that are greater than or equal to 0.0, and less than 1.0. Use the IntelliSense pop-up to help you choose the right method from the Random class to use in your code.

```
Random random = new Random();
double[] randomDoubles = new double[20];
for (int i = 0; i < 20; i++)
{
    double value = random.NextDouble();
    randomDoubles[i] = value;
}
```

This is really similar to the code that you used in your CardPicker class.

An instance uses fields to keep track of things

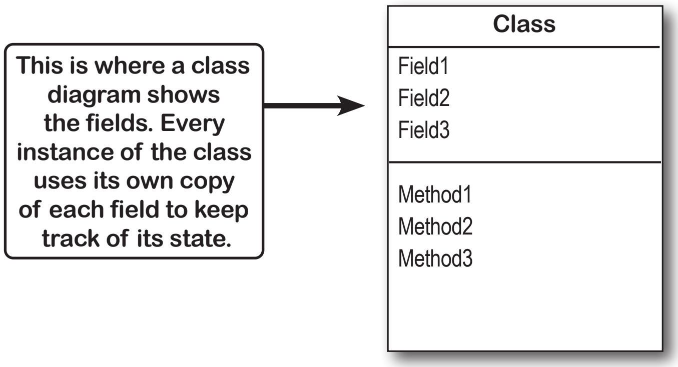
You've seen how classes can contain fields as well as methods. We just saw how you used the `static` keyword to declare a field in your CardPicker class:

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

What happens if you take away that `static` keyword? Then the field becomes an **instance field**, and every time you instantiate the class the new instance that was created *gets its own copy* of that field.

Sometimes people think the word "instantiate" sounds a little weird, but it makes sense when you think about what it means.

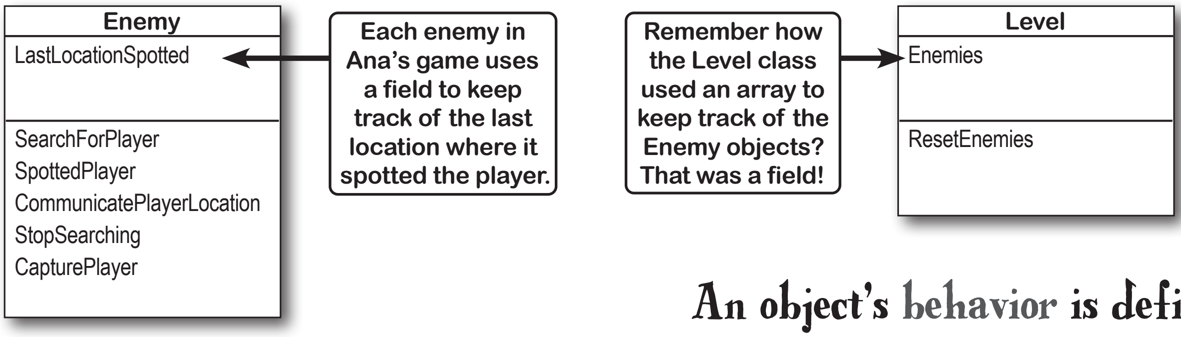
When we want to include fields a class diagram, we'll draw a horizontal line in the box. The fields go above the line, and methods go below the line.


This is where a class diagram shows the fields. Every instance of the class uses its own copy of each field to keep track of its state.
Class
Field1
Field2
Field3
Method1
Method2
Method3

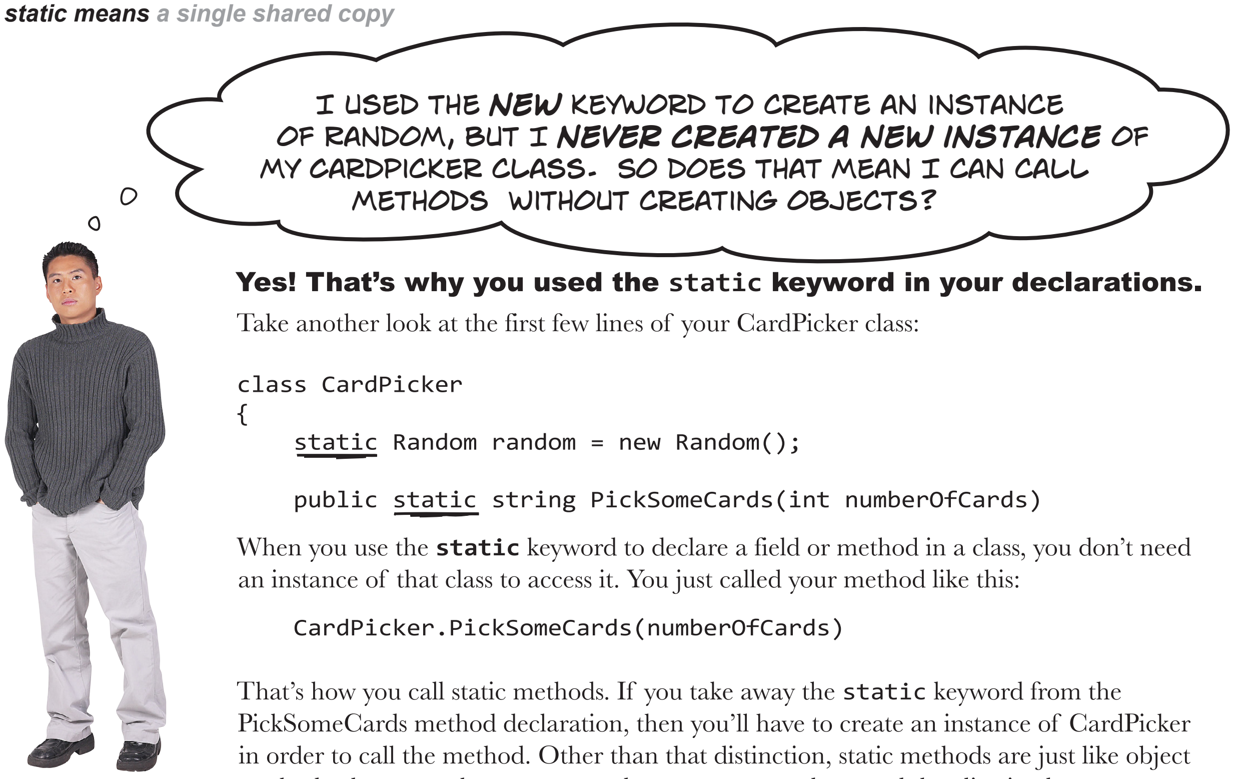
Class diagrams typically list all of the fields and methods in the class. We call them the class members.

Methods are what an object does. Fields are what the object knows.

When Ana's prototype created three instances of her Enemy class, each of those objects was used to keep track of a different enemy in the game. Every instance keeps separate copies of the same data: setting a field on the enemy2 instance won't have any effect on the enemy1 or enemy3 instances.


Enemy
LastLocationSpotted
SearchForPlayer
SpottedPlayer
CommunicatePlayerLocation
StopSearching
CapturePlayer
Each enemy in Ana's game uses a field to keep track of the last location where it spotted the player.
Remember how the Level class used an array to keep track of the Enemy objects? That was a field!
Level
Enemies
ResetEnemies

An object's behavior is defined by its methods, and it uses fields to keep track of its state.


I USED THE ***NEW*** KEYWORD TO CREATE AN INSTANCE OF RANDOM, BUT I ***NEVER CREATED A NEW INSTANCE*** OF MY CARDPICKER CLASS. SO DOES THAT MEAN I CAN CALL METHODS WITHOUT CREATING OBJECTS?

Yes! That's why you used the `static` keyword in your declarations.

Take another look at the first few lines of your CardPicker class:

```
class CardPicker
{
    static Random random = new Random();

    public static string PickSomeCards(int numberOfCards)
```

When you use the **`static`** keyword to declare a field or method in a class, you don't need an instance of that class to access it. You just called your method like this:

```
    CardPicker.PickSomeCards(numberOfCards)
```

That's how you call static methods. If you take away the `static` keyword from the PickSomeCards method declaration, then you'll have to create an instance of CardPicker in order to call the method. Other than that distinction, static methods are just like object methods: they can take arguments, they can return values, and they live in classes.

When a field is static **there's only one copy of it, and it's shared by all instances**. So if you created multiple instances of CardPicker, they would all share the same *random* field. You can even mark your **whole class** as static, and then all of its members **must** be static too. If you try to add a nonstatic method to a static class, your program won't build.

there are no Dumb Questions

Q: When I think of something that's "static" I think of something that doesn't change. Does that mean nonstatic methods can change, but static methods don't? Do they behave differently?

A: No, both static and nonstatic methods act exactly the same. The only difference is that static methods don't require an instance, while nonstatic methods do.

Q: So I can't use my class until I create an instance of an object?

A: You can use its static methods, but if you have methods that aren't static, then you need an instance before you can use them.

Q: Then why would I want a method that needs an instance? Why wouldn't I make all my methods static?

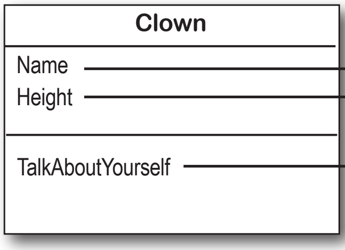
A: Because if you have an object that's keeping track of certain data—like Ana's instances of her Enemy class that each kept track of different enemies in her game—then you can use each instance's methods to work with that data. So when Ana's game calls the StopSearching method on the enemy2 instance, it only causes that one enemy to stop searching for the player. It doesn't affect the enemy1 or enemy3 objects, and they can keep searching. That's how Ana can create game prototypes with any number of enemies, and her programs can keep track of all of them at once.

When a field is static, there's only one copy of it shared by all instances.

Sharpen your pencil

Here's a .NET console app that writes several lines to the console. It includes a class called Clown that has two fields, Name and Height, and a method called TalkAboutYourself. Your job is to read the code and write down the lines that are printed to the console.

Here's the class diagram and code for the Clown class:



```
class Clown {
  public string Name;
  public int Height;

  public void TalkAboutYourself() {
    Console.WriteLine("My name is " + Name +
       " and I'm " + Height + " inches tall.");
  }
}
```

Here's the Main method for the console app. There are comments next to each of the calls to the TalkAboutYourself method, which prints a line to the console. Your job is to fill in the blanks in the comments so they match the output.

```
static void Main(string[] args) {
    Clown oneClown = new Clown();
    oneClown.Name = "Boffo";
    oneClown.Height = 14;
    oneClown.TalkAboutYourself();      // My name is _______ and I'm ____ inches tall."

    Clown anotherClown = new Clown();
    anotherClown.Name = "Biff";
    anotherClown.Height = 16;
    anotherClown.TalkAboutYourself();  // My name is _______ and I'm ____ inches tall."

    Clown clown3 = new Clown();
    clown3.Name = anotherClown.Name;
    clown3.Height = oneClown.Height - 3;
    clown3.TalkAboutYourself();        // My name is _______ and I'm ____ inches tall."

    anotherClown.Height *= 2;
    anotherClown.TalkAboutYourself();  // My name is _______ and I'm ____ inches tall."
}
```

The *= operator tells C# to take whatever's on the left of the operator and multiply it by whatever's on the right, so this will update the Height field.

Thanks for the memory

When your program creates an object, it lives in a part of the computer's memory called the **heap**. When your code creates an object with a `new` statement, C# immediately reserves space in the heap so it can store the data for that object.

Here's a picture of the heap before the project starts. Notice that it's empty.

When your program creates a new object, it gets added to the heap.

Sharpen your pencil Solution

Here's what the program prints to the console. It's worth taking a few minutes to create a new .NET console app, add the Clown class, and make its Main method match this one, then step through it with the debugger.

```
static void Main(string[] args) {
    Clown oneClown = new Clown();
    oneClown.Name = "Boffo";
    oneClown.Height = 14;
    oneClown.TalkAboutYourself();      // My name is _Boffo_ and I'm _14_ inches tall."

    Clown anotherClown = new Clown();
    anotherClown.Name = "Biff";
    anotherClown.Height = 16;
    anotherClown.TalkAboutYourself();  // My name is _Biff_ and I'm _16_ inches tall."

    Clown clown3 = new Clown();
    clown3.Name = anotherClown.Name;
    clown3.Height = oneClown.Height - 3;
    clown3.TalkAboutYourself();        // My name is _Biff_ and I'm _11_ inches tall."

    anotherClown.Height *= 2;
    anotherClown.TalkAboutYourself();  // My name is _Biff_ and I'm _32_ inches tall."
}
```

When you step through this method in the debugger, you should see the value of the Height field gets set to 14 after this line is executed.

This line uses the Height field of the old oneClown instance to set the Height field of the new clown3 instance.

What's on your program's mind

Let's take a closer look at the program in the "Sharpen your pencil" exercise, starting with the first line of the Main method. It's actually **two statements** combined into one:

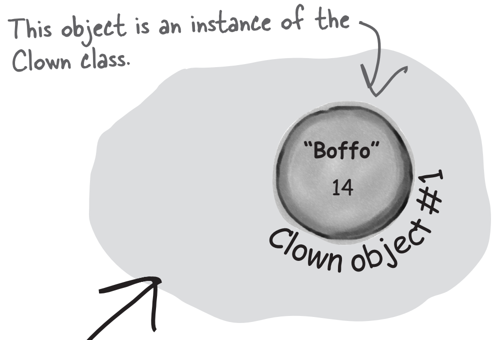
```
Clown oneClown = new Clown();
```

This is a statement that declares a variable called oneClown of type Clown.

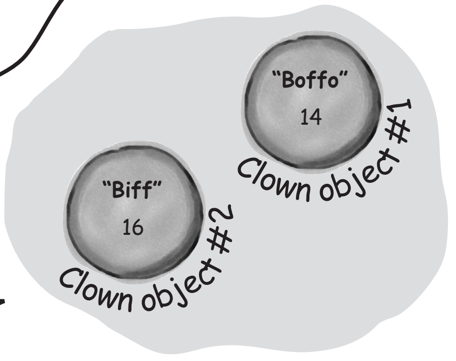
This statement creates a new object and assigns it to the oneClown variable.

Next, let's look closely at what the heap looks like after each group of statements is executed:

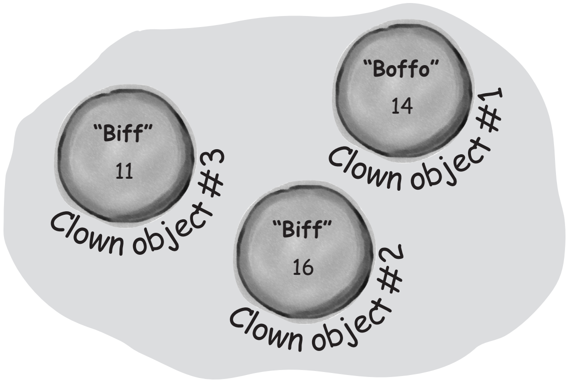
```
// These statements create an instance of
// Clown and then set its fields
Clown oneClown = new Clown();
oneClown.Name = "Boffo";
oneClown.Height = 14;
oneClown.TalkAboutYourself();
```



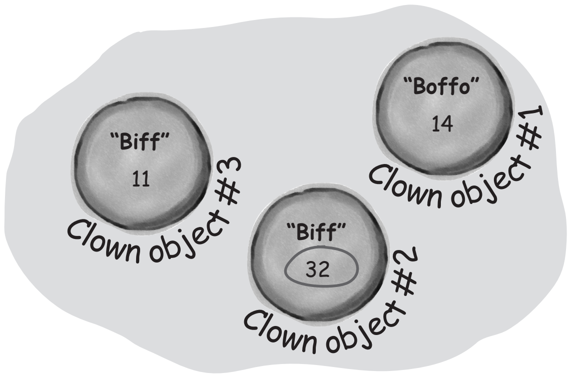
```
// These statements instantiate a second
// Clown object and fill it with data.
Clown anotherClown = new Clown();
anotherClown.Name = "Biff";
anotherClown.Height = 16;
anotherClown.TalkAboutYourself();
```



```
// Now we instantiate a third Clown object
// and use data from the other two
// instances to set its fields
Clown clown3 = new Clown();
clown3.Name = anotherClown.Name;
clown3.Height = oneClown.Height - 3;
clown3.TalkAboutYourself();
```



```
// Notice how there's no "new" statement
// here -- we're not creating a new object,
// just modifying one already in memory
anotherClown.Height *= 2;
anotherClown.TalkAboutYourself();
```



Sometimes code can be difficult to read

You may not realize it, but you're constantly making choices about how to structure your code. Do you use one method to do something? Do you split it into more than one? Do you even need a new method at all? The choices you make about methods can make your code much more intuitive—or if you're not careful, much more convoluted.

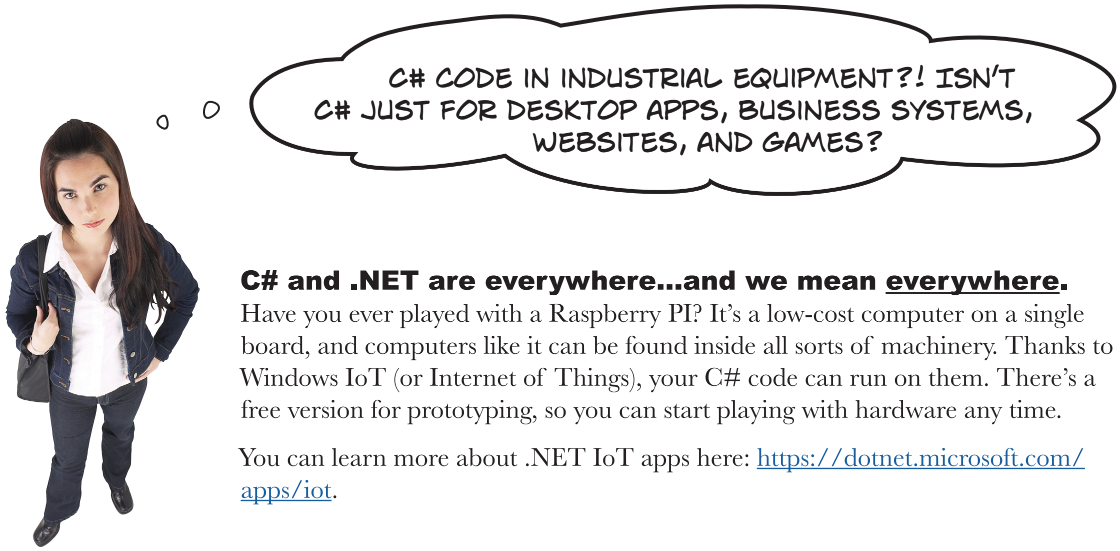
Here's a nice, compact chunk of code from a control program that runs a machine that makes candy bars:

```
int t = m.chkTemp();
if (t > 160) {
   T tb = new T();
   tb.clsTrpV(2);
   ics.Fill();
   ics.Vent();
   m.airsyschk();
}
```

Extremely compact code can be especially problematic

Take a second and look at that code. Can you figure out what it does? Don't feel bad if you can't—it's very difficult to read! Here are a few reasons why:

- We can see a few variable names: `tb`, `ics`, `m`. These are terrible names! We have no idea what they do. And what's that T class for?
- The chkTemp method returns an integer…but what does it do? We can guess maybe it has something to do with checking the temperature of…something?
- The clsTrpV method has one parameter. Do we know what that parameter is supposed to be? Why is it 2? What is that 160 number for?


C# CODE IN INDUSTRIAL EQUIPMENT?! ISN'T C# JUST FOR DESKTOP APPS, BUSINESS SYSTEMS, WEBSITES, AND GAMES?

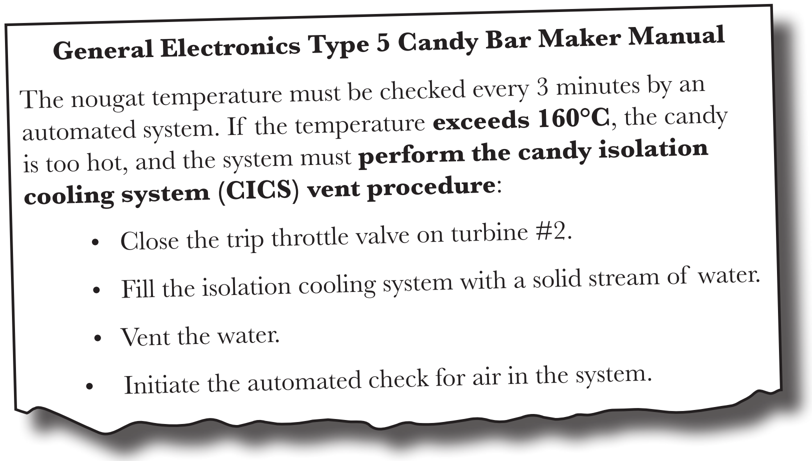
C# and .NET are everywhere...and we mean <u>everywhere</u>. Have you ever played with a Raspberry PI? It's a low-cost computer on a single board, and computers like it can be found inside all sorts of machinery. Thanks to Windows IoT (or Internet of Things), your C# code can run on them. There's a free version for prototyping, so you can start playing with hardware any time.

You can learn more about .NET IoT apps here: https://dotnet.microsoft.com/apps/iot.

Most code doesn't come with a manual

Those statements don't give you any hints about why the code's doing what it's doing. In this case, the programmer was happy with the results because she was able to get it all into one method. But making your code as compact as possible isn't really useful! Let's break it up into methods to make it easier to read, and make sure the classes are given names that make sense.

We'll start by figuring out what the code is supposed to do. Luckily, we happen to know that this code is part of an **embedded system**, or a controller that's part of a larger electrical or mechanical system. And we happen to have documentation for this code—specifically, the manual that the programmers used when they originally built the system.



General Electronics Type 5 Candy Bar Maker Manual

The nougat temperature must be checked every 3 minutes by an automated system. If the temperature **exceeds 160°C**, the candy is too hot, and the system must **perform the candy isolation cooling system (CICS) vent procedure**:

- Close the trip throttle valve on turbine #2.
- Fill the isolation cooling system with a solid stream of water.
- Vent the water.
- Initiate the automated check for air in the system.

How do you figure out what your code is supposed to do? Well, all code is written for a reason. So it's up to you to figure out that reason! In this case, we got lucky—we could look up the page in the manual that the developer followed.

We can compare the code with the manual that tells us what the code is supposed to do. Adding comments can definitely help us understand what it's supposed to do:

```
/* This code runs every 3 minutes to check the temperature.
 * If it exceeds 160C we need to vent the cooling system.
 */
int t = m.chkTemp();
if (t > 160) {
   // Get the controller system for the turbines
   T tb = new T();

   // Close throttle valve on turbine #2
   tb.clsTrpV(2);

   // Fill and vent the isolation cooling system
   ics.Fill();
   ics.Vent();

   // Initiate the air system check
   m.airsyschk();
}
```

Adding extra line breaks to your code in some places can make it easier to read.

BRAIN POWER

Code comments are a good start. Can you think of a way to make this code even easier to understand?

Use intuitive class and method names

That page from the manual made it a lot easier to understand the code. It also gave us some great hints about how to make our code easier to understand. Let's take a look at the first two lines:

```
/* This code runs every 3 minutes to check the temperature.
 * If it exceeds 160C we need to vent the cooling system.
 */
int t = m.chkTemp();
if (t > 160) {
```

The comment we added explains a lot. Now we know why the conditional test checks the variable **t** against 160—the manual says that any temperature above 160°C means the nougat is too hot. It turns out that **m** is a class that controls the candy maker, with static methods to check the nougat temperature and check the air system.

So let's put the temperature check into a method, and choose names for the class and the methods that make their purpose obvious. We'll move these first two lines into their own method that returns a Boolean value, true if the nougat is too hot or false if it's OK:

```
/// <summary>
/// If the nougat temperature exceeds 160C it's too hot.
/// </summary>
public bool IsNougatTooHot() {
   int temp = CandyBarMaker.CheckNougatTemperature();
   if (temp > 160) {
     return true;
   } else {
     return false;
   }
}
```

When we rename the class "CandyBarMaker" and the method "CheckNougatTemperature" it starts to make the code easier to understand.

Notice how the C in CandyBarMaker is uppercase? If we always start class names with an uppercase letter and variables with lowercase ones, it's easier to tell when you're calling a static method versus using an instance.

Did you notice the special /// comments above the method? That's called an *XML Documentation Comment*. The IDE uses those comments to show you documentation for methods—like the documentation you saw when you used the IntelliSense window to figure out which method from the Random class to use.

IDE Tip: XML documentation for methods and fields

Visual Studio helps you add XML documentation. Put your cursor in the line above any method and type three slashes, and it will add an empty template for your documentation. If your method has parameters and a return type, it will add **<param>** and **<returns>** tags for them as well. Try going back to your CardPicker class and typing **///** in the line above the PickSomeCards method—the IDE will add blank XML documentation. Fill it in and watch it show up in IntelliSense.

```
/// <summary>
/// Picks a number of cards and returns them.
/// </summary>
/// <param name="numberOfCards">The number of cards to pick.</param>
/// <returns>An array of strings that contain the card names.</returns>
```

You can create XML documentation for your fields, too. Try it out by going to the line just above any field and typing three slashes in the IDE. Anything you put after **<summary>** will show up in the IntelliSense window for the field.

What does the manual say to do if the nougat is too hot? It tells us to perform the candy isolation cooling system (or CICS) vent procedure. So let's make another method, and choose an obvious name for the T class (which turns out to control the turbine) and the ics class (which controls the isolation cooling system, and has two static methods to fill and vent the system), and cap it all off with some brief XML documentation:

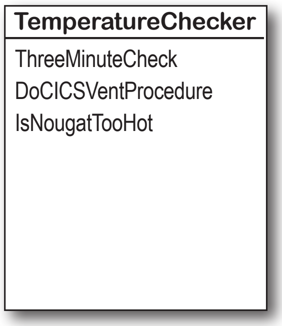
```
/// <summary>
/// Perform the Candy Isolation Cooling System (CICS) vent procedure.
/// </summary>
public void DoCICSVentProcedure() {
   TurbineController turbines = new TurbineController();
   turbines.CloseTripValve(2);
   IsolationCoolingSystem.Fill();
   IsolationCoolingSystem.Vent();
   Maker.CheckAirSystem();
}
```

When your method is declared with a void return type, that means it doesn't return a value and it doesn't need a `return` statement. All of the methods you wrote in the last chapter used the `void` keyword!

Now that we have the IsNougatTooHot and DoCICSVentProcedure methods, we can ***rewrite the original confusing code as a single method***—and we can give it a name that makes clear exactly what it does:

```
/// <summary>
/// This code runs every 3 minutes to check the temperature.
/// If it exceeds 160C we need to vent the cooling system.
/// </summary>
public void ThreeMinuteCheck() {
   if (IsNougatTooHot() == true) {
      DoCICSVentProcedure();
   }
}
```

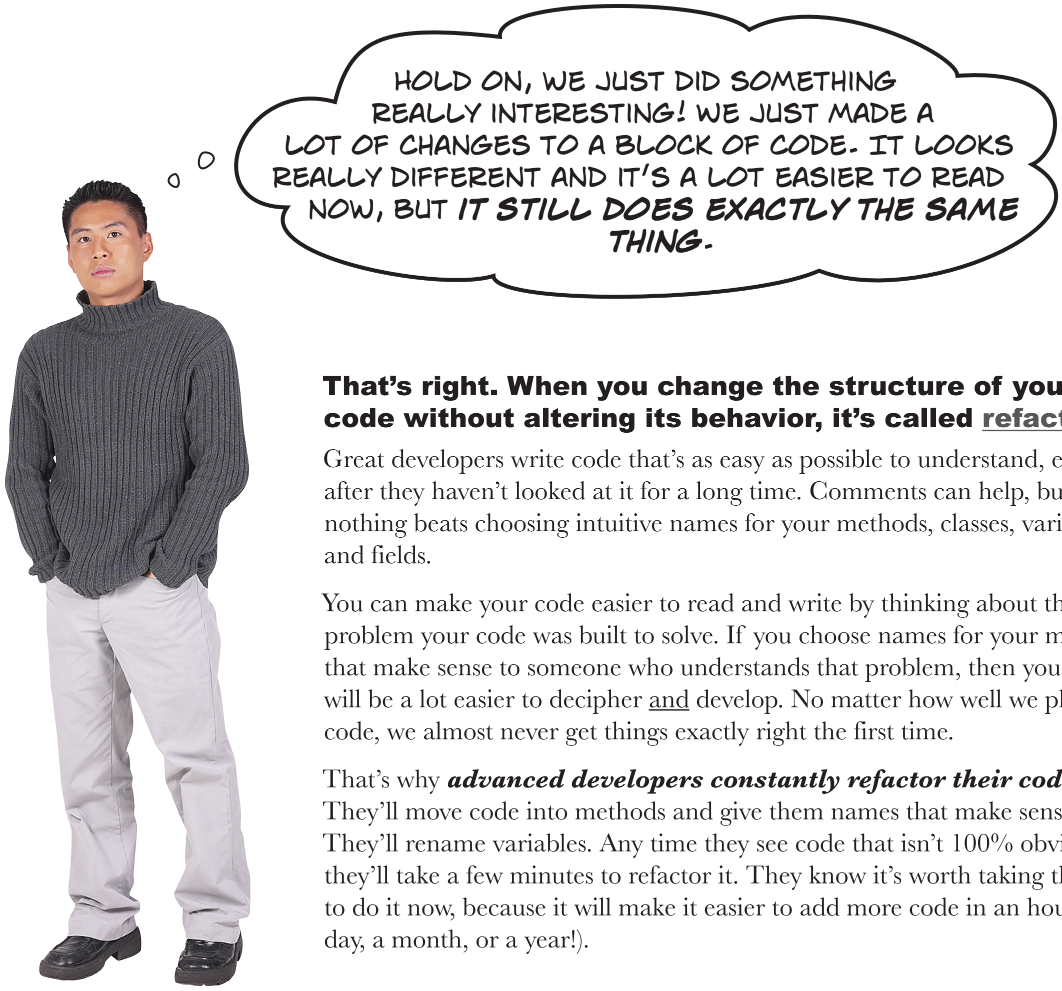
We bundled these new methods into a class called TemperatureChecker. Here's its class diagram.



Now the code is a lot more intuitive! Even if you don't know that the CICS vent procedure needs to be run if the nougat is too hot, **it's a lot more obvious what this code is doing**.

Use class diagrams to plan out your classes

A class diagram is valuable tool for designing your code BEFORE you start writing it. Write the name of the class at the top of the diagram. Then write each method in the box at the bottom. Now you can see all of the parts of the class at a glance—and that's your first chance to spot problems that might make your code difficult to use or understand later.


HOLD ON, WE JUST DID SOMETHING REALLY INTERESTING! WE JUST MADE A LOT OF CHANGES TO A BLOCK OF CODE. IT LOOKS REALLY DIFFERENT AND IT'S A LOT EASIER TO READ NOW, BUT *IT STILL DOES EXACTLY THE SAME THING.*

That's right. When you change the structure of your code without altering its behavior, it's called refactoring.

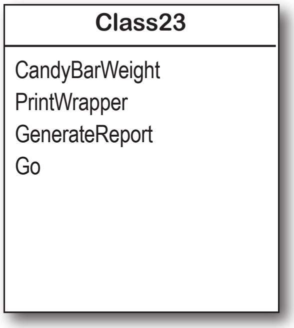
Great developers write code that's as easy as possible to understand, even after they haven't looked at it for a long time. Comments can help, but nothing beats choosing intuitive names for your methods, classes, variables, and fields.

You can make your code easier to read and write by thinking about the problem your code was built to solve. If you choose names for your methods that make sense to someone who understands that problem, then your code will be a lot easier to decipher and develop. No matter how well we plan our code, we almost never get things exactly right the first time.

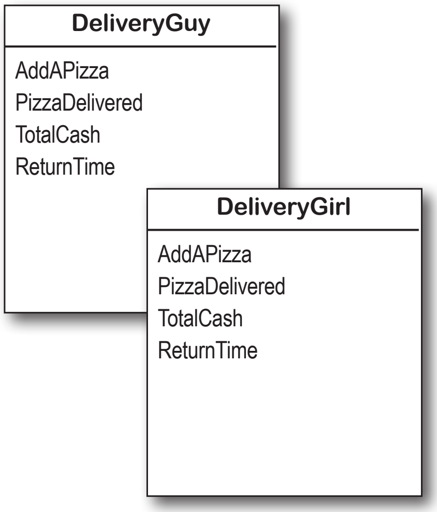
That's why ***advanced developers constantly refactor their code***. They'll move code into methods and give them names that make sense. They'll rename variables. Any time they see code that isn't 100% obvious, they'll take a few minutes to refactor it. They know it's worth taking the time to do it now, because it will make it easier to add more code in an hour (or a day, a month, or a year!).

Sharpen your pencil

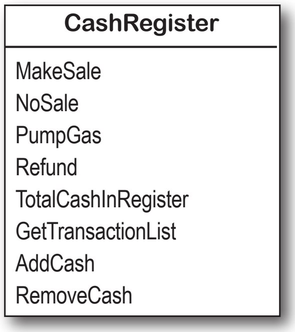
Each of these classes has a serious design flaw. Write down what you think is wrong with each class, and how you'd fix it.



This class is part of the candy manufacturing system from earlier.



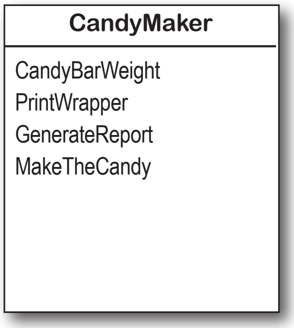
These two classes are part of a system that a pizza parlor uses to track the pizzas that are out for delivery.



The CashRegister class is part of a program that's used by an automated convenience store checkout system.

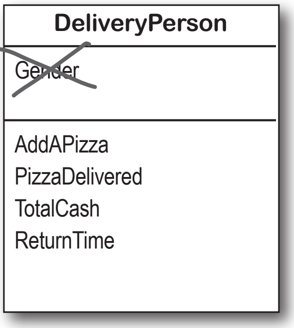
Sharpen your pencil Solution

Here's how we corrected the classes. We show just one possible way to fix the problems—but there are plenty of other ways you could design these classes depending on how they'll be used.



This class is part of the candy manufacturing system from earlier.

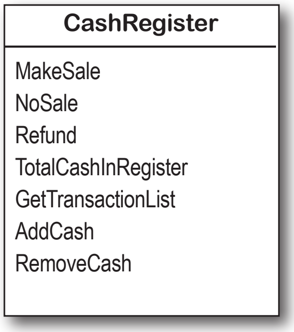
The class name doesn't describe what the class does. A programmer who sees a line of code that calls Class23.Go will have no idea what that line does. We'd also rename the method to something that's more descriptive—we chose MakeTheCandy, but it could be anything.



These two classes are part of a system that a pizza parlor uses to track the pizzas that are out for delivery.

It looks like the DeliveryGuy class and the DeliveryGirl class both do the same thing—they track a delivery person who's out delivering pizzas to customers. A better design would replace them with a single class that adds a field for gender.

We decided NOT to add a Gender field because there's actually no reason for this pizza delivery class to keep track of the gender of the people delivering pizza—and we should respect their privacy! Always look out for ways that bias can sneak into your code.



The CashRegister class is part of a program that's used by an automated convenience store checkout system.

All of the methods in the class do stuff that has to do with a cash register—making a sale, getting a list of transactions, adding cash...except for one: pumping gas. It's a good idea to pull that method out and stick it in another class.

Code Tip: A few ideas for designing intuitive classes

We're about to jump back into writing code. You'll be writing code for the rest of this chapter, and a LOT of code throughout the book. That means you'll be ***creating a lot of classes***. Here are a few things to keep in mind when you make choices about how to design them:

★ **You're building your program to solve a problem.**
Spend some time thinking about that problem. Does it break down into pieces easily? How would you explain that problem to someone else? These are good things to think about when designing your classes.

★ **What real-world things will your program use?**
A program to help a zookeeper track her animals' feeding schedules might have classes for different kinds of food and types of animals.

★ **Use descriptive names for classes and methods.**
Someone should be able to figure out what your classes and methods do just by looking at their names.

★ **Look for similarities between classes.**
Sometimes two classes can be combined into one if they're really similar. The candy manufacturing system might have three or four turbines, but there's only one method for closing the trip valve that takes the turbine number as a parameter.

It's OK if you get stuck when you're writing code. In fact, it's a good thing!

Writing code is all about solving problems—and some of them can be tricky! But if you keep a few things in mind, it'll make the code exercises go more smoothly:

★ It's easy to get caught up in syntax problems, like missing parentheses or quotes. One missing bracket can cause many build errors.

★ It's ***much better*** to look at the solution than to get frustrated with a problem. When you're frustrated, your brain doesn't like to learn.

★ All of the code in this book is tested and definitely works in Visual Studio 2019! But it's easy to accidentally type things wrong (like typing a one instead of a lowercase L).

★ If your solution just won't build, try downloading it from the GitHub repository for the book—it has working code for everything in the book: https://github.com/head-first-csharp/fourth-edition.

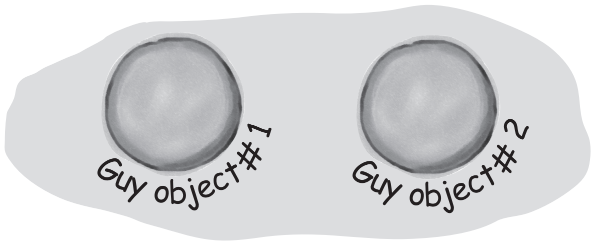
You can learn a lot from reading code. So if you run into a problem with a coding exercise, don't be afraid to peek at the solution. *It's not cheating!*

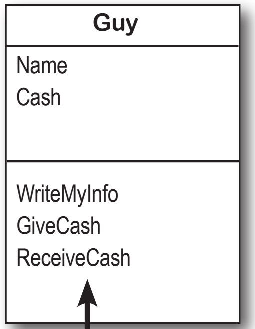
Build a class to work with some guys

Joe and Bob lend each other money all the time. Let's create a class to keep track of how much cash they each have. We'll start with an overview of what we'll build.

1 **We'll create two instances of a "Guy" class.**

We'll use two Guy variables called `joe` and `bob` to keep track of each of our instances. Here's what the heap will look like after they're created:

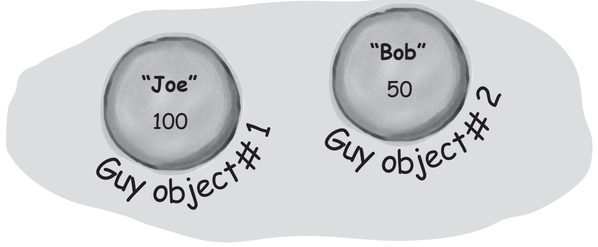




We chose names for the methods that make sense. You call a Guy object's GiveCash method to make him give up some of his cash, and his ReceiveCash method when you want to give cash to him (so he receives it).

2 **We'll set each Guy object's Cash and Name fields.**

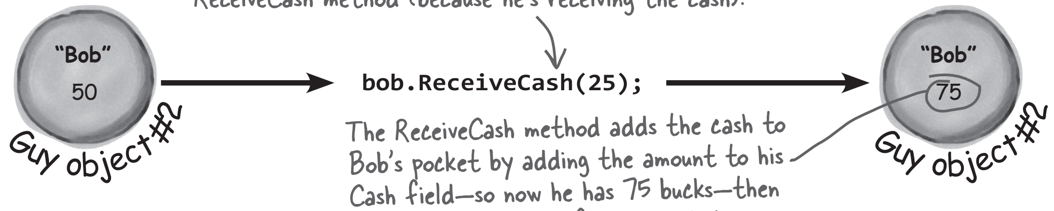
The two objects represent different guys, each with his own name and a different amount of cash in his pocket. Each guy has a Name field that keeps track of his name, and a Cash field that has the number of bucks in his pocket.



3 **We'll add methods to give and receive cash.**

We'll make a guy give cash from his pocket (and reduce his Cash field) by calling his GiveCash method, which will return the amount of cash he gave. We'll make him receive cash and add it to his pocket (increasing his Cash field) by calling his ReceiveCash method, which returns the amount of cash he received.

If we want to give Bob 25 bucks, we call his ReceiveCash method (because he's receiving the cash).



```
bob.ReceiveCash(25);
```

The ReceiveCash method adds the cash to Bob's pocket by adding the amount to his Cash field—so now he has 75 bucks—then returns the number of bucks added.

```
class Guy
{
    public string Name;
    public int Cash;

    /// <summary>
    /// Writes my name and the amount of cash I have to the console.
    /// </summary>
    public void WriteMyInfo()
    {
        Console.WriteLine(Name + " has " + Cash + " bucks.");
    }

    /// <summary>
    /// Gives some of my cash, removing it from my wallet (or printing
    /// a message to the console if I don't have enough cash).
    /// </summary>
    /// <param name="amount">Amount of cash to give.</param>
    /// <returns>
    /// The amount of cash removed from my wallet, or 0 if I don't
    /// have enough cash (or if the amount is invalid).
    /// </returns>
    public int GiveCash(int amount)
    {
        if (amount <= 0)
        {
            Console.WriteLine(Name + " says: " + amount + " isn't a valid amount");
            return 0;
        }
        if (amount > Cash)
        {
            Console.WriteLine(Name + " says: " +
                "I don't have enough cash to give you " + amount);
            return 0;
        }
        Cash -= amount;
        return amount;
    }

    /// <summary>
    /// Receive some cash, adding it to my wallet (or printing
    /// a message to the console if the amount is invalid).
    /// </summary>
    /// <param name="amount">Amount of cash to give.</param>
    public void ReceiveCash(int amount)
    {
        if (amount <= 0)
        {
            Console.WriteLine(Name + " says: " + amount + " isn't an amount I'll take");
        }
        else
        {
            Cash += amount;
        }
    }
}
```

The Name and Cash fields keep track of the guy's name and how much cash he has in his pocket.

Sometimes you want to ask an object to perform a task, like printing a description of itself to the console.

The GiveCash and ReceiveCash methods verify that the amount they're being asked to give or receive is valid. That way you can't ask a guy to receive a negative number, which would cause him to lose cash.

Compare the comments in this code to the class diagrams and illustrations of the Guy objects. If something doesn't make sense at first, take the time to really understand it.

There's an easier way to initialize objects with C#

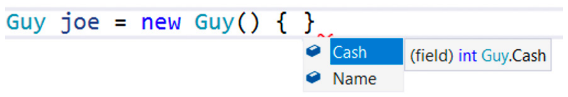
Almost every object that you create needs to be initialized in some way. The Guy object is no exception—it's useless until you set its Name and Cash fields. It's so common to have to initialize fields that C# gives you a shortcut for doing, it called an **object initializer**. The IDE's IntelliSense will help you do it.

You're about to do an exercise where you create two Guy objects. You ***could*** use one `new` statement and two more statements to set its fields:

```
joe = new Guy();
joe.Name = "Joe";
joe.Cash = 50;
```

Instead, type this: **`Guy joe = new Guy() {`**

As soon as you add the left curly bracket, the IDE will pop up an IntelliSense window that shows all of the fields that you can initialize:

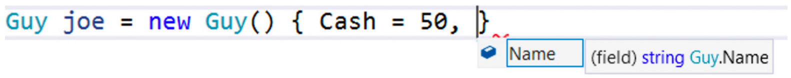


```
Guy joe = new Guy() { }
```

Choose the Name field, set it to 50, and add a comma:

```
Guy joe = new Guy() { Cash = 50,
```

Now type a space—another IntelliSense window will pop up with the remaining field to set:



```
Guy joe = new Guy() { Cash = 50, }
```

Set the Name field and add the semicolon. You now have a single statement that initializes your object:

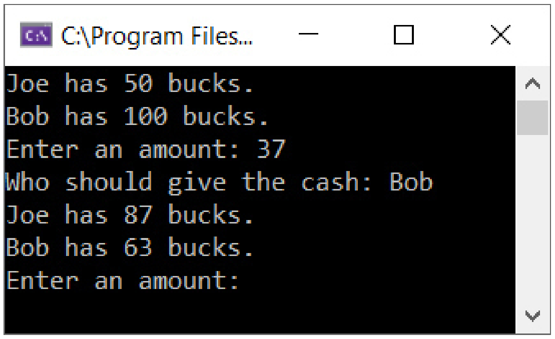
```
Guy joe = new Guy() { Cash = 50, Name = "Joe" };
```

← This new declaration does the same thing as the three lines of code at the top of the page, but it's shorter and easier to read.

Object initializers save you time and make your code more compact and easier to read… and the IDE helps you write them.

Now you have all of the pieces to build your console app that uses two instances of the Guy class. Here's how it will look: ⟶

First, it will call each Guy object's WriteMyInfo method. Then it will read an amount from the input and ask who to give the cash to. It will call one Guy object's GiveCash method, then the other Guy object's ReceiveCash method. It will keep going until the user enters a blank line.



```
Joe has 50 bucks.
Bob has 100 bucks.
Enter an amount: 37
Who should give the cash: Bob
Joe has 87 bucks.
Bob has 63 bucks.
Enter an amount:
```

Exercise

Here's the Main method for a console app that makes Guy objects give cash to each other. Your job is to replace the `comments` with code—read each comment and write code that does exactly what it says. When you're done, you'll have a program that looks like the screenshot on the previous page.

```
static void Main(string[] args)
{
    // Create a new Guy object in a variable called joe
    // Set its Name field to "Joe"
    // Set its Cash field to 50

    // Create a new Guy object in a variable called bob
    // Set its Name field to "Bob"
    // Set its Cash field to 100

    while (true)
    {
        // Call the WriteMyInfo methods for each Guy object

        Console.Write("Enter an amount: ");
        string howMuch = Console.ReadLine();
        if (howMuch == "") return;
        // Use int.TryParse to try to convert the howMuch string to an int
        // if it was successful (just like you did earlier in the chapter)
        {
            Console.Write("Who should give the cash: ");
            string whichGuy = Console.ReadLine();
            if (whichGuy == "Joe")
            {
                // Call the joe object's GiveCash method and save the results
                // Call the bob object's ReceiveCash method with the saved results
            }
            else if (whichGuy == "Bob")
            {
                // Call the bob object's GiveCash method and save the results
                // Call the joe object's ReceiveCash method with the saved results
            }
            else
            {
                Console.WriteLine("Please enter 'Joe' or 'Bob'");
            }
        }
        else
        {
            Console.WriteLine("Please enter an amount (or a blank line to exit).");
        }
    }
}
```

Replace all of the comments with code that does what the comments describe.

Exercise Solution

Here's the Main method for your console app. It uses an infinite loop to keep asking the user how much cash to move between the Guy objects. If the user enters a blank line for an amount, the method executes a `return` statement, which causes Main to exit and the program to end.

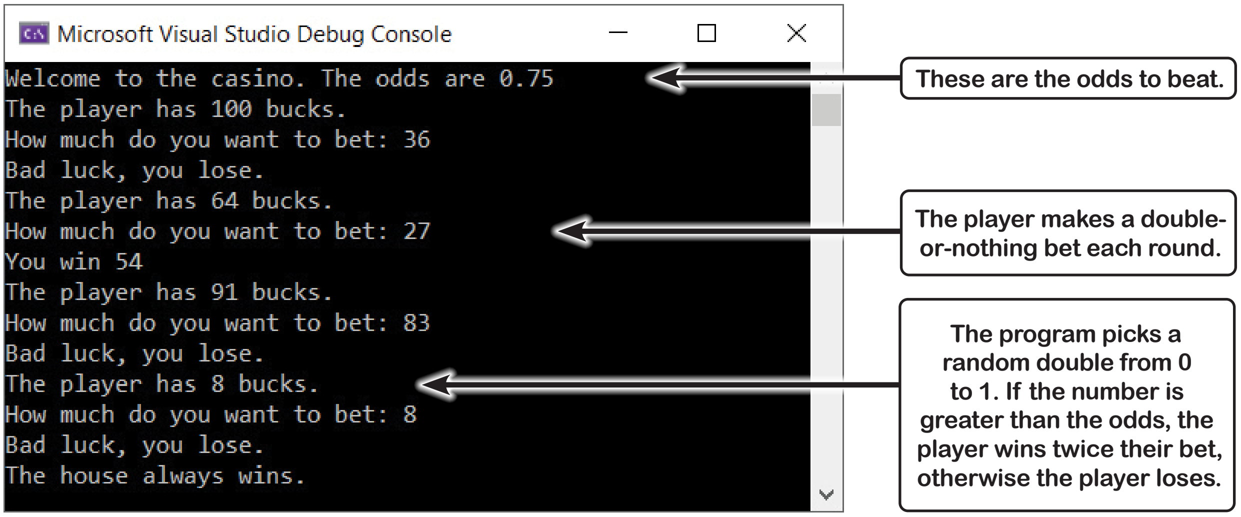
```
static void Main(string[] args)
{
    Guy joe = new Guy() { Cash = 50, Name = "Joe" };
    Guy bob = new Guy() { Cash = 100, Name = "Bob" };

    while (true)
    {
        joe.WriteMyInfo();
        bob.WriteMyInfo();
        Console.Write("Enter an amount: ");
        string howMuch = Console.ReadLine();
        if (howMuch == "") return;
        if (int.TryParse(howMuch, out int amount))
        {
            Console.Write("Who should give the cash: ");
            string whichGuy = Console.ReadLine();
            if (whichGuy == "Joe")
            {
                int cashGiven = joe.GiveCash(amount);
                bob.ReceiveCash(cashGiven);
            }
            else if (whichGuy == "Bob")
            {
                int cashGiven = bob.GiveCash(amount);
                joe.ReceiveCash(cashGiven);
            }
            else
            {
                Console.WriteLine("Please enter 'Joe' or 'Bob'");
            }
        }
        else
        {
            Console.WriteLine("Please enter an amount (or a blank line to exit).");
        }
    }
}
```

When the Main method executes this return statement it ends the program, because console apps stop when the Main method ends.

Here's the code where one Guy object gives cash from his pocket, and the other Guy object receives it.

Don't move on to the next part of the exercise until you have the first part working and you understand what's going on. It's worth taking a few minutes to use the debugger to step through the program and make sure you <u>really</u> get it.

Now that you have your Guy class working, let's see if you can reuse it in a betting game. Look closely at this screenshot to see how it works and what it prints to the console.


Microsoft Visual Studio Debug Console
These are the odds to beat.
The player makes a double-or-nothing bet each round.
The program picks a random double from 0 to 1. If the number is greater than the odds, the player wins twice their bet, otherwise the player loses.

```
Welcome to the casino. The odds are 0.75
The player has 100 bucks.
How much do you want to bet: 36
Bad luck, you lose.
The player has 64 bucks.
How much do you want to bet: 27
You win 54
The player has 91 bucks.
How much do you want to bet: 83
Bad luck, you lose.
The player has 8 bucks.
How much do you want to bet: 8
Bad luck, you lose.
The house always wins.
```

Create a new console app and add the same Guy class. Then, in your Main method, declare three variables: a Random variable called **`random`** with a new instance of the Random class; a double variable called **`odds`** that stores the odds to beat, set to .75; and a Guy variable called **`player`** for an instance of Guy named "The player" with 100 bucks.

Write a line to the console welcoming the player and printing the odds. Then run this loop:

1. Have the Guy object print the amount of cash it has.
2. Ask the user how much money to bet.
3. Read the line into a string variable called `howMuch`.
4. Try to parse it into an int variable called `amount`.
5. If it parses, the player gives the amount to an int variable called `pot`. It gets multiplied by two, because it's a double-or-nothing bet.
6. The program picks a random number between 0 and 1.
7. If the number is greater than `odds`, the player receives the pot.
8. If not, the player loses the amount they bet.
9. The program keeps running while the player has cash.

Sharpen your pencil Bonus question: Is Guy really the best name for this class? Why or why not?

....

....

Exercise Solution

Here's the working Main method for the betting game. Can you think of ways to make it more fun? See if you can figure out how to add additional players, or give different options for odds, or maybe you can think of something more clever. ***This is a chance to get creative!***

...and to get some practice. Getting practice writing code is the best way to become a great developer.

```
static void Main(string[] args)
{
    double odds = .75;
    Random random = new Random();

    Guy player = new Guy() { Cash = 100, Name = "The player" };

    Console.WriteLine("Welcome to the casino. The odds are " + odds);
    while (player.Cash > 0)
    {
        player.WriteMyInfo();
        Console.Write("How much do you want to bet: ");
        string howMuch = Console.ReadLine();
        if (int.TryParse(howMuch, out int amount))
        {
            int pot = player.GiveCash(amount) * 2;
            if (pot > 0)
            {
                if (random.NextDouble() > odds)
                {
                    int winnings = pot;
                    Console.WriteLine("You win " + winnings);
                    player.ReceiveCash(winnings);
                } else
                {
                    Console.WriteLine("Bad luck, you lose.");
                }
            }
        } else
        {
            Console.WriteLine("Please enter a valid number.");
        }

    }
    Console.WriteLine("The house always wins.");
}
```

Was your code a little different? If it still works and produces the right output, that's OK! There are many different ways to write the same program.

...and as you get further along in the book and the exercise solutions get longer, your code will look more and more different from ours. Remember, it's always OK to look at the solution when you're working on an exercise!

Sharpen your pencil Here's our solution to the bonus question—did you come up with a different answer?

When we used Guy to represent Joe and Bob, the name made sense. Now that it's used for a player in a game, a more descriptive class name like Bettor or Player might be more intuitive.

Sharpen your pencil

Here's a .NET console app that writes three lines to the console. Your job is to figure out what it writes, without using a computer. Start at the first line of the Main method and keep track of the values of each of the fields in the objects as it executes.

```
class Pizzazz
{
    public int Zippo;

    public void Bamboo(int eek)
    {
        Zippo += eek;
    }
}

class Abracadabra
{
    public int Vavavoom;

    public bool Lala(int floq)
    {
        if (floq < Vavavoom)
        {
            Vavavoom += floq;
            return true;
        }
        return false;
    }
}

class Program
{
    public static void Main(string[] args)
    {
        Pizzazz foxtrot = new Pizzazz() { Zippo = 2 };
        foxtrot.Bamboo(foxtrot.Zippo);
        Pizzazz november = new Pizzazz() { Zippo = 3 };
        Abracadabra tango = new Abracadabra() { Vavavoom = 4 };
        while (tango.Lala(november.Zippo))
        {
            november.Zippo *= -1;
            november.Bamboo(tango.Vavavoom);
            foxtrot.Bamboo(november.Zippo);
            tango.Vavavoom -= foxtrot.Zippo;
        }
        Console.WriteLine("november.Zippo = " + november.Zippo);
        Console.WriteLine("foxtrot.Zippo = " + foxtrot.Zippo);
        Console.WriteLine("tango.Vavavoom = " + tango.Vavavoom);
    }
}
```

What does this program write to the console?

november.Zippo = ...

foxtrot.Zippo = ...

tango.Vavavoom = ...

To find the solution, enter the program into Visual Studio and run it. If you didn't get the answer right, step through the code line by line and add watches for each of the objects' fields.

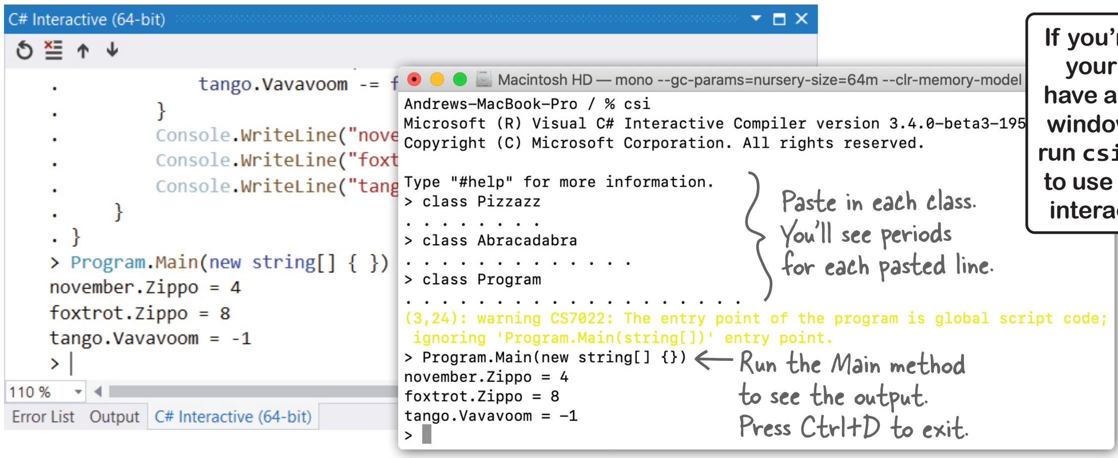
If you don't want to type the whole thing in, you can download it from GitHub: https://github.com/head-first-csharp/fourth-edition.

If you're using a Mac, the IDE generates a class called MainClass, not Program. That won't make a difference in this exercise.

Use the C# Interactive window to run C# code

If you just want to run some C# code, you don't always need to create a new project in Visual Studio. Any C# code entered into the **C# Interactive window** is run immediately. You can open it by choosing View >> Other Windows >> C# Interactive. Try it now, and **paste in the code** from the exercise solution. You can run it by typing this and pressing enter: `Program.Main(new string[] {})`

You're passing an empty array for the "args" parameter.


C# Interactive (64-bit)
110 %
Error List
Output
C# Interactive (64-bit)
Macintosh HD — mono --gc-params=nursery-size=64m --clr-memory-model
Paste in each class. You'll see periods for each pasted line.
Run the Main method to see the output. Press Ctrl+D to exit.

```
.               tango.Vavavoom -= f
.           }
.           Console.WriteLine("nove
.           Console.WriteLine("foxt
.           Console.WriteLine("tang
.       }
. }
> Program.Main(new string[] { })
november.Zippo = 4
foxtrot.Zippo = 8
tango.Vavavoom = -1
> |
```

```
Andrews-MacBook-Pro / % csi
Microsoft (R) Visual C# Interactive Compiler version 3.4.0-beta3-195
Copyright (C) Microsoft Corporation. All rights reserved.

Type "#help" for more information.
> class Pizzazz
. . . . . . . . .
> class Abracadabra
. . . . . . . . . . . . .
> class Program
. . . . . . . . . . . . . . . . . . . .
(3,24): warning CS7022: The entry point of the program is global script code;
 ignoring 'Program.Main(string[])' entry point.
> Program.Main(new string[] {})
november.Zippo = 4
foxtrot.Zippo = 8
tango.Vavavoom = -1
>
```

If you're using a Mac, your IDE may not have a C# Interactive window, but you can run `csi` from Terminal to use the `dotnet` C# interactive compiler.

Don't worry about an error about the entry point.

You can also run an interactive C# session from the command line. On Windows, search the Start menu for **`developer command prompt`**, start it, and then type **`csi`**. On macOS or Linux, run **`csharp`** to start the Mono C# Shell. In both cases, you can paste the Pizzazz, Abracadabra, and Program classes from the previous exercise directly into the prompt, then run `Program.Main(new string[] {})` to run your console app's entry point.

BULLET POINTS

- Use the **`new` keyword** to create instances of a class. A program can have many instances of the same class.
- Each **instance** has all of the methods from the class and gets its own copies of each of the fields.
- When you included `new Random();` in your code, you were creating an **instance of the Random class**.
- Use the **`static` keyword** to declare a field or method in a class as static. You don't need an instance of that class to access static methods or fields.
- When a field is **static**, there's only one copy of it shared by all instances. When you include the `static` keyword in a class declaration, all of its members must be static too.
- If you remove the `static` keyword from a static field, it becomes an **instance field**.
- We refer to fields and methods of a class as its **members**.
- When your program creates an object, it lives in a part of the computer's memory called the **heap**.
- Visual Studio helps you add **XML documentation** to your fields and methods, and displays it in its IntelliSense window.
- **Class diagrams** help you plan out your classes and make them easier to work with.
- When you change the structure of your code without altering its behavior, it's called **refactoring**. Advanced developers constantly refactor their code.
- **Object initializers** save you time and make your code more compact and easier to read.

4 types and references

Getting the reference

THIS DATA JUST GOT *GARBAGE-COLLECTED.*
PUBLIC TELEPHONE
YOU CAN HELP THROW IT HERE
NYCTS

What would your apps be without data? Think about it for a minute. Without data, your programs are…well, it's actually hard to imagine writing code without data. You need **information** from your users, and you use that to look up or produce new information to give back to them. In fact, almost everything you do in programming involves **working with data** in one way or another. In this chapter, you'll learn the ins and outs of C#'s **data types** and **references**, see how to work with data in your program, and even learn a few more things about **objects** (*guess what…objects are data, too!*).

Owen could use our help!

Owen is a game master—a really good one. He hosts a group that meets at his place every week to play different role-playing games (or RPGs), and like any good game master, he really works hard to keep things interesting for the players.

Storytelling, fantasy, and mechanics

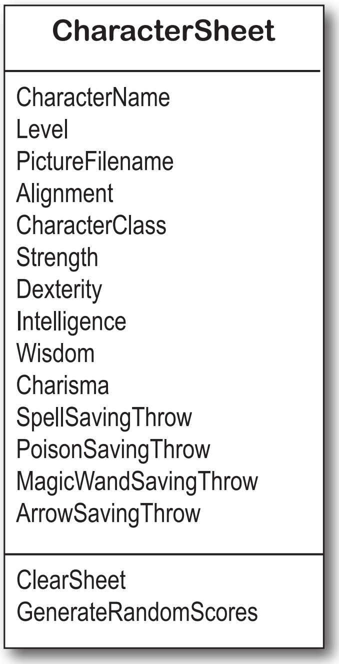
Owen is a particularly good storyteller. Over the last few months he's created an intricate fantasy world for his party, but he's not so happy with the mechanics of the game that they've been playing.

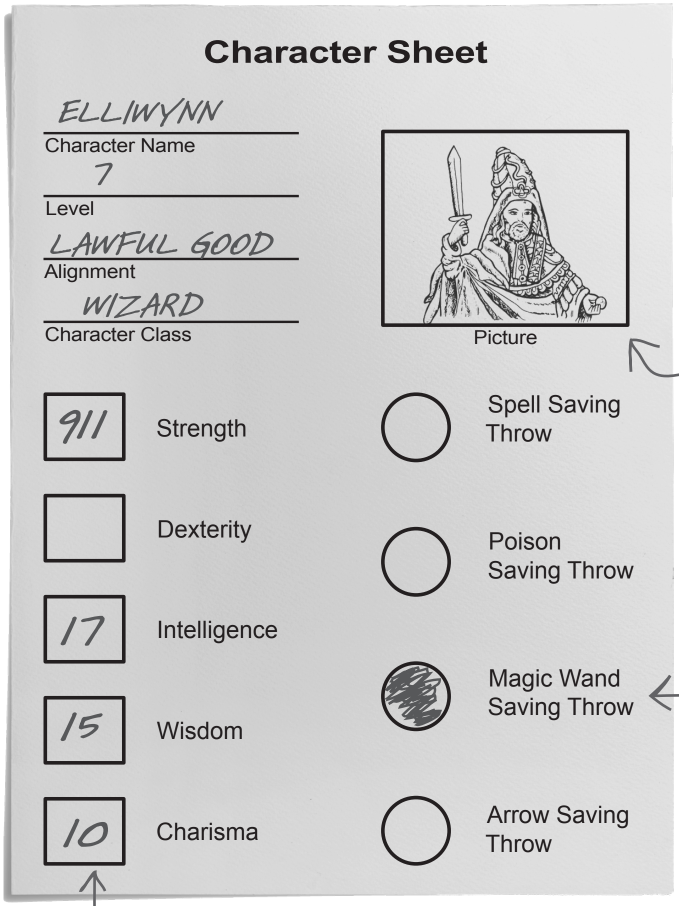
Can we find a way to help Owen improve his RPG?

Ability score (like strength, stamina, charisma, and intelligence) is an important mechanic in a lot of role-playing games. Players frequently roll dice and use a formula to determine their character's scores.

Character sheets store different types of data on paper

If you've ever played an RPG, you've seen character sheets: a page with details, statistics, background information, and any other notes you might see about a character. If you wanted to make a class to hold a character sheet, what types would you use for the fields?





Character Sheet

ELLIWYNN
Character Name

7
Level

LAWFUL GOOD
Alignment

WIZARD
Character Class

Picture

911	Strength	○	Spell Saving Throw
	Dexterity	○	Poison Saving Throw
17	Intelligence	●	Magic Wand Saving Throw
15	Wisdom	○	Arrow Saving Throw
10	Charisma		

This box is for a picture of the character. If you were building a C# class for a character sheet, you could save that picture in an image file.

In the RPG that Owen plays, saving throws give players a chance to roll dice and avoid certain types of attacks. This character has a magic wand saving throw, so the player filled in this circle.

Players create characters by rolling dice for each of their ability scores, which they write in these boxes.

BRAIN POWER

Look at the fields in the CharacterSheet class diagram. What type would you use for each field?

A variable's type determines what kind of data it can store

There are many **types** built into C#, and you'll use them to store many different kinds of data. You've already seen some of the most common ones, like int, string, bool, and float. There are a few others that you haven't seen, and they can really come in handy, too.

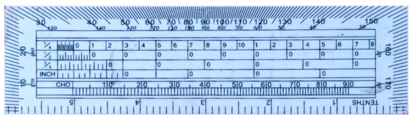
Here are some types you'll use a lot.

- **int** can store any **integer** from −2,147,483,648 to 2,147,483,647. Integers don't have decimal points.

- **string** can hold text of any length (including the empty string "").

- **bool** is a Boolean value—it's either true or false. You'll use it to represent anything that only has two options: it can either be one thing or another, but nothing else.

- **double** can store **real** numbers from ±5.0 × 10^{-324} to ±1.7 × 10^{308} with up to 16 significant digits. It's a really common type when you're working with XAML properties.



- **float** can store **real** numbers from ±1.5 × 10^{-45} to ±3.4 × 10^{38} with up to 8 significant digits.

BRAIN POWER

Why do you think C# has more than one type for storing numbers that have a decimal point?

C# has several types for storing integers

C# has several different types for integers, as well as int. This may seem a little odd (pun intended). Why have so many types for numbers without decimals? For most of the programs in this book, it won't matter if you use an int or a long. If you're writing a program that has to keep track of millions and millions of integer values, then choosing a smaller integer type like byte instead of a bigger type like long can save you a lot of memory.

- ***byte*** can store any **integer** between 0 and 255.
- ***sbyte*** can store any **integer** from –128 to 127.
- ***short*** can store any **integer** from –32,768 to 32,767.
- ***long*** can store any **integer** from –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

Notice how we're saying "integer" and not "whole number"? We're trying to be really careful—our high school math teachers always told us that integers are any numbers that can be written without a fraction, while whole numbers are integers starting at 0, and do not include negative numbers.

byte only stores small whole numbers from 0 to 255.

If you need to store a larger number, you can use a short, which stores integers from –32,768 to 32,767.

Long also stores integers, but it can store huge values.

Did you notice that byte only stores positive numbers, while sbyte stores negative numbers? They both have 256 possible values. The difference is that, like short and long, sbyte can have a negative sign—which is why those are called **signed** types, (the "s" in sbyte stands for signed). Just like byte is the **unsigned** version of sbyte, there are unsigned versions of short, int, and long that start with "u":

- ***ushort*** can store any **whole number** from 0 to 65,535.
- ***uint*** can store any **whole number** from 0 to 4,294,967,295.
- ***ulong*** can store any **whole number** from 0 to 18,446,744,073,709,551,615.

Types for storing really HUGE and really tiny numbers

Sometimes float just isn't precise enough. Believe it or not, sometimes 10^{38} isn't big enough and 10^{-45} isn't small enough. A lot of programs written for finance or scientific research run into these problems all the time, so C# gives us different **floating-point types** to handle huge and tiny values:

- ***float*** can store any number from $\pm1.5 \times 10^{-45}$ to $\pm3.4 \times 10^{38}$ with 6–9 significant digits.
- ***double*** can store any number from $\pm5.0 \times 10^{-324}$ to $\pm1.7 \times 10^{308}$ with 15–17 significant digits.
- ***decimal*** can store any number from $\pm1.0 \times 10^{-28}$ to $\pm7.9 \times 10^{28}$ with 28–29 significant digits. When your program **needs to deal with money or currency**, you always want to use a decimal to store the number.

The decimal type has a lot more precision (way more significant digits) which is why it's appropriate for financial calculations.

Floating-Point Numbers Up Close

The float and double types are called "floating-point" because the decimal point can move (as opposed to a "fixed-point" number, which always has the same number of decimal places). That—and, in fact, a lot stuff that has to do with floating-point numbers, especially precision—may seem a little ***weird***, so let's dig into the explanation.

"Significant digits" represents the precision of the number: 1,048,415, 104.8415, and .0000001048415 all have 7 significant digits. So when we say a float can store real numbers as big as 3.4×10^{38} or as small as -1.5×10^{-45}, that means it can store numbers as big as 8 digits followed by 30 zeros, or as small as 37 zeros followed by 8 digits.

If it's been a while since you've used exponents, 3.4×10^{38} means 34 followed by 37 zeros, and -1.5×10^{-45} is –.00…(40 more zeros)…0015.

The float and double types can also have special values, including both positive and negative zero, positive and negative infinity, and a special value called **NaN (not-a-number)** that represents, well, a value that isn't a number at all. They also have static methods that let you test for those special values. Try running this loop:

```
for (float f = 10; float.IsFinite(f); f *= f)
{
    Console.WriteLine(f);
}
```

Now try that same loop with double:

```
for (double d = 10; double.IsFinite(d); d *= d)
{
    Console.WriteLine(d);
}
```

Let's talk about strings

You've written code that works with **strings**. So what, exactly, is a string?

In any .NET app, a string is an object. Its full class name is System.String—in other words, the class name is String and it's in the System namespace (just like the Random class you used earlier). When you use the C# `string` keyword, you're working with System.String objects. In fact, you can replace `string` with `System.String` in any of the code you've written so far and it will still work! (The `string` keyword is called an *alias*—as far as your C# code is concerned, `string` and `System.String` mean the same thing.)

There are also two special values for strings: an empty string, "" (or a string with no characters), and a null string, or a string that isn't set to anything at all. We'll talk more about null later in the chapter.

Strings are made up of characters—specifically, Unicode characters (which you'll learn a lot more about later in the book). Sometimes you need to store a single character like `Q` or `j` or `$`, and when you do you'll use the **char** type. Literal values for char are always inside single quotes (`'x'`, `'3'`). You can include **escape sequences** in the quotes, too (`'\n'` is a line break, `'\t'` is a tab). You can write an escape sequence in your C# code using two characters, but your program stores each escape sequence as a single character in memory.

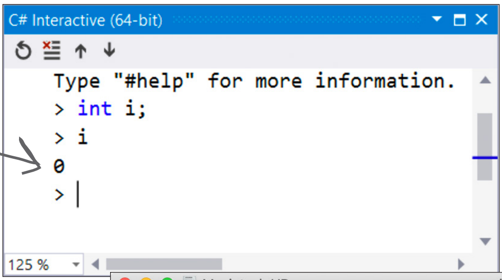
And finally, there's one more important type: **object**. If a variable has object as its type, ***you can assign any value to it***. The `object` keyword is also an alias—it's the same as `System.Object`.

Sharpen your pencil

Sometimes you declare a variable and set its value in a single statement like this: `int i = 37;`—but you already know that you don't have to set a value. So what happens if you use the variable without assigning it a value? Let's find out! Use the **C# Interactive window** (or the .NET console if you're using a Mac) to declare a variable and check its value.

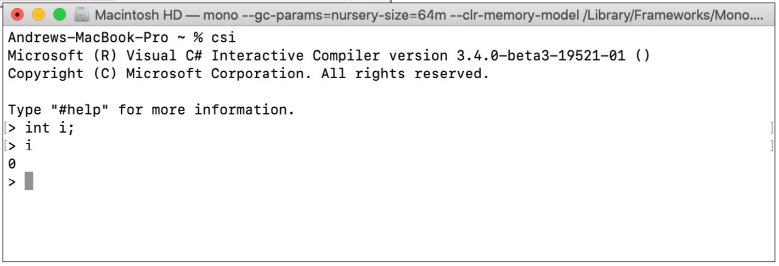
We wrote in the first answer for you.

0	`int i;`
....	`long l;`
....	`float f;`
....	`double d;`
....	`decimal m;`
....	`byte b;`
....	`char c;`
....	`string s;`
....	`bool t;`



```
Type "#help" for more information.
> int i;
> i
0
> |
```

Start the C# Interactive window (from the View >> Other Windows menu) or run `csi` from the Mac Terminal. Declare each variable, then enter the variable name to see its default value. Write the default value for each type in the space provided.


Macintosh HD — mono --gc-params=nursery-size=64m --clr-memory-model /Library/Frameworks/Mono....

```
Andrews-MacBook-Pro ~ % csi
Microsoft (R) Visual C# Interactive Compiler version 3.4.0-beta3-19521-01 ()
Copyright (C) Microsoft Corporation. All rights reserved.

Type "#help" for more information.
> int i;
> i
0
> 
```

A literal is a value written directly into your code

A **literal** is a number, string, or other fixed value that you include in your code. You've already used plenty of literals—here are some examples of numbers, strings, and other literals that you've used:

```
int number = 15;
string result = "the answer";
public bool GameOver = false;
Console.Write("Enter the number of cards to pick: ");
if (value == 1) return "Ace";
```

Can you spot all of the literals in these statements from code you've written in previous chapters? The last statement has two literals.

So when you type `int i = 5;`, the `5` is a literal.

Use suffixes to give your literals types

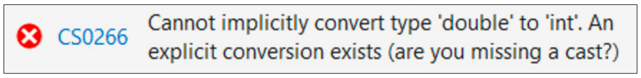
When you added statements like this in Unity, you may have wondered about the **F**:

```
InvokeRepeating("AddABall", 1.5F, 1);
```

Did you notice that your program won't build if you leave off the F in the literals 1.5F and 0.75F? That's because **literals have types**. Every literal is automatically assigned a type, and C# has rules about how you can combine different types. You can see for yourself how that works. Add this line to any C# program:

```
int wholeNumber = 14.7;
```

When you try to build your program, the IDE will show you this error in the Error List:


CS0266 Cannot implicitly convert type 'double' to 'int'. An explicit conversion exists (are you missing a cast?)

The IDE is telling you is that the literal 14.7 has a type—it's a double. You can use a suffix to change its type—try changing it to a float by sticking an F on the end (14.7F) or a decimal by adding M (14.7M—the M actually stands for "money"). The error message now says it can't convert float or decimal. Add a D (or leave off the suffix entirely) and the error goes away.

C# assumes that an integer literal without a suffix (like 371) is an int, and one with a decimal point (like 27.4) is a double.

Sharpen your pencil Solution

0	`int i;`
0	`long l;`
0	`float f;`
0	`double d;`
0	`decimal m;`
0	`byte b;`
'\0'	`char c;`
null	`string s;`
false	`bool t;`

If you used the C# command line on Mac or Unix, you might see '\x0' instead of '\0' as the default value for char. We'll take a deep dive into exactly what this means later in the book when we talk about Unicode.

Sharpen your pencil

C# has dozens of **reserved words called keywords**. They're words reserved by the C# compiler that you can't use for variable names. You've already learned many of them—here's a little review to help seal them into your brain. Write down what you think each of these keywords does in C#.

`namespace`	
`for`	
`class`	
`else`	
`new`	
`using`	
`if`	
`while`	

If you really want to use a reserved keyword as a variable name, put @ in front of it, but that's as close as the compiler will let you get to the reserved word. You can also do that with nonreserved names, if you want to.

Sharpen your pencil Solution

C# has dozens of **reserved words called *keywords***. They're words reserved by the C# compiler that you can't use for variable names. You've already learned many of them—here's a little review to help seal them into your brain. Write down what you think each of these keywords does in C#.

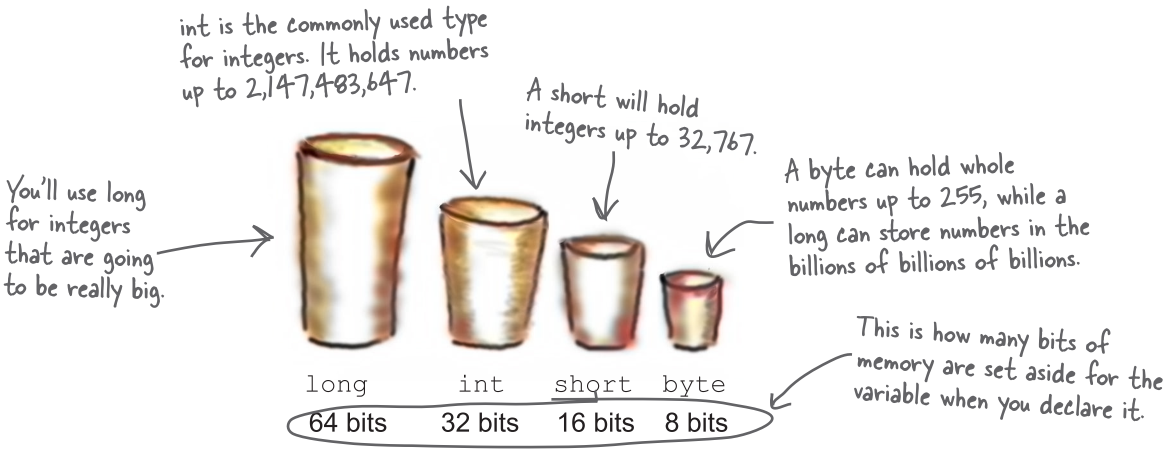
namespace	All of the classes and methods in a program are inside a namespace. Namespaces help make sure that the names you are using in your program don't clash with the ones in the .NET Framework or other classes.
for	This lets you do a loop that executes three statements. First it declares the variable it's going to use, then there's the statement that evaluates the variable against a condition. The third statement does something to the value.
class	Classes contain methods and fields, and you use them to instantiate objects. Fields are what objects know and methods are what they do.
else	A block of code that starts with else must immediately follow an if block, and will get executed if the if statement preceding it fails.
new	You use this to create a new instance of an object.
using	This is a way of listing off all of the namespaces you are using in your program. A using statement lets you use classes from various parts of the .NET Framework.
if	This is one way of setting up a conditional statement in a program. It says if one thing is true, do one thing; if not, do something else.
while	while loops are loops that keep on going as long as the condition at the beginning of the loop is true.

A variable is like a data to-go cup

All of your data takes up space in memory. (Remember the heap from the last chapter?) So part of your job is to think about how *much* space you're going to need whenever you use a string or a number in your program. That's one of the reasons you use variables. They let you set aside enough space in memory to store your data.

Not all data ends up on the heap. Value types usually keep their data in another part of memory called the stack. You'll learn all about that later in the book.

Think of a variable like a cup that you keep your data in. C# uses a bunch of different kinds of cups to hold different kinds of data. Just like the different sizes of cups at a coffee shop, there are different sizes of variables, too.


int is the commonly used type for integers. It holds numbers up to 2,147,483,647.
A short will hold integers up to 32,767.
You'll use long for integers that are going to be really big.
A byte can hold whole numbers up to 255, while a long can store numbers in the billions of billions of billions.
This is how many bits of memory are set aside for the variable when you declare it.
long
int
short
byte
64 bits
32 bits
16 bits
8 bits

Convert this!

Use the Convert class to explore bits and bytes

You've always heard that programming is about 1s and 0s. .NET has a **static Convert class** that converts between different numeric data types. Let's use it to see an example of how bits and bytes work.

A bit is a single 1 or 0. A byte is 8 bits, so a byte variable holds an 8-bit number, which means it's a number that can be represented with up to 8 bits. What does that look like? Let's use the Convert class to convert some binary numbers to bytes:

```
Convert.ToByte("10111", 2) // returns 23
Convert.ToByte("11111111", 2); // returns 255
```

The first argument to Convert.ToByte is the number to convert, and the second is its base. Binary numbers are base 2.

Bytes can hold numbers between 0 and 255 because they use 8 bits of memory—an 8-bit number is a binary number between 0 and 11111111 binary (or 0 and 255 decimal).

A short is a 16-bit value. Let's use Convert.ToInt16 to convert the binary value 111111111111111 (15 1s) to a short. An int is a 32-bit value, so we'll use Convert.ToInt32 to convert the 31 1s to an int:

```
Convert.ToInt16("111111111111111", 2); // returns 32767
Convert.ToInt32("1111111111111111111111111111111", 2); // returns 2147483647
```

Other types come in different sizes, too

Numbers that have decimal places are stored differently than integers, and the different floating-point types take up different amounts of memory. You can handle most of your numbers that have decimal places using **float**, the smallest data type that stores decimals. If you need to be more precise, use a **double**. If you're writing a financial application where you'll be storing currency values, you'll always want to use the **decimal** type.

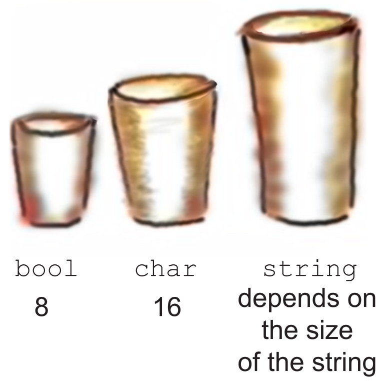
Oh, and one more thing: ***don't use double for money or currency, <u>only use decimal</u>*.**

float
32 bits
double
64 bits
decimal
128 bits

These types are for fractions. Larger variables store more decimal places.

We've talked about strings, so you know that the C# compiler also can handle **characters and non-numeric types**. The char type holds one character, and string is used for lots of characters "strung" together. There's no set size for a string object—it expands to hold as much data as you need to store in it. The bool data type is used to store true or false values, like the ones you've used for your `if` statements.

C# also has types for storing data that is not numeric.



Strings can be big... REALLY big! C# uses a 32-bit integer to keep track of the string length, so the maximum string length is 2^{31} (or 2,147,483,648) characters.

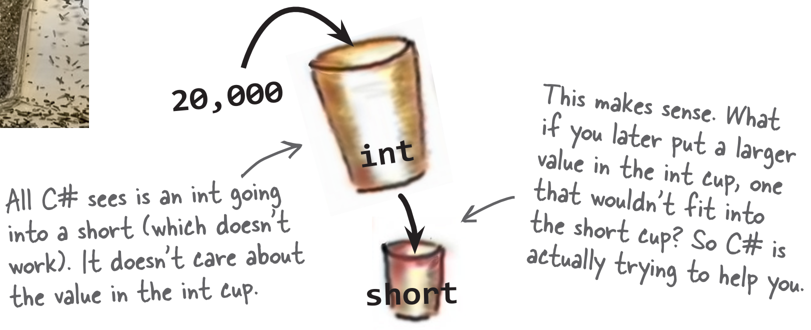
The different floating-point types take up different amounts of memory: float is smallest, and decimal is largest.

10 pounds of data in a 5-pound bag

When you declare your variable as one type, the C# compiler **allocates** (or reserves) all of the memory it would need to store the maximum value of that type. Even if the value is nowhere near the upper boundary of the type you've declared, the compiler will see the cup it's in, not the number inside. So this won't work:

```
int leaguesUnderTheSea = 20000;
short smallerLeagues = leaguesUnderTheSea;
```

20,000 would fit into a short, no problem. But because `leaguesUnderTheSea` is declared as an int, C# sees it as int-sized and considers it too big to put in a short container. The compiler won't make those translations for you on the fly. You need to make sure that you're using the right type for the data you're working with.


20,000
int
short
All C# sees is an int going into a short (which doesn't work). It doesn't care about the value in the int cup.
This makes sense. What if you later put a larger value in the int cup, one that wouldn't fit into the short cup? So C# is actually trying to help you.

Sharpen your pencil

Three of these statements won't build, either because they're trying to cram too much data into a small variable or because they're putting the wrong type of data in. Circle them and write a brief explanation of what's wrong.

```
int hours = 24;                          string taunt = "your mother";

short y = 78000;                         byte days = 365;

bool isDone = yes;                       long radius = 3;

short RPM = 33;                          char initial = 'S';

int balance = 345667 - 567;              string months = "12";
```

Casting lets you copy values that C# can't automatically convert to another type

Let's see what happens when you try to assign a decimal value to an int variable.

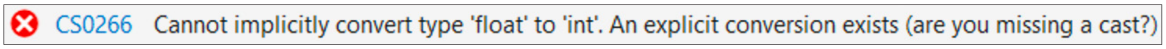
Do this!

1. Create a new Console App project and add this code to the Main method:

```
float myFloatValue = 10;
int myIntValue = myFloatValue;
Console.WriteLine("myIntValue is " + myIntValue);
```

Implicit conversion means C# has a way to automatically convert a value to another type without losing information.

2. Try building your program. You should get the same CS0266 error you saw earlier:



```
CS0266  Cannot implicitly convert type 'float' to 'int'. An explicit conversion exists (are you missing a cast?)
```

Look closely at the last few words of the error message: "are you missing a cast?" That's the C# compiler giving you a really useful hint about how to fix the problem.

3. Make the error go away by **casting** the decimal to an int. You do this by adding the type that you want to convert to in parentheses: **`(int)`**. Once you change the second line so it looks like this, your program will compile and run:

```
int myIntValue = (int) myFloatValue;
```

Here's where you cast the decimal value to an int.

When you cast a floating-point value to an int, it rounds the value down to the nearest integer.

So what happened?

The C# compiler won't let you assign a value to a variable if it's the wrong type—even if that variable can hold the value just fine! It turns out that a LOT of bugs are caused by type problems, and **the compiler is helping** by nudging you in the right direction. When you use casting, you're essentially saying to the compiler that you know the types are different, and promising that in this particular instance it's OK for C# to cram the data into the new variable.

Sharpen your pencil Solution

Three of these statements won't build, either because they're trying to cram too much data into a small variable or because they're putting the wrong type of data in. Circle them and write a brief explanation of what's wrong

```
short y = 78000;
```

The short type holds numbers from -32,767 to 32,768. This number's too big!

```
byte days = 365;
```

A byte can only hold a value between 0 and 255. You'll need a short for this.

```
bool isDone = yes;
```

You can only assign a value of "true" or "false" to a bool.

When you cast a value that's too big, C# adjusts it to fit its new container

You've already seen that a decimal can be cast to an int. It turns out that *any* number can be cast to *any other* number. That doesn't mean the ***value*** stays intact through the casting, though. Say you have an int variable set to 365. If you cast it to a byte variable (max value 255), instead of giving you an error, the value will just **wrap around**. 256 cast to a byte will have a value of 0, 25 will be converted to 1, 258 to 2, etc., up to 365, which will end up being **109**. Once you get back to 255 again, the conversion value "wraps" back to zero.

If you use + (or `*`, `/`, or `-`) with two different numeric types, the operator **automatically converts** the smaller type to the bigger one. Here's an example:

```
int myInt = 36;
float myFloat = 16.4F;
myFloat = myInt + myFloat;
```

Since an int can fit into a float but a float can't fit into an int, the + operator converts `myInt` to a float before adding it to `myFloat`.

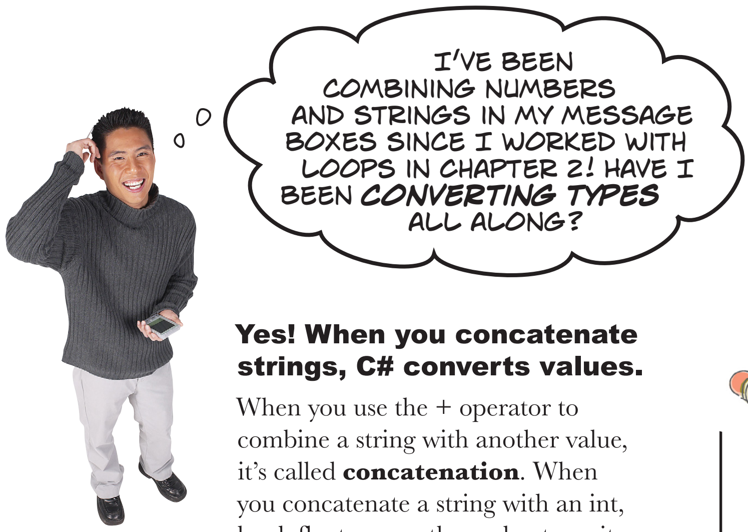
Sharpen your pencil

You can't always cast any type to any other type.

Create a new Console App project and type these statements into its Main method. Then build your program—it will give lots of errors. Cross out the ones that give errors. That'll help you figure out which types can be cast, and which can't!

```
int myInt = 10;

byte myByte = (byte)myInt;

double myDouble = (double)myByte;

bool myBool = (bool)myDouble;

string myString = "false";

myBool = (bool)myString;

myString = (string)myInt;

myString = myInt.ToString();

myBool = (bool)myByte;

myByte = (byte)myBool;

short myShort = (short)myInt;

char myChar = 'x';

myString = (string)myChar;

long myLong = (long)myInt;

decimal myDecimal = (decimal)myLong;

myString = myString + myInt +
myByte + myDouble + myChar;
```

You can read a lot more about the different C# value types here—it's worth taking a look: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/value-types.


I'VE BEEN COMBINING NUMBERS AND STRINGS IN MY MESSAGE BOXES SINCE I WORKED WITH LOOPS IN CHAPTER 2! HAVE I BEEN CONVERTING TYPES ALL ALONG?

Yes! When you concatenate strings, C# converts values.

When you use the + operator to combine a string with another value, it's called **concatenation**. When you concatenate a string with an int, bool, float, or another value type, it automatically converts the value. This kind of conversion is different from casting, because under the hood it's really calling the ToString method for the value…and one thing that .NET guarantees is that **every object has a ToString method** that converts it to a string (but it's up to the individual class to determine if that string makes sense).

Wrap it yourself!

There's no mystery to how casting "wraps" the numbers—you can do it yourself. Just open up any calculator app that has a Mod button (which does a modulus calculation—sometimes in a Scientific mode), and calculate 365 Mod 256.

Sharpen your pencil Solution

You can't always cast any type to any other type. Create a new Console App project and type these statements into its Main method. Then build your program—it will give lots of errors. Cross out the ones that give errors. That'll help you figure out which types can be cast , and which can't!

```
int myInt = 10;
byte myByte = (byte)myInt;
double myDouble = (double)myByte;
~~bool myBool = (bool)myDouble;~~
string myString = "false";
~~myBool = (bool)myString;~~
~~myString = (string)myInt;~~
myString = myInt.ToString();
~~myBool = (bool)myByte;~~
~~myByte = (byte)myBool;~~
short myShort = (short)myInt;
char myChar = 'x';
~~myString = (string)myChar;~~
long myLong = (long)myInt;
decimal myDecimal = (decimal)
myLong;
myString = myString + myInt +
myByte + myDouble + myChar;
```

C# does some conversions automatically

There are two important conversions that don't require you to do casting. The first is the automatic conversion that happens any time you use arithmetic operators, like in this example:

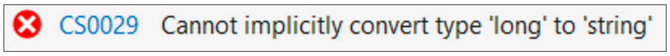
```
long l = 139401930;
short s = 516;
double d = l - s;
d = d / 123.456;
Console.WriteLine("The answer is " + d);
```

The – operator subtracted the short from the long, and the = operator converted the result to a double.

The other way C# converts types for you automatically is when you use the + operator to **concatenate** strings (which just means sticking one string on the end of another, like you've been doing with message boxes). When you use + to concatenate a string with something that's another type, it automatically converts the numbers to strings for you. Here's an example—try adding these lines to any C# program. The first two lines are fine, but the third one won't compile:

```
long number = 139401930;
string text = "Player score: " + number;
text = number;
```

The C# compiler gives you this error on the third line:



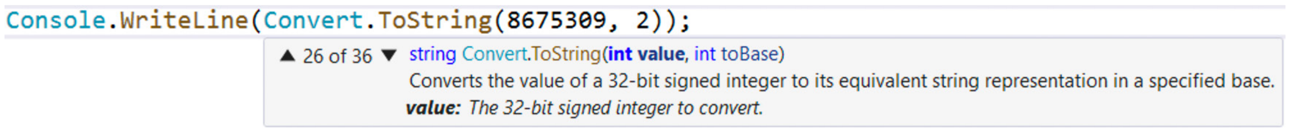
```
CS0029 Cannot implicitly convert type 'long' to 'string'
```

ScoreText.text is a string field, so when you used the + operator to concatenate a string it assigned the value just fine. But when you try to assign `x` to it directly, it doesn't have a way to automatically convert the long value to a string. You can convert it to a string by calling its ToString method.

there are no Dumb Questions

Q: You used the Convert.ToByte, Convert.ToInt32, and Convert.ToInt64 methods to convert strings with binary numbers into integer values. Can you convert integer values back to binary?

A: Yes. The Convert class has a **Convert.ToString method** that converts many different types of values to strings. The IntelliSense pop-up shows you how it works:



```
Console.WriteLine(Convert.ToString(8675309, 2));
```

▲ 26 of 36 ▼ string Convert.ToString(**int value**, int toBase)
Converts the value of a 32-bit signed integer to its equivalent string representation in a specified base.
value: *The 32-bit signed integer to convert.*

So `Convert.ToString(255, 2)` returns the string "11111111", and `Convert.ToString(8675309, 2)` returns the string "100001000101111111101101"—try experimenting with it to get a feel for how binary numbers work.

When you call a method, the arguments need to be compatible with the types of the parameters

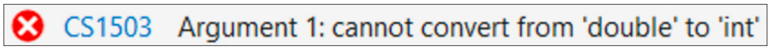
In the last chapter, you used the Random class to choose a random number from 1 up to (but not including) 5, which you used to pick a suit for a playing card:

```
int value = random.Next(1, 5);
```

Try changing the first argument from `1` to `1.0`:

```
int value = random.Next(1.0, 5);
```

You're passing a double literal to a method that's expecting an int value. So it shouldn't surprise you that the compiler won't build your program—instead, it shows an error:



```
CS1503   Argument 1: cannot convert from 'double' to 'int'
```

When the compiler gives you an "invalid argument" error, it means that you tried to call a method with variables whose types didn't match the method's parameters.

Sometimes C# can do the conversion automatically. It doesn't know how to convert a double to an int (like converting 1.0 to 1), but it does know how to convert an int to a double (like converting 1 to 1.0). More specifically:

- ★ The C# compiler knows how convert an integer to a floating-point type.
- ★ And it knows how to convert an integer type to another integer type, or a floating-point type to another floating-point type.
- ★ But it can only do those conversions if the type it's converting from is the same size as or smaller than the type it's converting to. So, it can convert an int to a long or a float to a double, but it can't convert a long to an int or a double to a float.

But Random.Next isn't the only method that will give you compiler errors if you try to pass it a variable whose type doesn't match the parameter. *All* methods will do that, ***even the ones you write yourself***. Add this method to a console app:

```
public int MyMethod(bool add3) {
   int value = 12;

   if (add3)
      value += 3;
   else
      value -= 2;

   return value;
}
```

Try passing it a string or long—you'll get one of those CS1503 errors telling you it can't convert the argument to a bool. Some folks have trouble remembering **the difference between parameter and an argument**. So just to be clear:

A parameter is what you define in your method. An argument is what you pass to it. You can pass a byte argument to a method with an int parameter.

there are no Dumb Questions

Q: That last `if` statement only said `if (add3)`. Is that the same thing as `if (add3 == true)`?

A: Yes. Let's take another look at that `if/else` statement:

```
if (add3)
   value += 3;
else
   value -= 2;
```

An `if` statement always checks if something's true. So because the type of the `add3` variable is bool, it evaluates to either true or false, which means we didn't have to explicitly include `== true`.

You can also check if something's false using ! (an exclamation point, or the NOT operator). Writing `if (!add3)` is the same thing as writing `if (add3 == false)`.

In our code examples from now on, if we're using the conditional test to check a Boolean variable, you'll usually just see us write `if (add3)` or `if (!add3)`, and not use == to explicitly check to see if the Boolean is true or false.

Q: You didn't include curly braces in the `if` or `else` blocks, either. Does that mean they're optional?

A: Yes—but only if there's a single statement in the `if` or `else` block. We could leave out the `{` curly braces `}` because there was just one statement in the `if` block (`return 45;`) and one statement in the `else` block (`return 61;`). If we wanted to add another statement to one of those blocks, we'd have to use curly braces for it:

```
if (add3)
   value += 3;
else {
   Console.WriteLine("Subtracting 2");
   value -= 2;
}
```

Be careful when you leave out curly braces because it's easy to accidentally write code that doesn't do what you want it to do. It never hurts to add curly braces, but it's also good to get used to seeing `if` statements both with and without them.

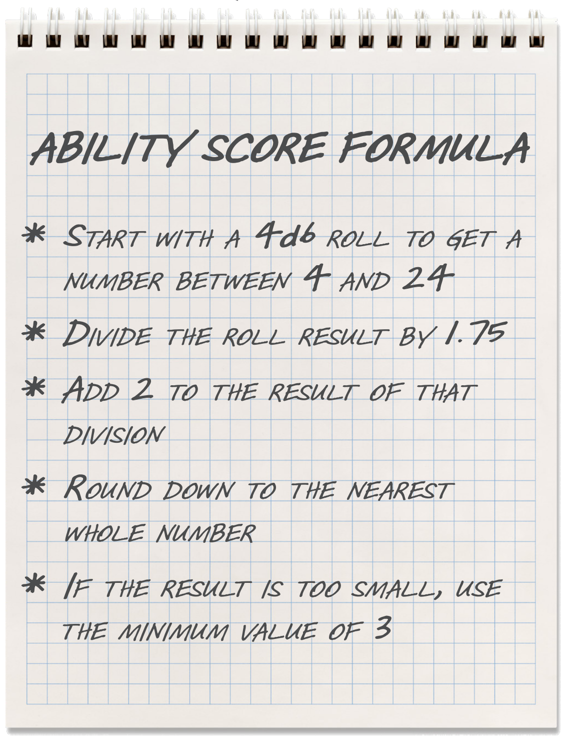
BULLET POINTS

- There are **value types** for variables that hold different sizes of numbers. The biggest numbers should be of type long and the smallest ones (up to 255) can be declared as bytes.
- Every value type has a **size**, and you can't put a value of a bigger type into a smaller variable, no matter what the actual size of the data is.
- When you're using **literal** values, use the F suffix to indicate a float (15.6F) and M for a decimal (36.12M).
- Use the **decimal type for money and currency**. Floating-point precision is…well, it's a little weird.
- There are a few types that C# knows how to **convert** automatically (an implicit conversion), like short to int, int to double, or float to double.
- When the compiler won't let you set a variable equal to a value of a different type, that's when you need to cast it. To **cast** a value (an explicit conversion) to another type, put the target type in parentheses in front of the value.
- There are some keywords that are **reserved** by the language and you can't name your variables with them. They're words (like `for`, `while`, `using`, `new`, and others) that do specific things in the language.
- A **parameter** is what you define in your method. An **argument** is what you pass to it.
- When you build your code in the IDE, it uses the **C# compiler** to turn it into an executable program.
- You can use methods on the static **Convert class** to convert values between different types.

Owen is constantly improving his game...

Good game masters are dedicated to creating the best experience they can for their players. Owen's players are about to embark on a new campaign with a brand-new set of characters, and he thinks a few tweaks to the formula that they use for their ability scores could make things more interesting.

When players fill out their character sheets at the start of the game, they follow these steps to calculate each of the ability scores for their character.



ABILITY SCORE FORMULA

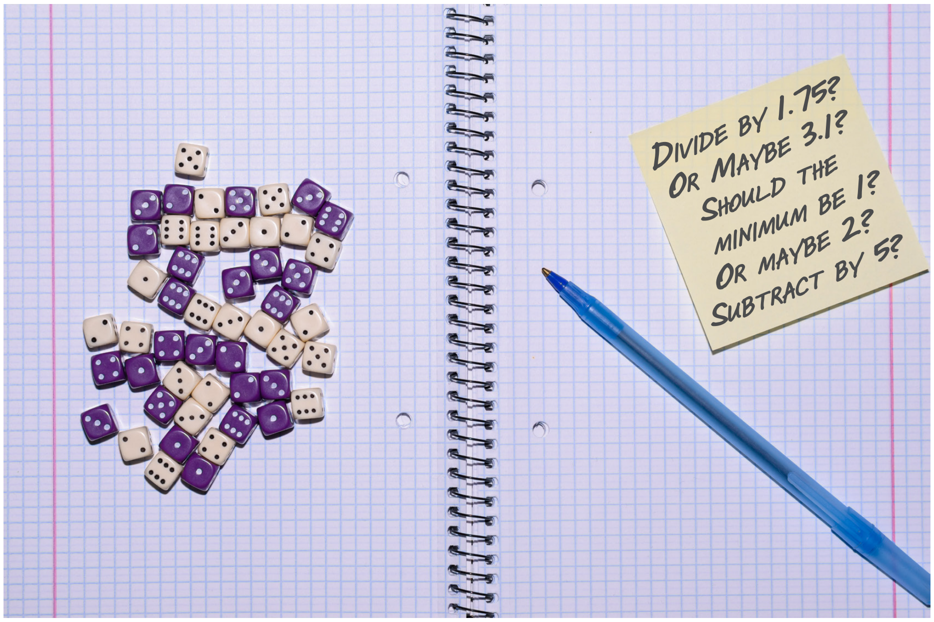
* START WITH A 4d6 ROLL TO GET A NUMBER BETWEEN 4 AND 24
* DIVIDE THE ROLL RESULT BY 1.75
* ADD 2 TO THE RESULT OF THAT DIVISION
* ROUND DOWN TO THE NEAREST WHOLE NUMBER
* IF THE RESULT IS TOO SMALL, USE THE MINIMUM VALUE OF 3

A "4d6 ROLL" means rolling four normal six-sided dice and adding up the results.

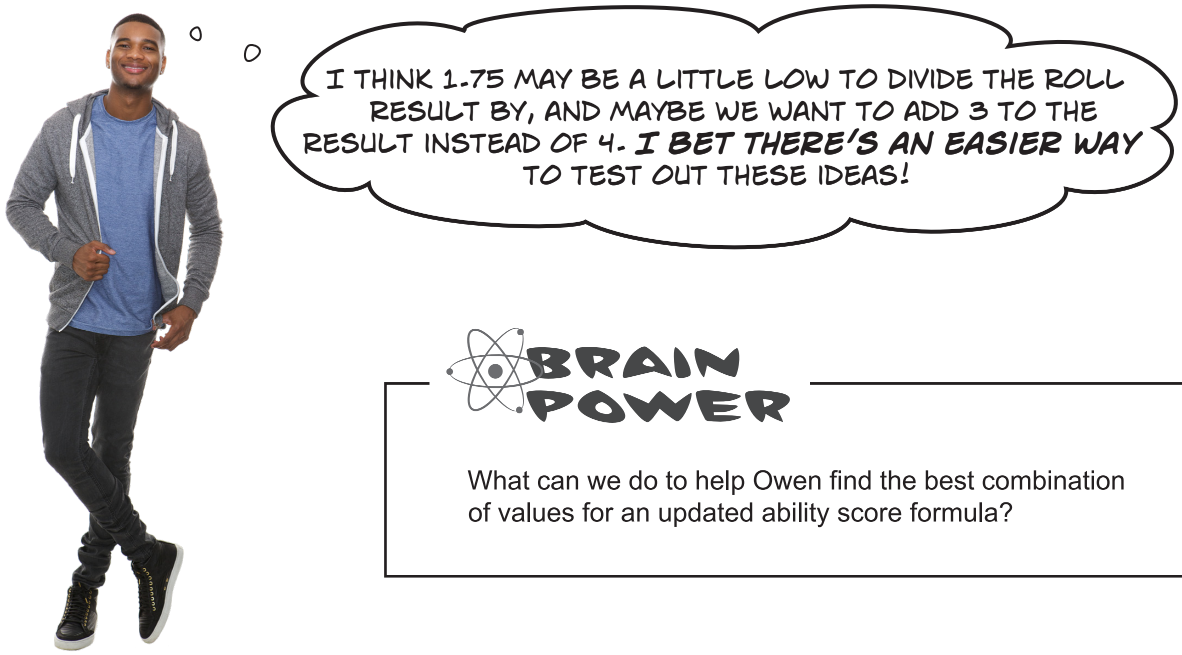
THE STANDARD RULES FOR THIS GAME ARE A GOOD STARTING POINT, BUT I KNOW WE CAN DO BETTER.

...but the trial and error can be time-consuming

Owen's been experimenting with ways to tweak the ability score calculation. He's pretty sure that he has the formula mostly right—but he'd really like to tweak the numbers.


DIVIDE BY 1.75? OR MAYBE 3.1? SHOULD THE MINIMUM BE 1? OR MAYBE 2? SUBTRACT BY 5?

Owen likes the overall formula: 4d6 roll, divide, subtract, round down, use a minimum value…but he's not sure that the actual numbers are right.


I THINK 1.75 MAY BE A LITTLE LOW TO DIVIDE THE ROLL RESULT BY, AND MAYBE WE WANT TO ADD 3 TO THE RESULT INSTEAD OF 4. ***I BET THERE'S AN EASIER WAY*** TO TEST OUT THESE IDEAS!

BRAIN POWER

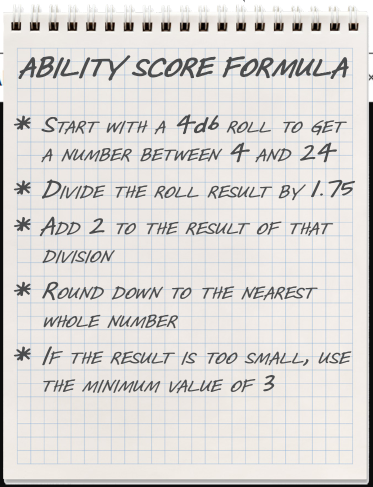
What can we do to help Owen find the best combination of values for an updated ability score formula?

Let's help Owen experiment with ability scores

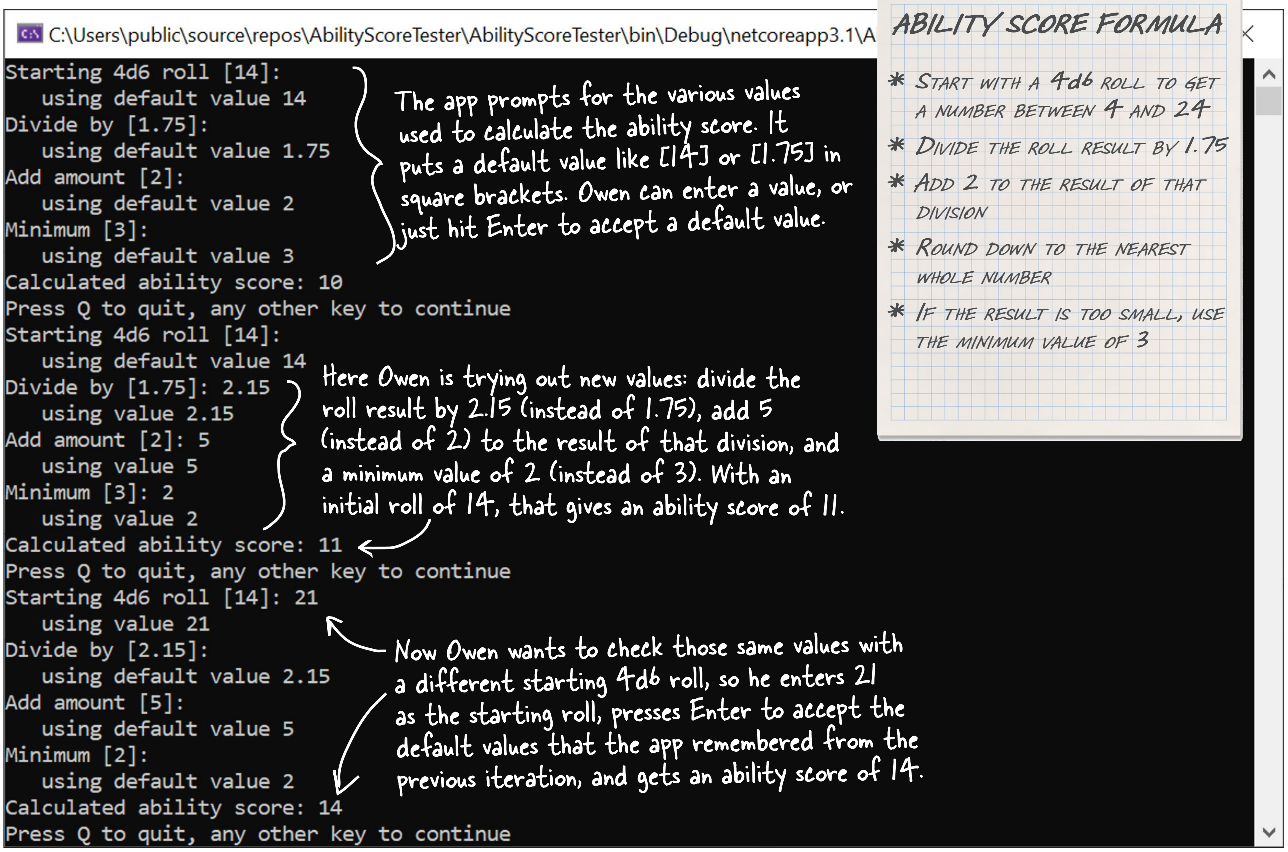
In this next project, you'll build a .NET Core console app that Owen can use to test his ability score formula with different values to see how they affect the resulting score. The formula has **four inputs**: the *starting 4d6 roll*, the *divide by* value that the roll result is divided by, the *add amount* value to add to the result of that division, and the *minimum* to use if the result is too small.

Owen will enter each of the four inputs into the app, and it will calculate the ability score using those inputs. He'll probably want to test a bunch of different values, so we'll make the app easier by to use by asking for new values over and over again until he quits the app, keeping track of the values he used in each iteration and using those previous inputs as **default values** for the next iteration.

Here's the page from Owen's game master notebook with the ability score formula.


ABILITY SCORE FORMULA
* START WITH A 4d6 ROLL TO GET A NUMBER BETWEEN 4 AND 24
* DIVIDE THE ROLL RESULT BY 1.75
* ADD 2 TO THE RESULT OF THAT DIVISION
* ROUND DOWN TO THE NEAREST WHOLE NUMBER
* IF THE RESULT IS TOO SMALL, USE THE MINIMUM VALUE OF 3

This is what it looks like when Owen runs the app:



C:\Users\public\source\repos\AbilityScoreTester\AbilityScoreTester\bin\Debug\netcoreapp3.1\A

```
Starting 4d6 roll [14]:
   using default value 14
Divide by [1.75]:
   using default value 1.75
Add amount [2]:
   using default value 2
Minimum [3]:
   using default value 3
Calculated ability score: 10
Press Q to quit, any other key to continue
Starting 4d6 roll [14]:
   using default value 14
Divide by [1.75]: 2.15
   using value 2.15
Add amount [2]: 5
   using value 5
Minimum [3]: 2
   using value 2
Calculated ability score: 11
Press Q to quit, any other key to continue
Starting 4d6 roll [14]: 21
   using value 21
Divide by [2.15]:
   using default value 2.15
Add amount [5]:
   using default value 5
Minimum [2]:
   using default value 2
Calculated ability score: 14
Press Q to quit, any other key to continue
```

The app prompts for the various values used to calculate the ability score. It puts a default value like [14] or [1.75] in square brackets. Owen can enter a value, or just hit Enter to accept a default value.

Here Owen is trying out new values: divide the roll result by 2.15 (instead of 1.75), add 5 (instead of 2) to the result of that division, and a minimum value of 2 (instead of 3). With an initial roll of 14, that gives an ability score of 11.

Now Owen wants to check those same values with a different starting 4d6 roll, so he enters 21 as the starting roll, presses Enter to accept the default values that the app remembered from the previous iteration, and gets an ability score of 14.

This project is a little larger than the previous console apps that you've built, so we'll tackle it in a few steps. First you'll Sharpen your pencil to understand the code to calculate the ability score, then you'll do an Exercise to write the rest of the code for the app, and finally you'll Sleuth out a bug in the code. *Let's get started!*

We've built a class to help Owen calculate ability scores. To use it, you set its Starting4D6Roll, DivideBy, AddAmount, and Minimum fields—or just leave the values set in their declarations—and call its CalculateAbilityScore method. Unfortunately, **there's one line of code that has problems**. Circle the line of code with problems and write down what's wrong with it.

See if you can spot the problem without typing the class into your IDE. Can you find the broken line that will cause a compiler error?

```
class AbilityScoreCalculator
{
    public int RollResult = 14;
    public double DivideBy = 1.75;
    public int AddAmount = 2;
    public int Minimum = 3;
    public int Score;

    public void CalculateAbilityScore()
    {
        // Divide the roll result by the DivideBy field
        double divided = RollResult / DivideBy;

        // Add AddAmount to the result of that division
        int added = AddAmount += divided;

        // If the result is too small, use Minimum
        if (added < Minimum)
        {
            Score = Minimum;
        } else
        {
            Score = added;
        }
    }
}
```

These fields are initialized with the values from the ability score formula. The app will use them to present default values to the user.

Here's a hint! Compare the comments in the code to the ability score formula on the page from Owen's game master notebook. What part of the formula is missing from the comments?

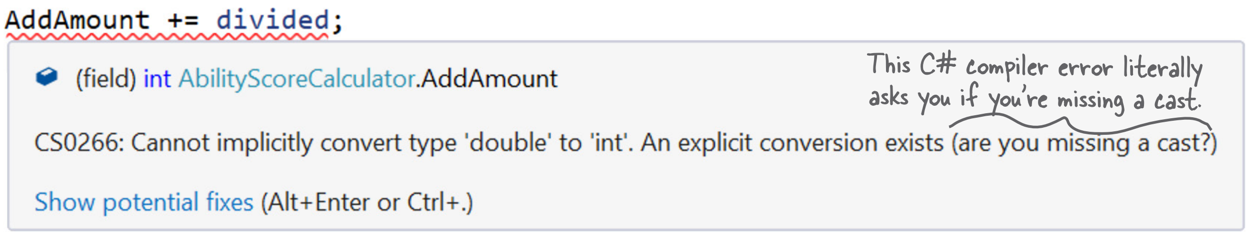
After you **circle the line of code that has problems**, write down the problems that you found with it.

....

....

Use the C# compiler to find the problematic line of code

Create a new .NET Core Console App project called AbilityScoreTester. Then **add the AbilityScoreCalculator class** with the code from the "Sharpen your pencil" exercise. If you entered the code correctly, you should see a C# compiler error:



```
AddAmount += divided;
```

(field) int AbilityScoreCalculator.AddAmount

CS0266: Cannot implicitly convert type 'double' to 'int'. An explicit conversion exists (are you missing a cast?)

Show potential fixes (Alt+Enter or Ctrl+.)

This C# compiler error literally asks you if you're missing a cast.

Any time the C# compiler gives you an error, read it carefully. It usually has a hint that can help you track down the problem. In this case, it's telling us exactly what went wrong: it can't convert a double to an int without a cast. The `divided` variable is declared as a double, but C# won't allow you to add it to an int field like AddAmount because it doesn't know how to convert it.

When the C# compiler asks "are you missing a cast?" it's giving you a huge hint that you need to explicitly cast the double variable `divided` before you can add it to the int field AddAmount.

Add a cast to get the AbilityScoreCalculator class to compile...

Now that you know what the problem is, you can **add a cast** to fix the problematic line of code in AbilityScoreCalculator. Here's the line that generated the "Cannot implicitly convert type" error:

```
    int added = AddAmount += divided;
```

It caused that error because **`AddAmount += divided`** ***returns a double value***, which can't be assigned to the int variable `added`.

You can fix it by **casting `divided` to an int**, so adding it to AddAmount returns another int. Modify that line of code to change `divided` to `(int)divided`:

```
    int added = AddAmount += (int)divided;
```

Cast this!

Adding that cast also addresses the missing part of Owen's ability score formula:

* ROUND DOWN TO THE NEAREST WHOLE NUMBER

When you cast a double to an int C# rounds it down—so for example `(int)19.7431D` gives us `19`. By adding that cast, you're adding the step from the ability score formula that was missing from the class.

...but there's still a bug!

We're not quite done yet! You fixed the compiler error, so now the project builds. But even though the C# compiler will accept it, ***there's still a problem***. Can you spot the bug in that line of code?

Looks like we can't fill in the "Sharpen your pencil" answer just yet!

Exercise

Finish building the console app that uses the AbilityScoreCalculator class. In this exercise, we'll give you the Main method for the console app. Your job is to write code for two methods: a method called ReadInt that reads user input and converts it to an int using int.TryParse, and a method called ReadDouble that does exactly the same thing except it parses doubles instead of int values.

1. Add the following Main method. Almost everything was used in previous projects. There's only one new thing—it calls the Console.ReadKey method:

```
        char keyChar = Console.ReadKey(true).KeyChar;
```

Console.ReadKey reads a single key from the console. When you pass the argument **`true`** it intercepts the input so that it doesn't get printed to the console. Adding **`.KeyChar`** causes it to return the key pressed as a **char**.

You'll use a single instance of AbilityScoreCalculator, using the user input to update its fields so it remembers the default values for the next iteration of the `while` loop.

Here's the full Main method—add it to your program:

```
static void Main(string[] args)
{
    AbilityScoreCalculator calculator = new AbilityScoreCalculator();
    while (true)
    {
        calculator.RollResult = ReadInt(calculator.RollResult, "Starting 4d6 roll");
        calculator.DivideBy = ReadDouble(calculator.DivideBy, "Divide by");
        calculator.AddAmount = ReadInt(calculator.AddAmount, "Add amount");
        calculator.Minimum = ReadInt(calculator.Minimum, "Minimum");
        calculator.CalculateAbilityScore();
        Console.WriteLine("Calculated ability score: " + calculator.Score);
        Console.WriteLine("Press Q to quit, any other key to continue");
        char keyChar = Console.ReadKey(true).KeyChar;
        if ((keyChar == 'Q') || (keyChar == 'q')) return;
    }
}
```

2. Add a method called ReadInt. It takes two parameters: a prompt to display to the user, and a default value. It writes the prompt to the console, followed by the default value in square brackets. Then it reads a line from the console and attempts to parse it. If the value can be parsed, it uses that value; otherwise, it uses the default value.

```
/// <summary>
/// Writes a prompt and reads an int value from the console.
/// </summary>
/// <param name="lastUsedValue">The default value.</param>
/// <param name="prompt">Prompt to print to the console.</param>
/// <returns>The int value read, or the default value if unable to parse</returns>
static int ReadInt(int lastUsedValue, string prompt)
{
    // Write the prompt followed by [default value]:
    // Read the line from the input and use int.TryParse to attempt to parse it
    // If it can be parsed, write "   using value" + value to the console
    // Otherwise write "   using default value" + lastUsedValue to the console
}
```

3. Add a ReadDouble method that's exactly like ReadInt, except that **it uses double.TryParse** instead of int.TryParse. The double.TryParse method works exactly like int.TryParse, except its **`out`** variable needs to be a double, not an int.

Here are the ReadInt and ReadDouble methods that display a prompt that includes the default value, read a line from the console, try to convert it to an int or a double, and either use the converted value or the default value, writing a message to the console with the value returned.

```
static int ReadInt(int lastUsedValue, string prompt)
{
    Console.Write(prompt + " [" + lastUsedValue + "]: ");
    string line = Console.ReadLine();
    if (int.TryParse(line, out int value))
    {
        Console.WriteLine("   using value " + value);
        return value;
    } else
    {
        Console.WriteLine("   using default value " + lastUsedValue);
        return lastUsedValue;
    }
}
```

Really take some time to understand how each iteration of the `while` loop in the Main method uses fields to save the values that the user entered, then uses them for the default values in the next iteration.

```
static double ReadDouble(double lastUsedValue, string prompt)
{
    Console.Write(prompt + " [" + lastUsedValue + "]: ");
    string line = Console.ReadLine();
    if (double.TryParse(line, out double value))
    {
        Console.WriteLine("   using value " + value);
        return value;
    }
    else
    {
        Console.WriteLine("   using default value " + lastUsedValue);
        return lastUsedValue;
    }
}
```

Here's the call to double.TryParse, which works exactly like the int version except that you need to use double as the output variable type.

THANKS FOR WRITING THIS APP FOR ME! I CAN'T WAIT TO TRY IT OUT.

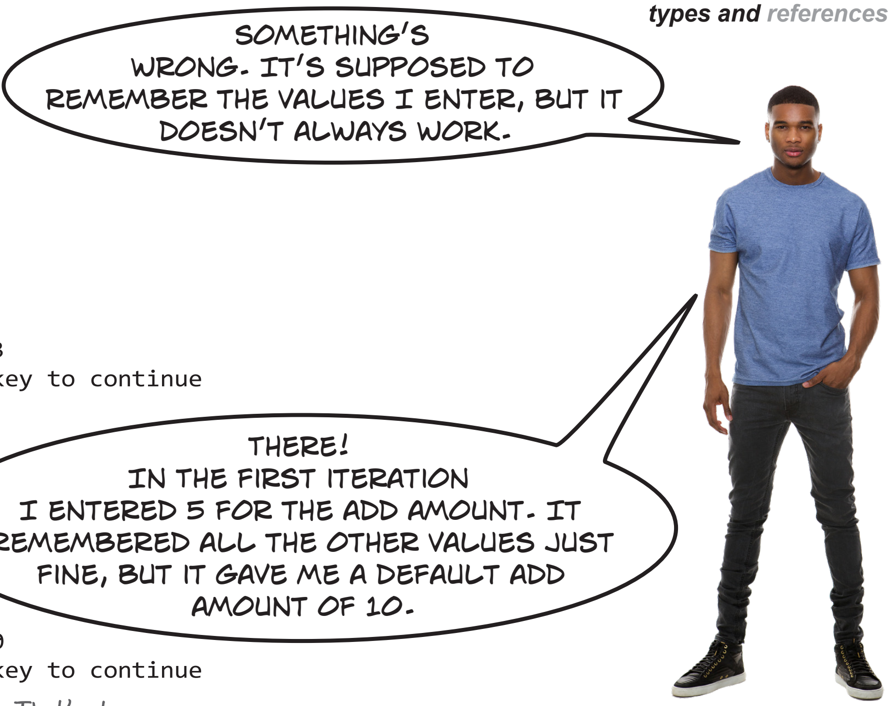
Here's the output from the app.

```
Starting 4d6 roll [14]: 18
   using value 18
Divide by [1.75]: 2.15
   using value 2.15
Add amount [2]: 5
   using value 5
Minimum [3]:
   using default value 3
Calculated ability score: 13
Press Q to quit, any other key to continue
Starting 4d6 roll [18]:
   using default value 18
Divide by [2.15]: 3.5
   using value 3.5
Add amount [13]: 5
   using value 5
Minimum [3]:
   using default value 3
Calculated ability score: 10
Press Q to quit, any other key to continue
Starting 4d6 roll [18]:
   using default value 18
Divide by [3.5]:
   using default value 3.5
Add amount [10]: 7
   using value 7
Minimum [3]:
   using default value 3
Calculated ability score: 12
Press Q to quit, any other key to continue
Starting 4d6 roll [18]:
   using default value 18
Divide by [3.5]:
   using default value 3.5
Add amount [12]: 4
   using value 4
Minimum [3]:
   using default value 3
Calculated ability score: 9
Press Q to quit, any other key to continue
Starting 4d6 roll [18]:
   using default value 18
Divide by [3.5]:
   using default value 3.5
Add amount [9]:
   using default value 9
Minimum [3]:
   using default value 3
Calculated ability score: 14
Press Q to quit, any other key to continue
```

That's strange. Owen entered 5 for the previous add amount, but the program is giving him 10 as a default option.

Again, the last amount Owen entered was 7, but it's giving 12 as a default option. Weird.

Where did this 9 number come from? Did we see it before? Can that give us a hint about what's causing this bug?


SOMETHING'S WRONG. IT'S SUPPOSED TO REMEMBER THE VALUES I ENTER, BUT IT DOESN'T ALWAYS WORK.
THERE! IN THE FIRST ITERATION I ENTERED 5 FOR THE ADD AMOUNT. IT REMEMBERED ALL THE OTHER VALUES JUST FINE, BUT IT GAVE ME A DEFAULT ADD AMOUNT OF 10.

You're right, Owen. There's a bug in the code.

Owen wants to try out different values to use in his ability score formula, so we used a loop to make the app ask for those values over and over again.

To make it easier for Owen to just change one value at a time, we included a feature in the app that remembers the last values he entered and presents them as default options. We implemented that feature by keeping an instance of the AbilityScoreCalculator class in memory, and updating its fields in each iteration of the `while` loop.

But something's gone wrong with the app. It remembers most of the values just fine, but it remembers the wrong number for the "add amount" default value. In the first iteration Owen entered 5, but it gave him 10 as a default option. Then he entered 7, but it gave a default of 12. What's going on?

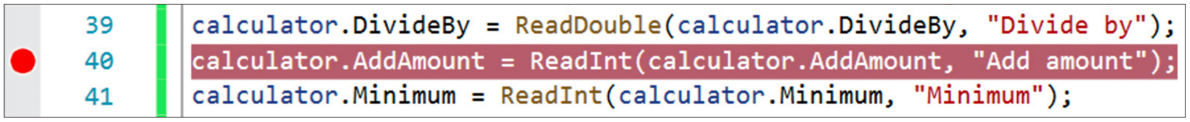
BRAIN POWER

What steps can you take to track down the bug in the ability score calculator app?

Sleuth it out

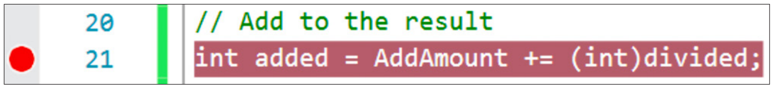
When you're debugging code, you're acting like a **code detective**. Something is causing the bug, so your job is to identify suspects and retrace their steps. Let's do an investigation and see if we can apprehend the culprit, Sherlock Holmes style.

The problem seems to be isolated to the "add amount" value, so let's start by looking for any line of code that touches the AddAmount field. Here's a line in the Main method that uses the AddAmount field—put a breakpoint on it:



```
calculator.DivideBy = ReadDouble(calculator.DivideBy, "Divide by");
calculator.AddAmount = ReadInt(calculator.AddAmount, "Add amount");
calculator.Minimum = ReadInt(calculator.Minimum, "Minimum");
```

And here's another one in the AbilityScoreCalculator.CalculateAbilityScore method—breakpoint that suspect, too:

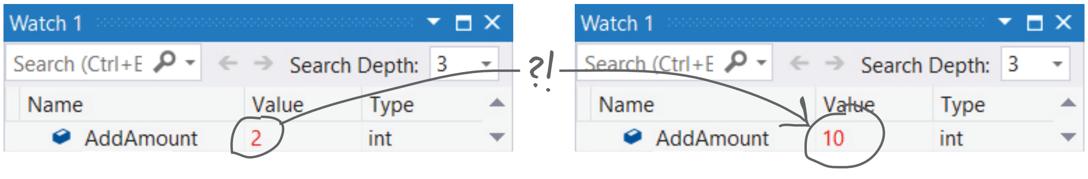


```
// Add to the result
int added = AddAmount += (int)divided;
```

This statement is meant to update the "added" variable but not change the AddAmount field.

Now run your program. When your Main method breaks, **select <u>calculator.AddAmount</u> and add a watch** (if you just right-click on AddAmount and choose "Add Watch" from the menu, it will only add a watch for AddAmount and not calculator.AddAmount). Does anything look weird there? We're not seeing anything unusual. It seems to read the value and update it just fine. OK, that's probably not the issue—you can disable or remove that breakpoint.

Continue running your program. When the breakpoint in AbilityScoreCalculator.CalculateAbilityScore is hit, **add a watch for AddAmount**. According to Owen's formula, this line of code is supposed to add AddAmount to the result of dividing the roll result. Now **step over** the statement and...



Name	Value	Type
AddAmount	2	int

Name	Value	Type
AddAmount	10	int

Wait, what?! AddAmount changed. But...but that's not supposed to happen—it's impossible! Right? As Sherlock Holmes said, "When you have eliminated the impossible, whatever remains, however improbable, must be the truth."

It looks like we've sleuthed out the source of the problem. That statement is supposed to cast `divided` to an int to round it down to an integer, then add it to AddAmount and store the result in `added`. It also has an unexpected side effect: it's updating AddAmount with the sum because **the statement uses the += operator**, which returns the sum but assigns the sum to AddAmount.

And now we can finally fix Owen's bug

Now that you know what's happening, you can **fix the bug**—and it turns out to be a pretty small change. You just need to <u>change the statement to use + instead of +=</u>:

```
int added = AddAmount + (int)divided;
```

Change the += to a + to keep this line of code from updating the "added" variable and fix the bug. Like Sherlock would say, "It's elementary."

Now that we've found the problem, we can finally give the "Sharpen your pencil" solution.

Sharpen your pencil Solution

We've built a class to help Owen calculate ability scores. To use it, you set its Starting4D6Roll, DivideBy, SubtractBy, and Minimum fields—or just leave the values set in their declarations—and call its CalculateAbilityScore method. Unfortunately, **there's one line of code that has problems**. Circle the line of code with problems and write down what's wrong with it.

```
int added = AddAmount += divided;
```

After you **circle the line of code that has problems**, write down the problems that you found with it.

First, it won't compile because AddAmount += divided is a double, so a cast needs to happen to assign it to an int. Second, it uses += and not +, which causes the line to update AddAmount.

there are no Dumb Questions

Q: I'm still not clear on the difference between the + operator and the += operator. How do they work, and why would I use one and not the other?

A: There are several operators that you can combine with an equals sign. They include += for adding, -= for subtracting, /= for dividing, *= for multiplying, and %= for remainder. Operators like + that combine two values are called **binary operators**. Some people find this name a little confusing, but "binary" refers to the fact that the operator combines two values—"binary" means "involving two things"—not that it somehow operates only on binary numbers.

With binary operators, you can do something called **compound assignment**, which means instead of this:

```
a = a + c;
```

you can do this:

```
a += c;
```

The += operator tells C# to add a + c and then store the result in a.

and it means the same thing. The compound assignment `x op= y` is equivalent to `x = x op y` (that's the technical way of explaining it). They do exactly the same thing.

Operators like += or *= that combine a binary operator with an equals sign are called compound assignment operators.

Q: But then how did the added variable get updated?

A: What caused confusion in the score calculator is that the **assignment operator = also returns a value**. You can do this:

```
int q = (a = b + c)
```

which will calculate `a = b + c` as usual. The = operator ***returns*** a value, so **it will update q with the result** as well. So:

```
int added = AddAmount += divided;
```

is just like doing this:

```
int added = (AddAmount = AddAmount + divided);
```

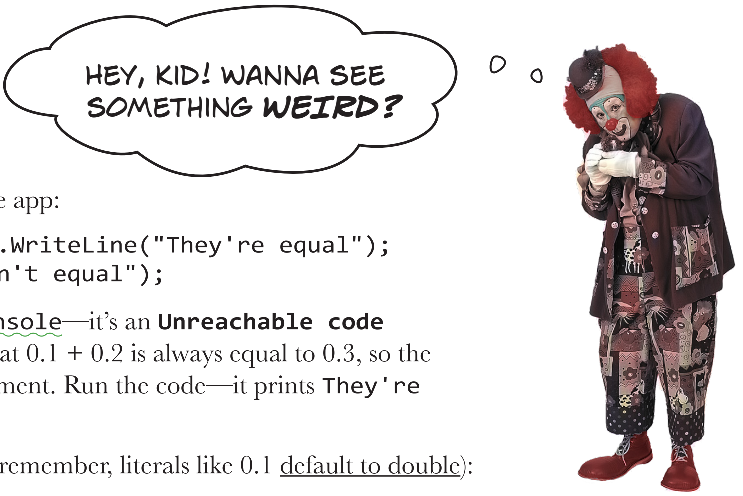
which causes `AddAmount` to be increased by `divided`, but stores that result in `added` as well.

Q: Wait, what? The equals operator returns a value?

A: Yes, = returns the value being set. So in this code:

```
int first;
int second = (first = 4);
```

both `first` and `second` will end up equal to 4. Open up a console app and use the debugger to test this. It really works!



Try this!

Try adding this `if/else` statement to a console app:

```
if (0.1M + 0.2M == 0.3M) Console.WriteLine("They're equal");
else Console.WriteLine("They aren't equal");
```

You'll see a green squiggle under the second `Console`—it's an **`Unreachable code detected`** warning. The C# compiler knows that 0.1 + 0.2 is always equal to 0.3, so the code will never reach the `else` part of the statement. Run the code—it prints `They're equal` to the console.

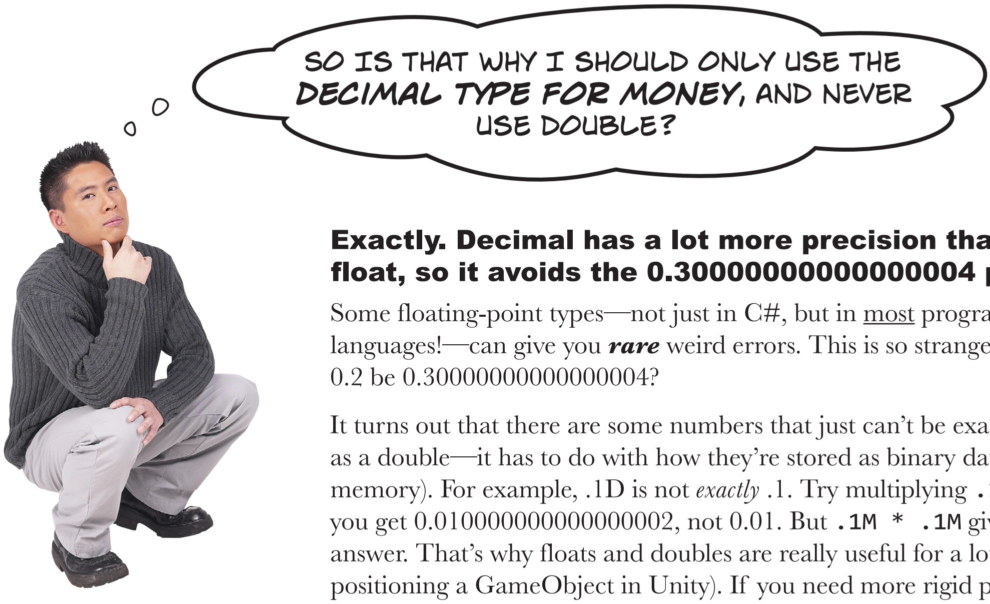
Next, **change the float literals to doubles** (remember, literals like 0.1 default to double):

```
if (0.1 + 0.2 == 0.3) Console.WriteLine("They're equal");
else Console.WriteLine("They aren't equal");
```

That's really strange. The warning moved to the first line of the `if` statement. Try running the program. Hold on, that can't be right! It printed `They aren't equal` to the console. How is 0.1 + 0.2 not equal to 0.3?

Now do one more thing. Change 0.3 to 0.30000000000000004 (with 15 zeros between the 3 and 4). Now it prints `They're equal` again. So apparently 0.1D plus 0.2D equals 0.30000000000000004D.

Wait, what?!


SO IS THAT WHY I SHOULD ONLY USE THE DECIMAL TYPE FOR MONEY, AND NEVER USE DOUBLE?

Exactly. Decimal has a lot more precision than double or float, so it avoids the 0.30000000000000004 problem.

Some floating-point types—not just in C#, but in most programming languages!—can give you ***rare*** weird errors. This is so strange! How can 0.1 + 0.2 be 0.30000000000000004?

It turns out that there are some numbers that just can't be exactly represented as a double—it has to do with how they're stored as binary data (0s and 1s in memory). For example, .1D is not *exactly* .1. Try multiplying `.1D * .1D`—you get 0.010000000000000002, not 0.01. But `.1M * .1M` gives you the right answer. That's why floats and doubles are really useful for a lot of things (like positioning a GameObject in Unity). If you need more rigid precision—like for a financial app that deals with money—decimal is the way to go.

there are no Dumb Questions

Q: I'm still not clear on the difference between conversion and casting. Can you explain it a little more clearly?

A: Conversion is a general, all-purpose term for converting data from one type to another. Casting is a much more specific operation, with explicit rules about which types can be cast to other types, and what to do when the data for the value from one doesn't quite match the type it's being cast to. You just saw an example of one of those rules—when a floating-point number is cast to an int, it's rounded down by dropping any decimal value. You saw another rule earlier about wrapping for integer types, where a number that's too big to fit into the type it's being cast to is wrapped using the remainder operator.

Q: Hold on a minute. Earlier you had me "wrap" numbers myself using the mod function on my calculator app. Now you're talking about remainders. What's the difference?

A: Mod and remainder are very similar operations. For positive numbers they're exactly the same: A % B is the remainder when you divide B into A, so: 5 % 2 is the remainder of 5 ÷ 2, or 1. (If you're trying to remember how long division works, that just means that 5 ÷ 2 is equal to 2 × 2 + 1, so the rounded quotient is 2 and the remainder is 1.) But when you start dealing with negative numbers, there's a difference between mod (or modulus) and remainder. You can see for yourself: your calculator will tell you that –397 mod 17 = 11, but if you use the C# remainder operator you'll get –397 % 17 = –6.

Q: Owen's formula had me dividing two values and then rounding the result down to the nearest integer. How does that fit in with casting?

A: Let's say you have some floating-point values:

```
float f1 = 185.26F;
double d2 = .0000316D;
decimal m3 = 37.26M;
```

and you want to cast them to int values so you can assign them to int variables `i1`, `i2`, and `i3`. We know that those int variables can only hold integers, so your program needs to do *something* to the decimal part of the number.

So C# has a consistent rule: it drops the decimal and rounds down: `f1` becomes 185, `d2` becomes 0, and `m3` becomes 37. But don't take our word for it—write your own C# code that casts those three floating-point values to int to see what happens.

There's a whole web page dedicated to the 0.30000000000000004 problem! Check out https://0.30000000000000004.com to see examples in a lot of different languages.

The 0.1D + 0.2D != 0.3D example is an edge case, or a problem or situation that only happens under certain rare conditions, usually when a parameter is at one of its extremes (like a very big or very small number). If you want to learn more about it, there's a great article by Jon Skeet about how floating-point numbers are stored in memory in .NET. You can read it here: https://csharpindepth.com/Articles/FloatingPoint.

Jon gave us some amazing technical review feedback for the 1st edition of this book that made a huge difference for us. Thanks so much, Jon!

Use reference variables to access your objects

When you create a new object, you use a `new` statement to instantiate it, like `new Guy()` in your program at the end of the last chapter—the `new` statement created a new Guy object on the heap. You still needed a way to *access* that object, and that's where a variable like `joe` came in: `Guy joe = new Guy()`. Let's dig a little deeper into exactly what's going on there.

The `new` statement creates the instance, but just creating that instance isn't enough. ***You need a reference to the object***. So you created a **reference variable**: a variable of type Guy with a name, like `joe`. So `joe` is a reference to the new Guy object you created. Any time you want to use that particular Guy, you can reference it with the reference variable called `joe`.

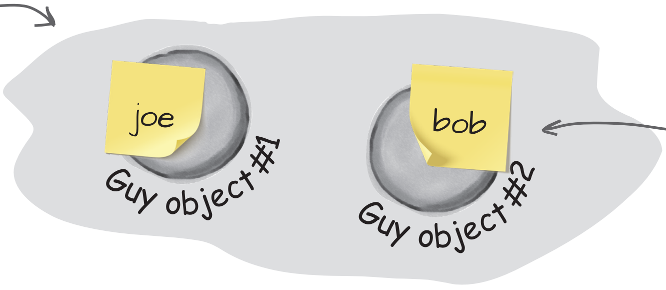
When you have a variable that's an object type, it's a reference variable: a reference to a particular object. Let's just make sure we get the terminology right since we'll be using it a lot. We'll use the first two lines of the "Joe and Bob" program from the last chapter:

Creating a reference is like writing a name on a sticky note and sticking it to the object. You're using it to label an object so you can refer to it later.

```
static void Main(string[] args)
{
    Guy joe = new Guy() { Cash = 50, Name = "Joe" };
    Guy bob = new Guy() { Cash = 100, Name = "Bob" };
```

This is the reference variable.

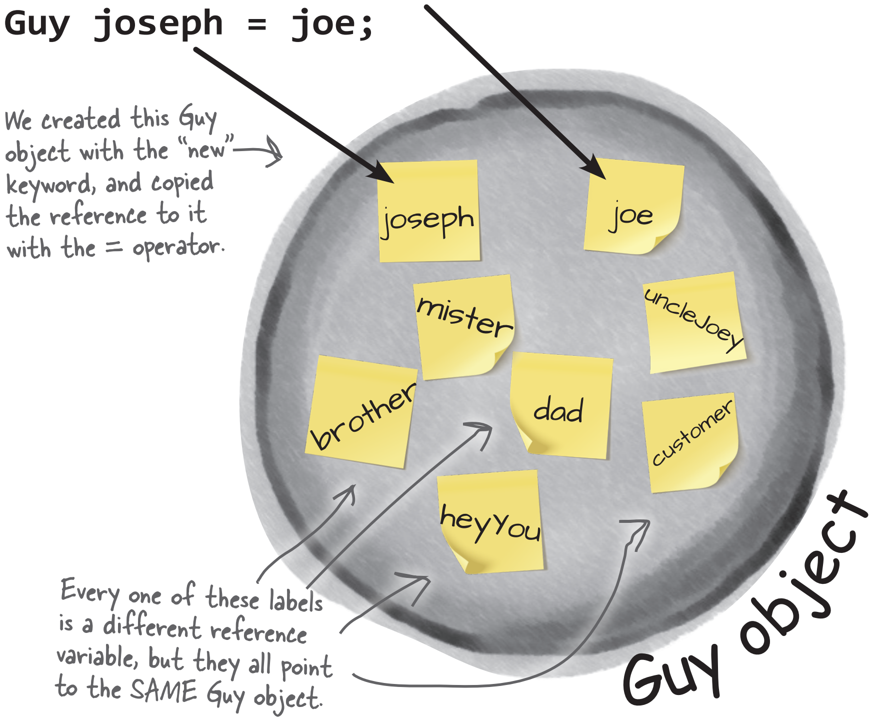
This creates the object that it will refer to.


And here's the heap after this code runs. It has two objects, with the variable "joe" referring to one object and the variable "bob" referring to the other one.
joe
Guy object #1
bob
Guy object #2
The ONLY way to reference this Guy object is through the reference variable called "bob".

References are like sticky notes for your objects

In your kitchen, you probably have containers of salt and sugar. If you switched their labels, it would make for a pretty disgusting meal—even though you changed the labels, the contents of the containers stayed the same. ***References are like labels.*** You can move labels around and point them at different things, but it's the **object** that dictates what methods and data are available, not the reference itself—and you can **copy references** just like you copy values.

```
Guy joe = new Guy();
Guy joseph = joe;
```


We created this Guy object with the "new" keyword, and copied the reference to it with the = operator.
joseph
joe
mister
uncleJoey
brother
dad
customer
heyYou
Every one of these labels is a different reference variable, but they all point to the SAME Guy object.
Guy object

A reference is like a label that your code uses to talk about a specific object. You use it to access fields and call methods on an object that it points to.

We stuck a lot of sticky notes on that object! In this particular case, there are a lot of different references to this same Guy object—because a lot of different methods use it for different things. Each reference has a different name that makes sense in its context.

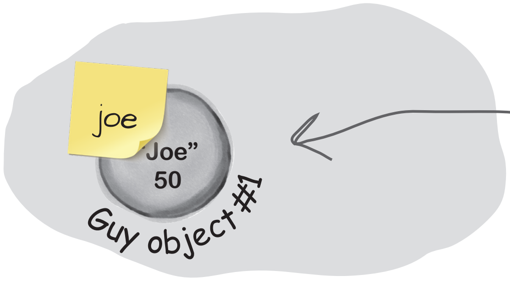
That's why it can be really useful to have ***multiple references pointing to the same instance***. So you could say `Guy dad = joe`, and then call `dad.GiveCash()` (that's what Joe's kid does every day). If you want to write code that works with an object, you need a reference to that object. If you don't have that reference, you have no way to access the object.

If there aren't any more references, your object gets garbage-collected

If all of the labels come off of an object, programs can no longer access that object. That means C# can mark the object for **garbage collection**. That's when C# gets rid of any unreferenced objects and reclaims the memory those objects took up for your program's use.

1. **Here's some code that creates an object.**
Just to recap what we've been talking about: when you use the `new` statement, you're telling C# to create an object. When you take a reference variable like `joe` and assign it to that object, it's like you're slapping a new sticky note on it.

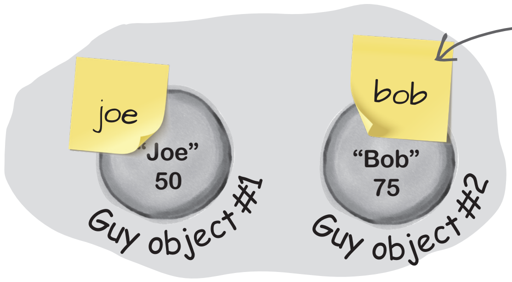
```
Guy joe = new Guy() { Cash = 50, Name = "Joe" };
```



We used an object initializer to create this Guy object. Its Name field has the string "Joe", its Cash field has the int 50, and we put a reference to the object in a variable called "joe".

2. **Now let's create our second object.**
Once we do this we'll have two Guy object instances and two reference variables: one variable (`joe`) for the first Guy object, and another variable (`bob`) for the second.

```
Guy bob = new Guy() { Cash = 100, Name = "Bob" };
```



We created another Guy object and created a variable called "bob" that points to it. Variables are like sticky notes—they're just labels that you can "stick" to any object.

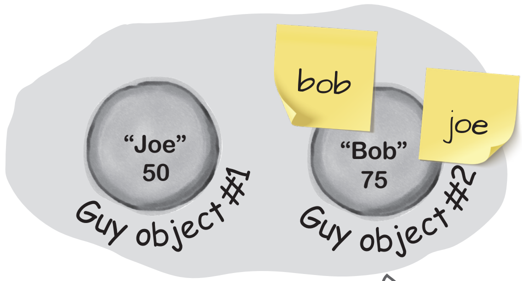
3 **Let's take the reference to the first Guy object and change it to point to the second Guy object.**

Take a really close look at what you're doing when you create a new Guy object. You're taking a variable and using the = assignment operator to set it—in this case, to a reference that's returned by the `new` statement. That assignment works because **you can copy a reference just like you copy a value**.

So let's go ahead and copy that value:

```
joe = bob;
```

That tells C# to take make `joe` point to the same object that `bob` does. Now the `joe` and `bob` variables **both point to the same object**.

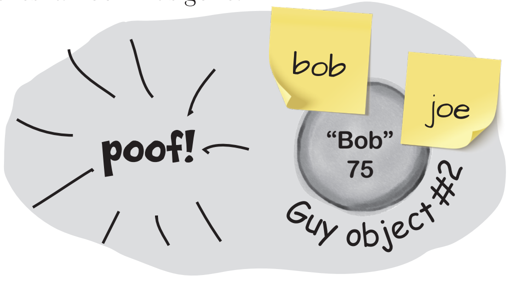


After the CLR (coming up in "Garbage Collection Exposed" interview!) removes the last reference to the object, it marks it for garbage collection.

4 **There's no longer a reference to the first Guy object...so it gets garbage-collected.**

Now that `joe` is pointing to the same object as `bob`, there's no longer a reference to the Guy object it used to point to. So what happens? C# marks the object for garbage collection, and ***eventually*** trashes it. Poof—it's gone!

The CLR keeps track of all of the references to each object, and when the last reference disappears it marks it for removal. But it might have other things to do right now, so the object could stick around for a few milliseconds—or even longer!

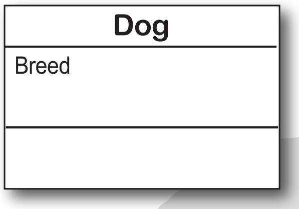


For an object to stay in the heap, it has to be referenced. Some time after the last reference to the object disappears, so does the object.

```
public partial class Dog {
  public void GetPet() {
    Console.WriteLine("Woof!");
  }
}
```

Multiple references and their side effects

You've got to be careful when you start moving reference variables around. Lots of times, it might seem like you're simply pointing a variable to a different object. You could end up removing all references to another object in the process. That's not a bad thing, but it may not be what you intended. Take a look:

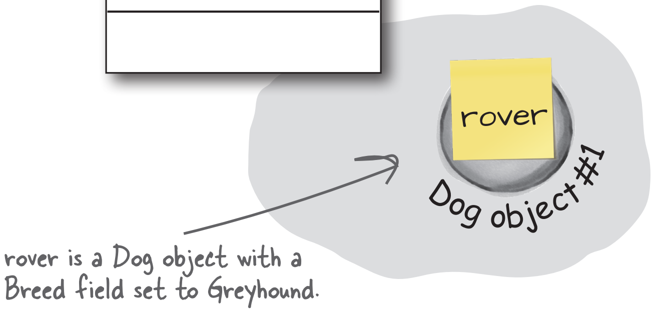


1

```
Dog rover = new Dog();
rover.Breed = "Greyhound";
```

Objects: 1

References: 1

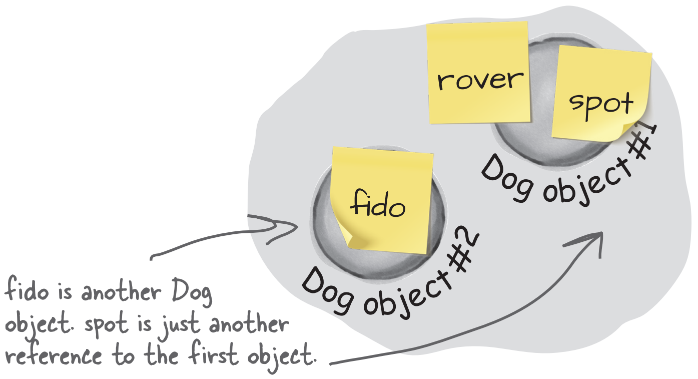


2

```
Dog fido = new Dog();
fido.Breed = "Beagle";
Dog spot = rover;
```

Objects: 2

References: 3

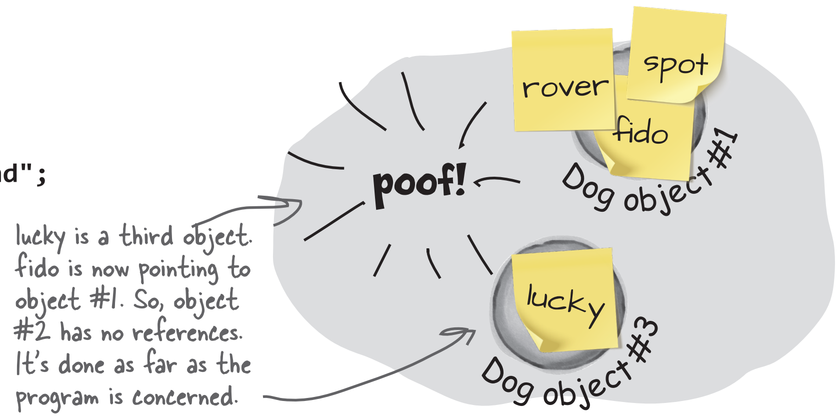

rover
spot
Dog object #1
fido
Dog object #2
fido is another Dog object. spot is just another reference to the first object.

3

```
Dog lucky = new Dog();
lucky.Breed = "Dachshund";
fido = rover;
```

Objects: 2

References: 4


rover
spot
fido
Dog object #1
poof!
lucky
Dog object #3
lucky is a third object. fido is now pointing to object #1. So, object #2 has no references. It's done as far as the program is concerned.

Sharpen your pencil

Now it's your turn. Here's one long block of code. Figure out how many objects and references there are at each stage. On the right-hand side, draw a picture of the objects and sticky notes in the heap.

1

```
Dog rover = new Dog();
rover.Breed = "Greyhound";
Dog rinTinTin = new Dog();
Dog fido = new Dog();
Dog greta = fido;
```

Objects:______

References:_____

2

```
Dog spot = new Dog();
spot.Breed = "Dachshund";
spot = rover;
```

Objects:______

References:_____

3

```
Dog lucky = new Dog();
lucky.Breed = "Beagle";
Dog charlie = fido;
fido = rover;
```

Objects:______

References:_____

4

```
rinTinTin = lucky;
Dog laverne = new Dog();
laverne.Breed = "pug";
```

Objects:______

References:_____

5

```
charlie = laverne;
lucky = rinTinTin;
```

Objects:______

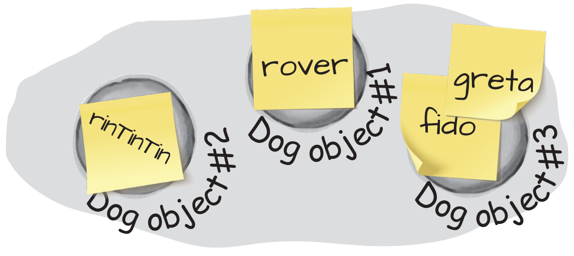
References:_____

Sharpen your pencil Solution

1
```
Dog rover = new Dog();
rover.Breed = "Greyhound";
Dog rinTinTin = new Dog();
Dog fido = new Dog();
Dog greta = fido;
```

Objects: 3

References: 4

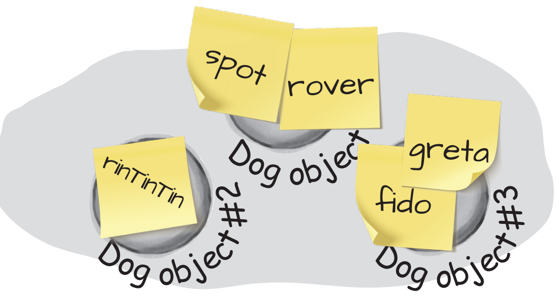


One new Dog object is created, but spot is the only reference to it. When spot is set to rover, that object goes away.

2
```
Dog spot = new Dog();
spot.Breed = "Dachshund";
spot = rover;
```

Objects: 3

References: 5

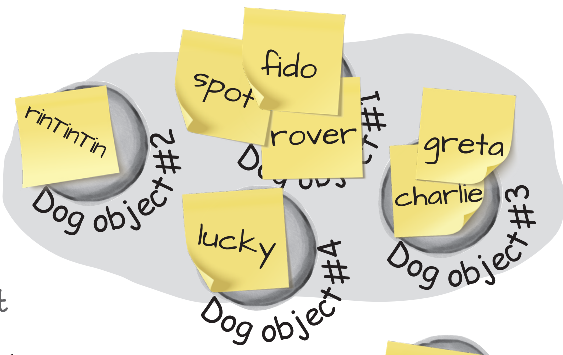


3
```
Dog lucky = new Dog();
lucky.Breed = "Beagle";
Dog charlie = fido;
fido = rover;
```

charlie was set to fido when fido was still on object #3. Then, after that, fido moved to object #1, leaving charlie behind.

Objects: 4

References: 7



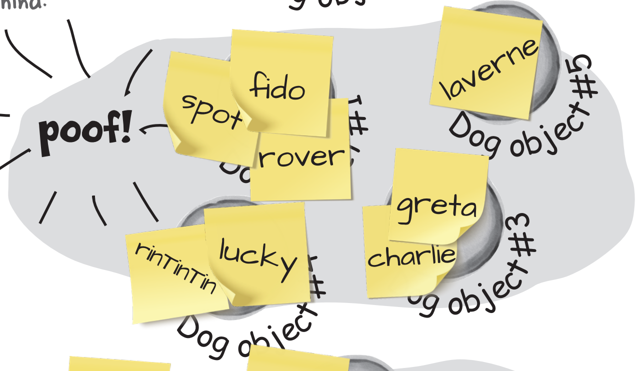
Dog #2 lost its last reference, and it went away.

4
```
rinTinTin = lucky;
Dog laverne = new Dog();
laverne.Breed = "pug";
```

Objects: 4

References: 8

When rinTinTin moved to lucky's object, the old rinTinTin object disappeared.

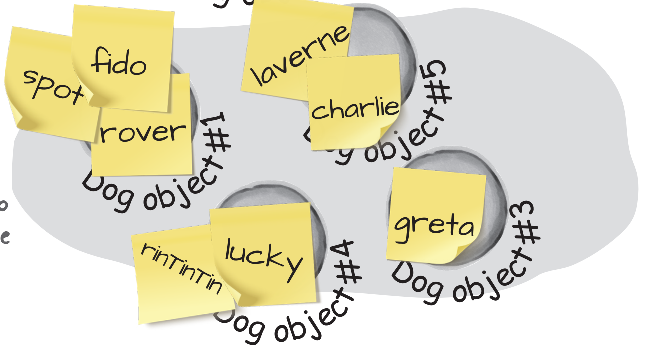


5
```
charlie = laverne;
lucky = rinTinTin;
```

Objects: 4

References: 8

Here the references move around, but no new objects are created. Setting lucky to rinTinTin did nothing because they already pointed to the same object.



Garbage Collection Exposed

This week's interview:
The .NET Common Language Runtime

Head First: So, we understand that you do a pretty important job for us. Can you tell us a little more about what you do?

Common Language Runtime (CLR): In a lot of ways, it's pretty simple. I run your code. Any time you're using a .NET app, I'm making it work.

Head First: What do you mean by making it work?

CLR: I take care of the low-level "stuff" for you by doing a sort of "translation" between your program and the computer running it. When you talk about instantiating objects or doing garbage collection, I'm the one that's managing all of those things.

Head First: So how does that work, exactly?

CLR: Well, when you run a program on Windows, Linux, macOS, or most other operating systems, the OS loads machine language from a binary.

Head First: I'm going to stop you right there. Can you back up and tell us what machine language is?

CLR: Sure. A program written in machine language is made up of code that's executed directly by the CPU—and it's a whole lot less readable than C#.

Head First: If the CPU is executing the actual machine code, what does the OS do?

CLR: The OS makes sure each program gets its own process, respects the system's security rules, and provides APIs.

Head First: And for our readers who don't know what an API is?

CLR: An API—or application programming interface—is a set of methods provided by an OS, library, or program. OS APIs help you do things like work with the filesystem and interact with hardware. But they're often pretty difficult to use—especially for memory management—and they vary from OS to OS.

Head First: So back to your job. You mentioned a binary. What exactly is that?

CLR: A binary is a file that's (usually) created by a **compiler**, a program whose job it is to convert high-level language into low-level code like machine code. Windows binaries usually end with *.exe* or *.dll*.

Head First: But I'm guessing that there's a twist here. You said "low-level code like machine code"—does that mean there are other kinds of low-level code?

CLR: Exactly. I don't run the same machine language as the CPU. When you build your C# code, Visual Studio asks the C# compiler to create **Common Intermediate Language (CIL)**. That's what I run. C# code is turned into CIL, which I read and execute.

Head First: You mentioned managing memory. Is that where garbage collection fits into all of this?

CLR: Yes! One really, really useful thing that I do for you is that I tightly manage your computer's memory by figuring out when your program's finished with certain objects. When it is, I get rid of them for you to free up that memory. That's something programmers used to have to do themselves—but thanks to me, it's something that you don't have to be bothered with. You might not have known it at the time, but I've been making your job of learning C# a whole lot easier.

Head First: You mentioned Windows binaries. What if I'm running .NET programs on Mac or Linux? Are you doing the same thing for those OSs?

CLR: If you're using a macOS or Linux—or running Mono on Windows— then you're technically not using me. You're using my cousin, the Mono Runtime, which implements the same *ECMA Common Language Infrastructure (CLI)* that I do. So when it comes to all the stuff I've talked about so far, we both do exactly the same thing.

Create a program with an Elephant class. Instantiate two Elephant instances and then swap the reference values that point to them, ***without*** getting any Elephant instances garbage-collected. Here's what it will look like when your program runs.

You're going to build a new console app that has a class called Elephant.

Here's an example of the output of the program:

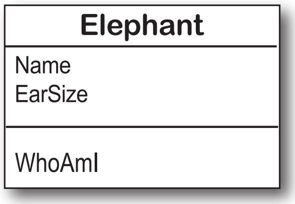
```
Press 1 for Lloyd, 2 for Lucinda, 3 to swap
You pressed 1
Calling lloyd.WhoAmI()
My name is Lloyd.
My ears are 40 inches tall.

You pressed 2
Calling lucinda.WhoAmI()
My name is Lucinda.
My ears are 33 inches tall.

You pressed 3
References have been swapped

You pressed 1
Calling lloyd.WhoAmI()
My name is Lucinda.
My ears are 33 inches tall.

You pressed 2
Calling lucinda.WhoAmI()
My name is Lloyd.
My ears are 40 inches tall.

You pressed 3
References have been swapped

You pressed 1
Calling lloyd.WhoAmI()
My name is Lloyd.
My ears are 40 inches tall.

You pressed 2
Calling lucinda.WhoAmI()
My name is Lucinda.
My ears are 33 inches tall.
```

The Elephant class has a WhoAmI method that writes these two lines to the console to display the values in the Name and EarSize fields.

Swapping the references causes the lloyd variable to call the Lucinda object's method, and vice versa.

Swapping them again returns things to the way they were when the program started.

Here's the class diagram for the Elephant class you'll need to create.



The CLR garbage-collects any object with no references to it. So here's a hint for this exercise: if you want to pour a cup of coffee into another cup that's currently full of tea, you'll need a third glass to pour the tea into...

Your job is to create a .NET Core console app with an Elephant class that matches the class diagram and uses its fields and methods to generate output that matches the example output.

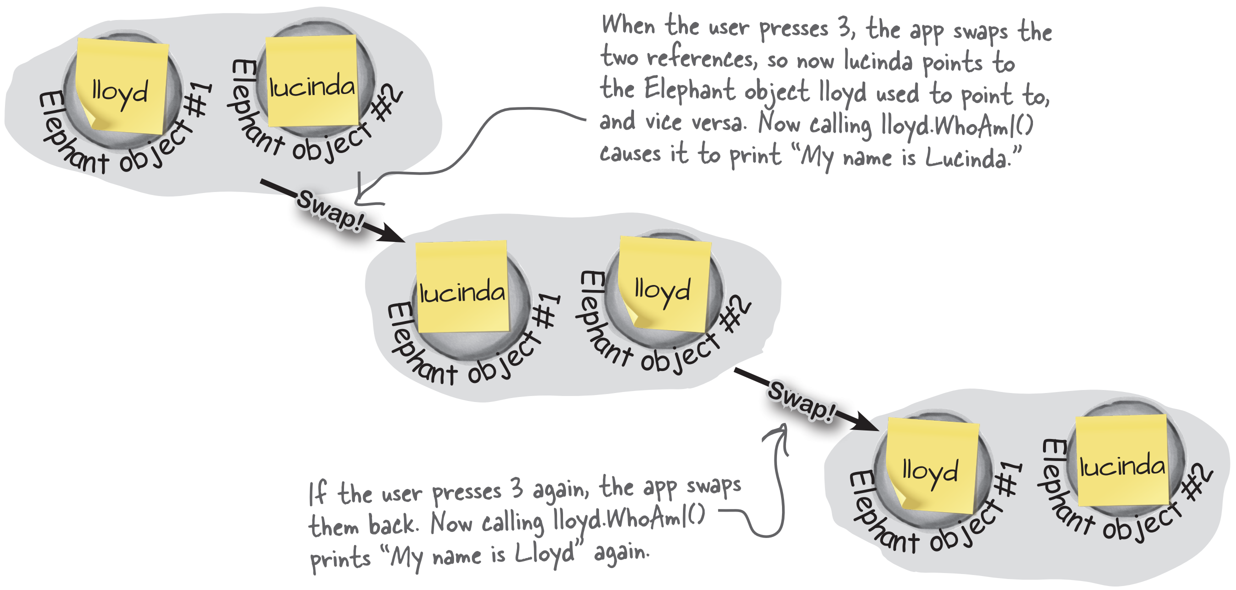
1 **Create a new .NET Core console app and add the Elephant class.**
Add an Elephant class to the project. Have a look at the Elephant class diagram—you'll need an int field called EarSize and a string field called Name. Add them, and make sure both are public. Then add a method called WhoAmI that writes two lines to the console to tell you the name and ear size of the elephant. Look at the example output to see exactly what it's supposed to print.

2 **Create two Elephant instances and a reference.**
Use object initializers to instantiate two Elephant objects:

```
Elephant lucinda = new Elephant() { Name = "Lucinda", EarSize = 33 };
Elephant lloyd = new Elephant() { Name = "Lloyd", EarSize = 40 };
```

3 **Call their WhoAmI methods.**
When the user presses 1, call lloyd.WhoAmI. When the user presses 2, call lucinda.WhoAmI. Make sure that the output matches the example.

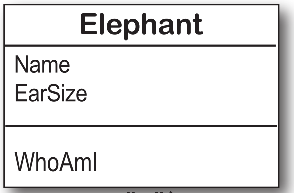
4 **Now for the fun part: swap the references.**
Here's the interesting part of this exercise. When the user presses 3, make the app call a method that ***exchanges the two references***. You'll need to write that method. After you swap references, pressing 1 should write Lucinda's message to the console, and pressing 2 should write Lloyd's message. If you swap the references again, everything should go back to normal.


lloyd
Elephant object #1
lucinda
Elephant object #2
Swap!
lucinda
Elephant object #1
lloyd
Elephant object #2
Swap!
lloyd
Elephant object #1
lucinda
Elephant object #2
When the user presses 3, the app swaps the two references, so now lucinda points to the Elephant object lloyd used to point to, and vice versa. Now calling lloyd.WhoAmI() causes it to print "My name is Lucinda."
If the user presses 3 again, the app swaps them back. Now calling lloyd.WhoAmI() prints "My name is Lloyd" again.

Exercise Solution

Create a program with an Elephant class. Instantiate two Elephant instances and then swap the reference values that point to them, ***without*** getting any Elephant instances garbage-collected.

Here's the Elephant class:



```
class Elephant
{
    public int EarSize;
    public string Name;
    public void WhoAmI()
    {
        Console.WriteLine("My name is " + Name + ".");
        Console.WriteLine("My ears are " + EarSize + " inches tall.");
    }
}
```

Here's the Main method inside the Program class:

```
static void Main(string[] args)
{
    Elephant lucinda = new Elephant() { Name = "Lucinda", EarSize = 33 };
    Elephant lloyd = new Elephant() { Name = "Lloyd", EarSize = 40 };

    Console.WriteLine("Press 1 for Lloyd, 2 for Lucinda, 3 to swap");
    while (true)
    {
        char input = Console.ReadKey(true).KeyChar;
        Console.WriteLine("You pressed " + input);
        if (input == '1')
        {
            Console.WriteLine("Calling lloyd.WhoAmI()");
            lloyd.WhoAmI();
        } else if (input == '2')
        {
            Console.WriteLine("Calling lucinda.WhoAmI()");
            lucinda.WhoAmI();
        } else if (input == '3')
        {
            Elephant holder;
            holder = lloyd;
            lloyd = lucinda;
            lucinda = holder;
            Console.WriteLine("References have been swapped");
        }
        else return;
        Console.WriteLine();
    }
}
```

If you just point Lloyd to Lucinda, there won't be any more references pointing to Lloyd, and his object will be lost. That's why you need to have an extra variable (we called it "holder") to keep track of the Lloyd object reference until Lucinda can get there.

There's no "new" statement when we declare the "holder" variable because we don't want to create another instance of Elephant.

Two references mean TWO variables that can change the same object's data

Besides losing all the references to an object, when you have multiple references to an object, you can unintentionally change the object. In other words, one reference to an object may ***change*** that object, while another reference to that object has ***no idea*** that something has changed. Let's see how that works.

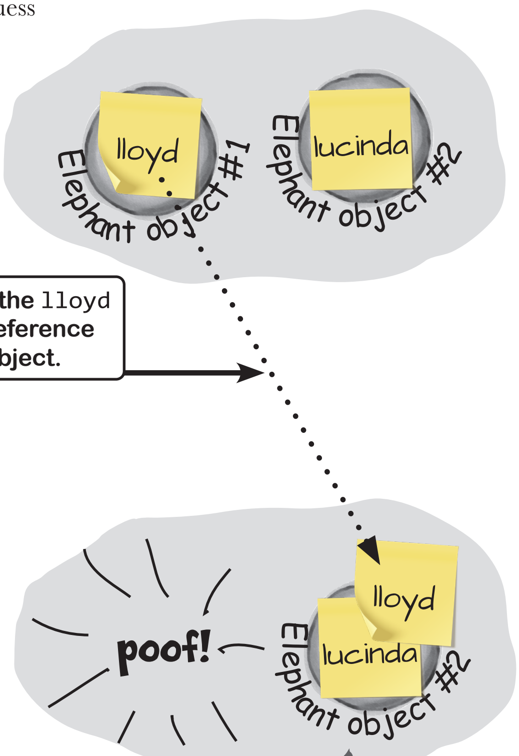
Do this!

Add one more "else if" block to your Main method. Can you guess what will happen once it runs?

```
else if (input == '3')
{
    Elephant holder;
    holder = lloyd;
    lloyd = lucinda;
    lucinda = holder;
    Console.WriteLine("References have been swapped");
}
else if (input == '4')
{
    lloyd = lucinda;
    lloyd.EarSize = 4321;
    lloyd.WhoAmI();
}
else
{
    return;
}
```

After this statement, both the `lloyd` and `lucinda` variables reference the SAME Elephant object.

This statement says to set EarSize to 4321 on whatever object the reference stored in the `lloyd` variable happens to point to.



Swapping these two sticky notes won't change anything because they're stuck to the same object.

And since the lloyd reference is no longer pointing to the first Elephant object, it gets garbage-collected... and there's no way to bring it back!

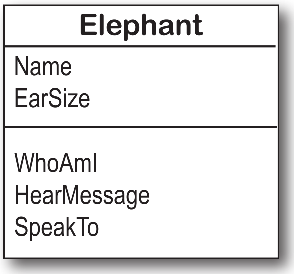
Now go ahead and run your program. Here's what you'll see:

```
You pressed 4
My name is Lucinda
My ears are 4321 inches tall.

You pressed 1
Calling lloyd.WhoAmI()
My name is Lucinda
My ears are 4321 inches tall.

You pressed 2
Calling lucinda.WhoAmI()
My name is Lucinda
My ears are 4321 inches tall.
```

The program acts normally... until you press 4. Once you do that, pressing either 1 or 2 prints the same output—and pressing 3 to swap the references doesn't do anything anymore.

After you press 4 and run the new code that you added, both the `lloyd` and lucinda variables **contain the same reference** to the second Elephant object. Pressing 1 to call lloyd.WhoAmI prints exactly the same message as pressing 2 to call `lucinda`.WhoAmI. Swapping them makes no difference because you're swapping two identical references.

Objects use references to talk to each other



So far, you've seen forms talk to objects by using reference variables to call their methods and check their fields. Objects can call one another's methods using references, too. In fact, there's nothing that a form can do that your objects can't do, because **your form is just another object**. When objects talk to each other, one useful keyword that they have is `this`. Any time an object uses the `this` keyword, it's referring to itself—it's a reference that points to the object that calls it. Let's see what that looks like by modifying the Elephant class so instances can call each other's methods.

1 Add a method that lets an Elephant hear a message.

Let's add a method to the Elephant class. Its first parameter is a message from another Elephant object. Its second parameter is the Elephant object that sent the message:

Do this

```
public void HearMessage(string message, Elephant whoSaidIt) {
    Console.WriteLine(Name + " heard a message");
    Console.WriteLine(whoSaidIt.Name + " said this: " + message);
}
```

Here's what it looks like when it's called:

```
lloyd.HearMessage("Hi", lucinda);
```

We called `lloyd's` HearMessage method, and passed it two parameters: the string `"Hi"` and a reference to Lucinda's object. The method uses its `whoSaidIt` parameter to access the Name field of whatever elephant was passed in.

2 Add a method that lets an Elephant send a message.

Now let's add a SpeakTo method to the Elephant class. It uses a special keyword: **`this`**. That's a reference that **lets an object get a reference to itself**.

```
public void SpeakTo(Elephant whoToTalkTo, string message) {
    whoToTalkTo.HearMessage(message, this);
}
```

An Elephant's SpeakTo method uses the "this" keyword to send a reference to itself to another Elephant.

Let's take a closer look at what's going on.

When we call the Lucinda object's SpeakTo method:

```
lucinda.SpeakTo(lloyd, "Hi, Lloyd!");
```

It calls the Lloyd object's HearMessage method like this:

```
whoToTalkTo.HearMessage("Hi, Lloyd!", this);
```

Lucinda uses `whoToTalkTo` (which has a reference to Lloyd) to call `HearMessage`.

`this` is replaced with a reference to Lucinda's object.

```
[a reference to Lloyd].HearMessage("Hi, Lloyd!", [a reference to Lucinda]);
```

3 Call the new methods.

Add one more `else if` block to the Main method to make the Lucinda object send a message to the Lloyd object:

```
else if (input == '4')
{
    lloyd = lucinda;
    lloyd.EarSize = 4321;
    lloyd.WhoAmI();
}
else if (input == '5')
{
    lucinda.SpeakTo(lloyd, "Hi, Lloyd!");
}
else
{
    return;
}
```

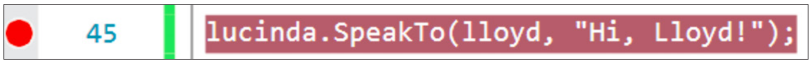
Now run your program and press 5. You should see this output:

```
You pressed 5
Lloyd heard a message
Lucinda said this: Hi, Lloyd!
```

The "this" keyword lets an object get a reference to itself.

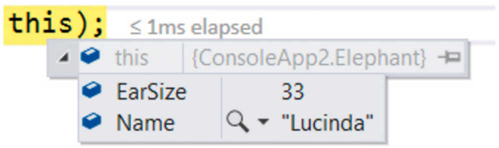
4 Use the debugger to understand what's going on.

Place a breakpoint on the statement that you just added to the Main method:



```
lucinda.SpeakTo(lloyd, "Hi, Lloyd!");
```

1. Run your program and press 5.
2. When it hits the breakpoint, use Debug >> Step Into (F11) to step into the SpeakTo method.
3. Add a watch for Name to show you which Elephant object you're inside. You're currently inside the Lucinda object—which makes sense because the Main method called lucinda.SpeakTo.
4. Hover over the **`this`** keyword at the end of the line and expand it. It's a reference to the Lucinda object.



```
this);  ≤ 1ms elapsed
  this       {ConsoleApp2.Elephant}
    EarSize     33
    Name        "Lucinda"
```

   Hover over **`whoToTalkTo`** and expand it—it's a reference to the Lloyd object.

5. The SpeakTo method has one statement—it calls whoToTalkTo.HearMessage. Step into it.
6. You should now be inside the HearMessage method. Check your watch again—now the value of the Name field is "Lloyd"—the Lucinda object called the Lloyd object's HearMessage method.
7. Hover over **`whoSaidIt`** and expand it. It's a reference to the Lucinda object.

Finish stepping through the code. Take a few minutes to really understand what's going on.

Arrays hold multiple values

Strings and arrays are different from the other data types you've seen in this chapter because they're the only ones without a set size (think about that for a bit).

If you have to keep track of a lot of data of the same type, like a list of prices or a group of dogs, you can do it in an **array**. What makes an array special is that it's a **group of variables** that's treated as one object. An array gives you a way of storing and changing more than one piece of data without having to keep track of each variable individually. When you create an array, you declare it just like any other variable, with a name and a type—except **the type is followed by square brackets**:

```
bool[] myArray;
```

Use the `new` keyword to create an array. Let's create an array with 15 bool elements:

```
myArray = new bool[15];
```

Use square brackets to set one of the values in the array. This statement sets the value of the fifth element of `myArray` to `true` by using square brackets and specifying the **index** 4. It's the fifth one because the first is `myArray[0]`, the second is `myArray[1]`, etc.:

```
myArray[4] = false;
```

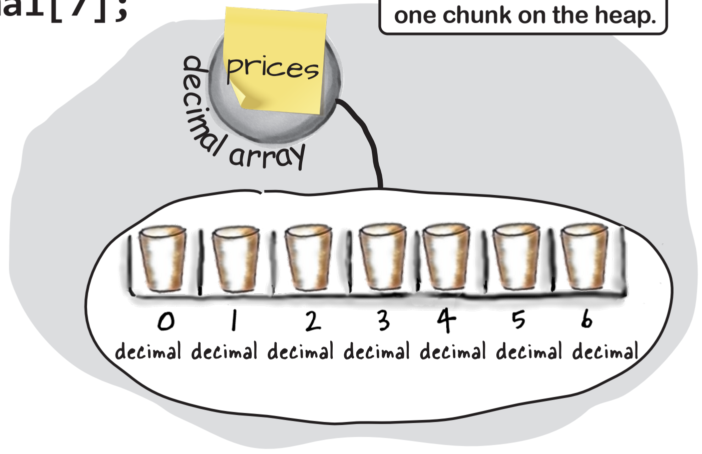
You use the new keyword to create an array because it's an object—so an array variable is a kind of reference variable. In C#, arrays are zero-based, which means the first element has index 0.

Use each element in an array like it's a normal variable

When you use an array, first you need to **declare a reference variable** that points to the array. Then you need to **create the array object** using the `new` statement, specifying how big you want the array to be. Then you can **set the elements** in the array. Here's an example of code that declares and fills up an array—and what's happening in the heap when you do it. The first element in the array has an **index** of 0.

The `prices` variable is a reference, just like any other object reference. The object it points to is an array of decimal values, all in one chunk on the heap.

```
// declare a new 7-element decimal array
decimal[] prices = new decimal[7];
prices[0] = 12.37M;
prices[1] = 6_193.70M;

// we didn't set the element
// at index 2, it remains
// the default value of 0

prices[3] = 1193.60M;
prices[4] = 58_000_000_000M;
prices[5] = 72.19M;
prices[6] = 74.8M;
```



Arrays can contain reference variables

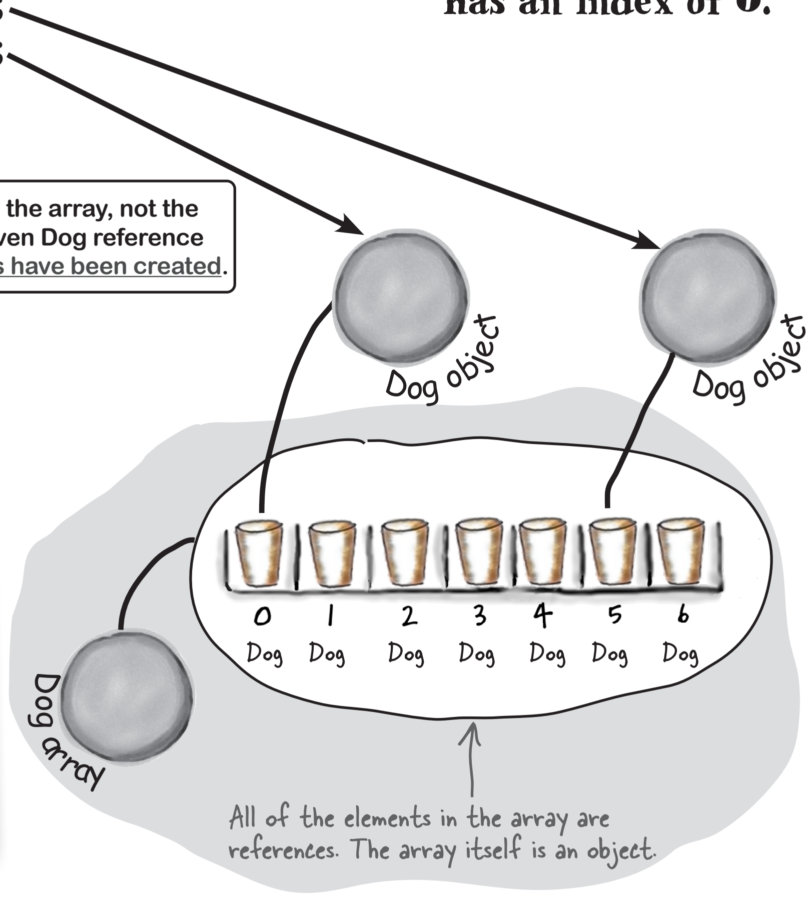
You can create an **array of object references** just like you create an array of numbers or strings. Arrays don't care what type of variable they store; it's up to you. So you can have an array of ints, or an array of Duck objects, with no problem.

Here's code that creates an array of seven Dog variables. The line that initializes the array only creates reference variables. Since there are only two `new Dog()` lines, only two actual instances of the Dog class are created.

```
// Declare a variable that holds an
// array of references to Dog objects
Dog[] dogs = new Dog[7];
// Create two new instances of Dog
// and put them at indexes 0 and 5
dogs[5] = new Dog();
dogs[0] = new Dog();
```

When you set or retrieve an element from an array, the number inside the brackets is called the index. The first element in the array has an index of 0.

The first line of code only created the array, not the instances. The array is a list of seven Dog reference variables—but only two Dog objects have been created.



All of the elements in the array are references. The array itself is an object.

An array's length

You can find out how many elements are in an array using its Length property. So if you've got an array called "prices", then you can use prices.Length to find out how long it is. If there are seven elements in the array, that'll give you 7—which means the array elements are numbered 0 to 6.

Sharpen your pencil

Here's an array of Elephant objects and a loop that will go through it and find the one with the biggest ears. What's the value of biggestEars.EarSize **after** each iteration of the `for` loop?

```
private static void Main(string[] args)
{
    Elephant[] elephants = new Elephant[7];
    elephants[0] = new Elephant() { Name = "Lloyd", EarSize = 40 };
    elephants[1] = new Elephant() { Name = "Lucinda", EarSize = 33 };
    elephants[2] = new Elephant() { Name = "Larry", EarSize = 42 };
    elephants[3] = new Elephant() { Name = "Lucille", EarSize = 32 };
    elephants[4] = new Elephant() { Name = "Lars", EarSize = 44 };
    elephants[5] = new Elephant() { Name = "Linda", EarSize = 37 };
    elephants[6] = new Elephant() { Name = "Humphrey", EarSize = 45 };

    Elephant biggestEars = elephants[0];
    for (int i = 1; i < elephants.Length; i++)
    {
        Console.WriteLine("Iteration #" + i);

        if (elephants[i].EarSize > biggestEars.EarSize)
        {
            biggestEars = elephants[i];
        }

        Console.WriteLine(biggestEars.EarSize.ToString());
    }
}
```

We're creating an array of seven Elephant references.

Arrays start with index 0, so the first Elephant in the array is elephants[0].

This sets the biggestEars reference to the object that elephants[i] points to.

Be careful—this loop starts with the second element of the array (at index 1) and iterates six times until "i" is equal to the length of the array.

Iteration #1 biggestEars.EarSize = ________

Iteration #2 biggestEars.EarSize = ________

Iteration #3 biggestEars.EarSize = ________

Iteration #4 biggestEars.EarSize = ________

Iteration #5 biggestEars.EarSize = ________

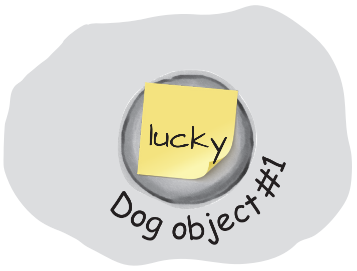
Iteration #6 biggestEars.EarSize = ________

null means a reference points to nothing

There's another important keyword that you'll use with objects. When you create a new reference and don't set it to anything, it has a value. It starts off set to `null`, which means **it's not pointing to any object at all**. Let's have a closer look at this:

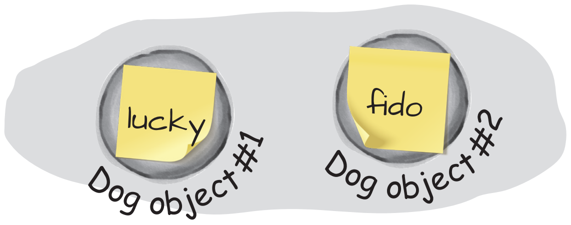
The default value for any reference variable is `null`. Since we haven't assigned a value to `fido`, it's set to `null`.

```
Dog fido;
Dog lucky = new Dog();
```



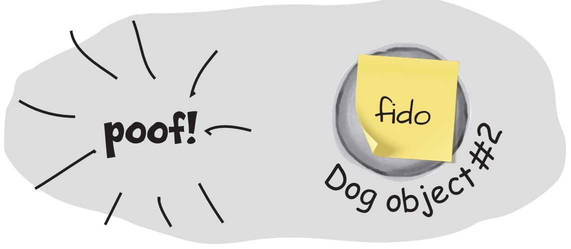
Now `fido` is set to a reference to another object, so it's not equal to `null` anymore.

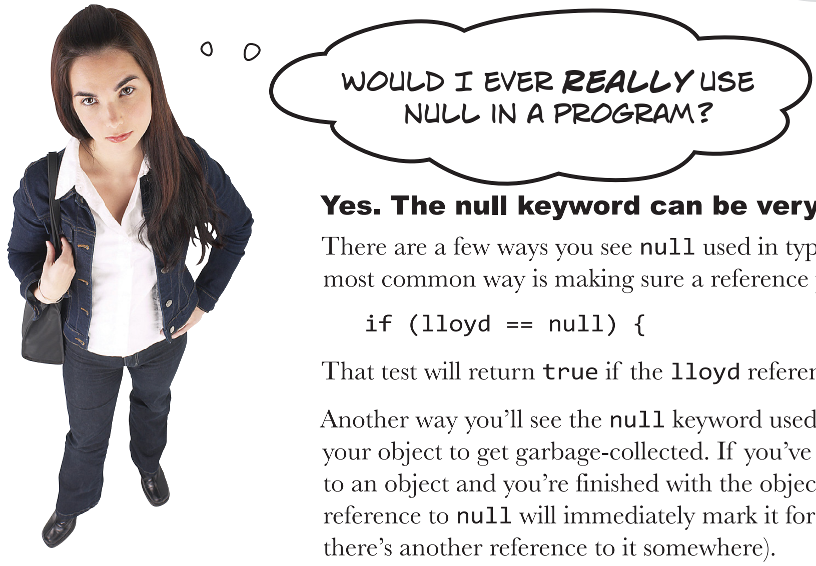
```
fido = new Dog();
```



Once we set `lucky` to `null` it no longer points to its object, so it gets marked for garbage collection.

```
lucky = null;
```





Yes. The null keyword can be very useful.

There are a few ways you see `null` used in typical programs. The most common way is making sure a reference points to an object:

```
if (lloyd == null) {
```

That test will return `true` if the `lloyd` reference is set to `null`.

Another way you'll see the `null` keyword used is when you ***want*** your object to get garbage-collected. If you've got a reference to an object and you're finished with the object, setting the reference to `null` will immediately mark it for collection (unless there's another reference to it somewhere).

Sharpen your pencil Solution

Here's an array of Elephant objects and a loop that will go through it and find the one with the biggest ears. What's the value of biggestEars.EarSize **after** each iteration of the for loop?

The for loop starts with the second Elephant and compares it to whatever Elephant biggestEars points to. If its ears are bigger, it points biggestEars at that Elephant instead. Then it moves to the next one, then the next one...by the end of the loop, biggestEars points to the one with the biggest ears.

```
private static void Main(string[] args)
{
    Elephant[] elephants = new Elephant[7];
    elephants[0] = new Elephant() { Name = "Lloyd", EarSize = 40 };
    elephants[1] = new Elephant() { Name = "Lucinda", EarSize = 33 };
    elephants[2] = new Elephant() { Name = "Larry", EarSize = 42 };
    elephants[3] = new Elephant() { Name = "Lucille", EarSize = 32 };
    elephants[4] = new Elephant() { Name = "Lars", EarSize = 44 };
    elephants[5] = new Elephant() { Name = "Linda", EarSize = 37 };
    elephants[6] = new Elephant() { Name = "Humphrey", EarSize = 45 };

    Elephant biggestEars = elephants[0];
    for (int i = 1; i < elephants.Length; i++)
    {
        Console.WriteLine("Iteration #" + i);

        if (elephants[i].EarSize > biggestEars.EarSize)
        {
            biggestEars = elephants[i];
        }

        Console.WriteLine(biggestEars.EarSize.ToString());
    }
}
```

Did you remember that the loop starts with the second element of the array? Why do you think that is?

The biggestEars reference keeps track of which Elephant we've seen so far has the biggest ears. Use the debugger to check this! Put your breakpoint here and watch biggestEars.EarSize.

Iteration #1 biggestEars.EarSize = 40

Iteration #2 biggestEars.EarSize = 42

Iteration #3 biggestEars.EarSize = 42

Iteration #4 biggestEars.EarSize = 44

Iteration #5 biggestEars.EarSize = 44

Iteration #6 biggestEars.EarSize = 45

there are no Dumb Questions

Q: **I'm still not sure I get how references work.**

A: References are the way you use all of the methods and fields in an object. If you create a reference to a Dog object, you can then use that reference to access any methods you've created for the Dog object. If the Dog class has (nonstatic) methods called Bark and Fetch, you can create a reference called `spot`, and then you can use that to call spot.Bark() or spot.Fetch(). You can also change information in the fields for the object using the reference (so you could change a Breed field using spot.Breed).

Q: **Then doesn't that mean that every time I change a value through a reference I'm changing it for all of the other references to that object, too?**

A: Yes. If the `rover` variable contains a reference to the same object as `spot`, changing rover.Breed to "beagle" would make it so that spot.Breed was "beagle".

Q: **Remind me again—what does `this` do?**

A: `this` is a special variable that you can only use inside an object. When you're inside a class, you use `this` to refer to any field or method of that particular instance. It's especially useful when you're working with a class whose methods call other classes. One object can use it to send **a reference to itself** to another object. So if `spot` calls one of `rover`'s methods passing `this` as a parameter, he's giving `rover` a reference to the `spot` object.

Any time you've got code in an object that's going to be instantiated, the instance can use the special this variable that has a reference to itself.

Q: **You keep talking about garbage-collecting, but what's actually doing the collecting?**

A: Every .NET app runs inside the **Common Language Runtime** (or the Mono Runtime if you're running your apps on macOS, Linux, or using Mono on Windows). The CLR does a lot of stuff, but there are two *really important things* the CLR does that we're concerned about right now. First, it **executes your code**—specifically, the output produced by the C# compiler. Second, it manages the memory that your program uses. That means it keeps track of all of your objects, figures out when the last reference to an object disappears, and frees up the memory that it was using. The .NET team at Microsoft and the Mono team at Xamarin (which was a separate company for many years, but is now a part of Microsoft) have done an enormous amount of work making sure that it's fast and efficient.

Q: **I still don't get that stuff about different types holding different-sized values. Can you go over that one more time?**

A: Sure. The thing about variables is they assign a size to your number no matter how big its value is. So if you name a variable and give it a long type even though the number is really small (like, say, 5), the CLR sets aside enough memory for it to get really big. When you think about it, that's really useful. After all, they're called variables because they change all the time.

The CLR assumes you know what you're doing and you're not going to give a variable a type bigger than it needs. So even though the number might not be big now, there's a chance that after some math happens, it'll change. The CLR gives it enough memory to handle the largest value that type can accommodate.

Game design... and beyond

Tabletop Games

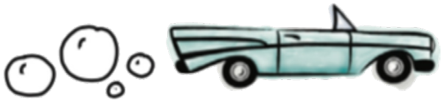
There's a rich history to tabletop games—and, as it turns out, a long history of tabletop games influencing video games, at least as early as the very first commercial role-playing game.

- The first edition of Dungeons and Dragons (D&D) was released in 1974, and that same year games with names like "dungeon" and "dnd" started popping up on university mainframe computers.
- You've used the Random class to create numbers. The idea of games based on random numbers has a long history—for example, tabletop games that use dice, cards, spinners, and other sources of randomness.
- We saw in the last chapter how a paper prototype can be a valuable first step in designing a video game. Paper prototypes have a strong resemblance to tabletop games. In fact, you can often turn the paper prototype of a video game into a playable tabletop game, and use it to test some game mechanics.
- You can use tabletop games—especially card games and board games—as learning tools to understand the more general concept of game mechanics. Dealing, shuffling, dice rolling, rules for moving pieces around the board, use of a sand timer, and rules for cooperative play are all examples of mechanics.
- The mechanics of Go Fish include dealing cards, asking another player for a card, saying "Go Fish" when asked for a card you don't have, determining the winner, etc. Take a minute and read the rules here: https://en.wikipedia.org/wiki/Go_Fish#The_game.

If you've never played Go Fish, take a few minutes and read the rules. We'll use them later in the book!

Even if we're not writing code for video games, there's a lot we can learn from tabletop games.

A lot of our programs depend on **random numbers**. For example, you've already used the Random class to create random numbers for several of your apps. Most of us don't actually have a lot of real-world experience with genuine random numbers… except when we play games. Rolling dice, shuffling cards, spinning spinners, flipping coins…these are all great examples of **random number generators**. The Random class is .NET's random number generator—you'll use it in many of your programs, and your experience using random numbers when playing tabletop games will make it a lot easier for you to understand what it does.



A random test drive

You'll be using the .NET Random class throughout the book, so let's get to know it better by kicking its tires and taking it for a spin. Fire up Visual Studio and follow along—and make sure you run your code multiple times, since you'll get different random numbers each time.

1 **Create a new console app**—all of this code will go in the Main method. Start by creating a new instance of Random, generating a random int, and writing it to the console:

```
Random random = new Random();
int randomInt = random.Next();
Console.WriteLine(randomInt);
```

Specify a **maximum value** to get random numbers from 0 up to—but not including—the maximum value. A maximum of 10 generates random numbers from 0 to 9:

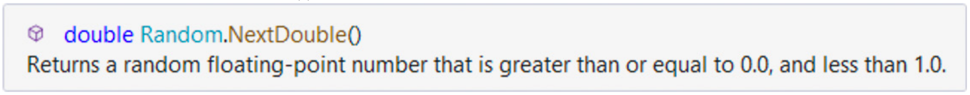
```
int zeroToNine = random.Next(10);
Console.WriteLine(zeroToNine);
```

2 Now **simulate the roll of a die**. You can specify a minimum and maximum value. A minimu of 1 and maximum of 7 generates random numbers from 1 to 6:

```
int dieRoll = random.Next(1, 7);
Console.WriteLine(dieRoll);
```

3 The **NextDouble method** generates random double values. Hover over the method name to see a tooltip—it generates a floating-point number from 0.0 up to 1.0:

```
double randomDouble = random.NextDouble();
```



double Random.NextDouble()
Returns a random floating-point number that is greater than or equal to 0.0, and less than 1.0.

You can use **multiply a random double** to generate much larger random numbers. So if you want a random double value from 1 to 100, multiply the random double by 100:

```
Console.WriteLine(randomDouble * 100);
```

Use **casting** to convert the random double to other types. Try running this code a bunch of times—you'll see tiny precision differences in the float and decimal values.

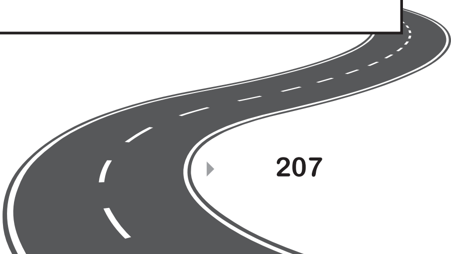
```
Console.WriteLine((float)randomDouble * 100F);
Console.WriteLine((decimal)randomDouble * 100M);
```

4 Use a maximum value of 2 to **simulate a coin toss**. That generates a random value of either 0 or 1. Use the special **Convert class**, which has a static ToBoolean method that will convert it to a Boolean value:

```
int zeroOrOne = random.Next(2);
bool coinFlip = Convert.ToBoolean(zeroOrOne);
Console.WriteLine(coinFlip);
```

BRAIN POWER

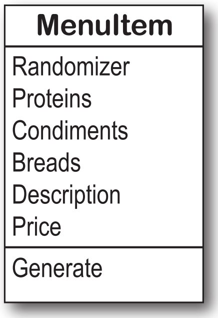
How would you use Random to choose a random string from an array of strings?



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

Sloppy Joe has a pile of meat, a whole lotta bread, and more condiments than you can shake a stick at. What he doesn't have is a menu! Can you build a program that makes a new *random* menu for him every day? You definitely can… with a **new WPF app**, some arrays, and a couple of useful new techniques.

Do this!



1. **Add a new MenuItem class to your project and add its fields.**
Have a look at the class diagram. It has four fields: an instance of Random and three arrays to hold the various sandwich parts. The array fields use **collection initializers**, which let you define the items in an array by putting them inside curly braces.

```
class MenuItem
{
    public Random Randomizer = new Random();

    public string[] Proteins = { "Roast beef", "Salami", "Turkey",
                "Ham", "Pastrami", "Tofu" };
    public string[] Condiments = { "yellow mustard", "brown mustard",
                "honey mustard", "mayo", "relish", "french dressing" };
    public string[] Breads = { "rye", "white", "wheat", "pumpernickel", "a roll" };

            public string Description = "";
            public string Price;
}
```

2. **Add the GenerateMenuItem method to the MenuItem class.**
This method uses the same Random.Next method you've seen many times to pick random items from the arrays in the Proteins, Condiments, and Breads fields and concatenate them together into a string.

```
public void Generate()
{
    string randomProtein = Proteins[Randomizer.Next(Proteins.Length)];
    string randomCondiment = Condiments[Randomizer.Next(Condiments.Length)];
    string randomBread = Breads[Randomizer.Next(Breads.Length)];
    Description = randomProtein + " with " + randomCondiment + " on " + randomBread;

    decimal bucks = Randomizer.Next(2, 5);
    decimal cents = Randomizer.Next(1, 98);
    decimal price = bucks + (cents * .01M);
    Price = price.ToString("c");
}
```

This method makes a random price between 2.01 and 5.97 by converting two random ints to decimals. Have a close look at the last line—it returns `price.ToString("c")`. The parameter to the ToString method is a format. In this case, the `"c"` format tells ToString to format the value with the local currency: if you're in the United States you'll see a $; in the UK you'll get a £, in the EU you'll see €, etc.

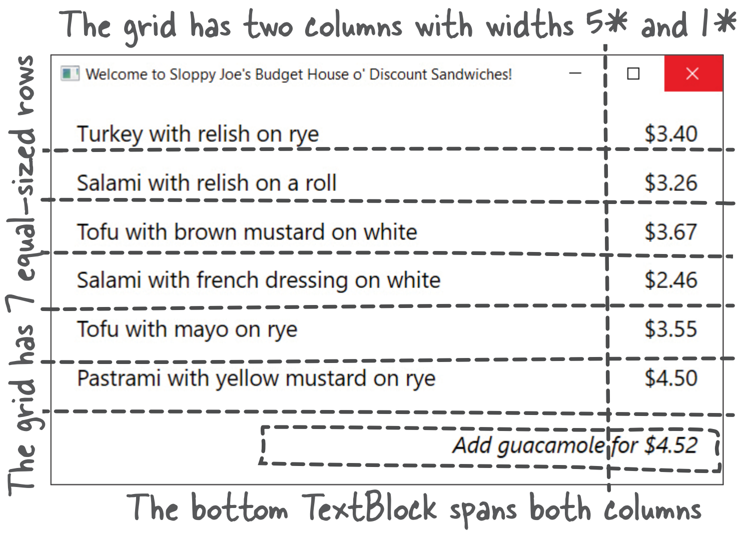
Go to the Visual Studio for Mac Learner's Guide for the Mac version of this project.

3 **Create the XAML to lay out the window.**

Your app will display random menu items in a window with two columns, a wide one for the menu item and a narrow one for the price. Each cell in the grid has a TextBlock control with its FontSize set to `18px`—except for the bottom row, which just has a single right-aligned TextBlock that spans both columns. The window's title is "Welcome to Sloppy Joe's Budget House o' Discount Sandwiches!" and it's got a height of `350` and width of `550`. The grid has a margin of `20`.

We're building on the XAML you learned in the last two WPF projects. You can lay it out in the designer, type it in by hand, or do some of each.

The grid has a margin of 20 to give the whole menu a little extra space.


The grid has two columns with widths 5* and 1*
The grid has 7 equal-sized rows
The bottom TextBlock spans both columns

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

Turkey with relish on rye $3.40
Salami with relish on a roll $3.26
Tofu with brown mustard on white $3.67
Salami with french dressing on white $2.46
Tofu with mayo on rye $3.55
Pastrami with yellow mustard on rye $4.50
Add guacamole for $4.52

Name each of the TextBlocks in the left column item1, item2, item3, etc., and the TextBlocks in the right column price1, price2, price3, etc. Name the bottom TextBlock *`guacamole`*.

```
<Grid Margin="20">
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="5*"/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>

    <TextBlock x:Name="item1" FontSize="18px" />
    <TextBlock x:Name="price1" FontSize="18px" HorizontalAlignment="Right" Grid.Column="1"/>
    <TextBlock x:Name="item2" FontSize="18px" Grid.Row="1"/>
    <TextBlock x:Name="price2" FontSize="18px" HorizontalAlignment="Right"
               Grid.Row="1" Grid.Column="1"/>
    <TextBlock x:Name="item3" FontSize="18px" Grid.Row="2" />
    <TextBlock x:Name="price3" FontSize="18px" HorizontalAlignment="Right" Grid.Row="2"
               Grid.Column="1"/>
    <TextBlock x:Name="item4" FontSize="18px" Grid.Row="3" />
    <TextBlock x:Name="price4" FontSize="18px" HorizontalAlignment="Right" Grid.Row="3"
               Grid.Column="1"/>
    <TextBlock x:Name="item5" FontSize="18px" Grid.Row="4" />
    <TextBlock x:Name="price5" FontSize="18px" HorizontalAlignment="Right" Grid.Row="4"
               Grid.Column="1"/>
    <TextBlock x:Name="item6" FontSize="18px" Grid.Row="5" />
    <TextBlock x:Name="price6" FontSize="18px" HorizontalAlignment="Right" Grid.Row="5"
               Grid.Column="1"/>
    <TextBlock x:Name="guacamole" FontSize="18px" FontStyle="Italic" Grid.Row="6"
               Grid.ColumnSpan="2" HorizontalAlignment="Right" VerticalAlignment="Bottom"/>
</Grid>
```

4 **Add the code-behind for your XAML window.**

The menu is generated by a method called MakeTheMenu, which your window calls right after it calls InitializeComponent. It uses an array of MenuItem classes to generate each item in the menu. We want the first three items to be normal menu items. The next two are only served on bagels. The last is a special item with its own set of ingredients.

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

private void MakeTheMenu()
{
    MenuItem[] menuItems = new MenuItem[5];
    string guacamolePrice;

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

    item1.Text = menuItems[0].Description;
    price1.Text = menuItems[0].Price;
    item2.Text = menuItems[1].Description;
    price2.Text = menuItems[1].Price;
    item3.Text = menuItems[2].Description;
    price3.Text = menuItems[2].Price;
    item4.Text = menuItems[3].Description;
    price4.Text = menuItems[3].Price;
    item5.Text = menuItems[4].Description;
    price5.Text = menuItems[4].Price;

    MenuItem specialMenuItem = new MenuItem()
    {
        Proteins = new string[] { "Organic ham", "Mushroom patty", "Mortadella" },
        Breads = new string[] { "a gluten free roll", "a wrap", "pita" },
        Condiments = new string[] { "dijon mustard", "miso dressing", "au jus" }
    };
    specialMenuItem.Generate();

    item6.Text = specialMenuItem.Description;
    price6.Text = specialMenuItem.Price;

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

    guacamole.Text = "Add guacamole for " + guacamoleMenuItem.Price;
}
```

Let's take a closer look at what's going on here. Menu items #4 and #5 (at indexes 3 and 4) get a MenuItem object that's initialized with an object initializer, just like you used with Joe and Bob. This object initializer sets the Breads field to a new string array. That string array uses a collection initializer with four strings that describe different types of bagels. Did you notice that this collection initializer includes the array type (`new string[]`)? You didn't include that when you defined your fields. You can add `new string[]` to the collection initializers in the MenuItem fields if you want—but you don't have to. They're optional because the fields had the type definitions in their declarations.

This uses "new string[]" to declare the type of the array being initialized. The MenuItem fields didn't need to include that because they already have a type.

Make sure you call the Generate method, otherwise the MenuItem's fields will be empty and your page will be mostly blank.

The last item on the menu is for the daily special sandwich made from premium ingredients, so it gets its own MenuItem object with all three of its string array fields initialized with object initializers.

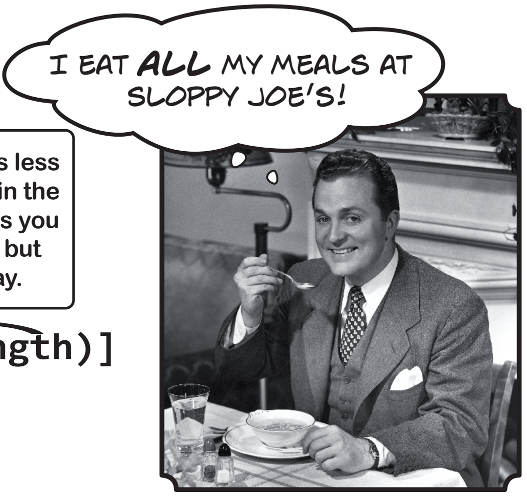
There's a separate menu item just to create a new price for the guacamole.

How it works...

The Randomizer.Next(7) method gets a random int that's less than 7. Breads.Length returns the number of elements in the Breads array. So Randomizer.Next(Breads.Length) gives you a random number that's greater than or equal to zero, but less than the number of elements in the Breads array.

```
Breads[Randomizer.Next(Breads.Length)]
```

Breads is an array of strings. It's got five elements, numbered from 0 to 4. So Breads[0] equals "rye", and Breads[3] equals "a roll".



If your computer is fast enough, your program may not run into this problem. If you run it on a much slower computer, you'll see it.

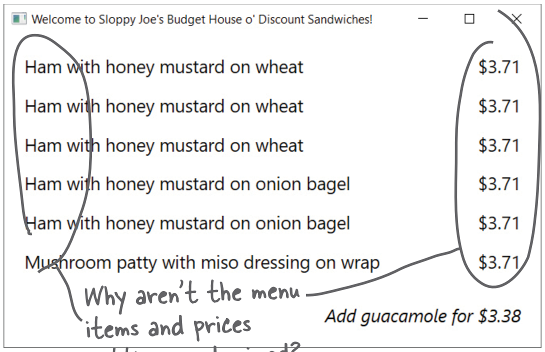
5 **Run your program and behold the new randomly generated menu.**
Uh…something's wrong. The prices on the menu are all the same, and the menu items are weird—the first three are the same, so are the next two, and they all seem to have the same protein. What's going on?

It turns out that the .NET Random class is actually a **pseudo-random number** generator, which means that it uses a mathematical formula to generate a sequence of numbers that can pass certain statistical tests for randomness. That makes them good enough to use in any app we'll build (but don't use it as part of a security system that depends on truly random numbers!). That's why the method is called Next—you're getting the next number in the sequence. The formula starts with a "seed value"—it uses that value to find the next one in the sequence. When you create a new instance of Random, it uses the system clock to "seed" the formula, but you can provide your own seed. Try using the C# Interactive window to call **`new Random(12345).Next();`** a bunch of times. You're telling it to create a new instance of Random with the same seed value (12345), so the Next method will give you the same "random" number each time.

When you see a bunch of different instances of Random give you the same value, it's because they were all seeded close enough that the system clock didn't change time, so they all have the same seed value. So how do we fix this? Use a single instance of Random by making the Randomizer field static so all MenuItems share a single Random instance:

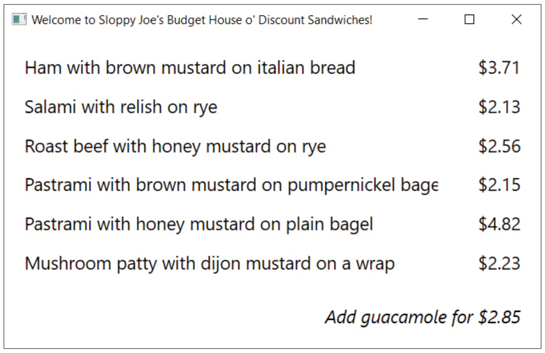
```
public static Random Randomizer = new Random();
```

Run your program again—now the menu will be randomized.


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

```
Ham with honey mustard on wheat                        $3.71
Ham with honey mustard on wheat                        $3.71
Ham with honey mustard on wheat                        $3.71
Ham with honey mustard on onion bagel                  $3.71
Ham with honey mustard on onion bagel                  $3.71
Mushroom patty with miso dressing on wrap              $3.71

                                  Add guacamole for $3.38
```

Why aren't the menu items and prices getting randomized?


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

```
Ham with brown mustard on italian bread                $3.71
Salami with relish on rye                              $2.13
Roast beef with honey mustard on rye                   $2.56
Pastrami with brown mustard on pumpernickel bage       $2.15
Pastrami with honey mustard on plain bagel             $4.82
Mushroom patty with dijon mustard on a wrap            $2.23

                                  Add guacamole for $2.85
```

BULLET POINTS

- The `new` keyword **returns a reference to an object** that you can store in a reference variable.
- You can have **multiple references to the same object**. You can change an object with one reference and access the results of that change with another.
- For an object to stay in the heap, it has to be **referenced**. Once the last reference to an object disappears, it eventually gets garbage-collected and the memory it used is reclaimed.
- Your .NET programs run in the **Common Language Runtime**, a "layer" between the OS and your program. The C# compiler builds your code into **Common Intermediate Language (CIL)**, which the CLR executes.
- The **`this` keyword** lets an object get a reference to itself.
- **Arrays** are objects that hold multiple values. They can contain either values or references.
- **Declare array variables** by putting square brackets after the type in the variable declaration (like **`bool[] trueFalseValues`** or **`Dog[] kennel`**).
- Use the `new` keyword to **create a new array**, specifying the array length in square brackets (like **`new bool[15]`** or **`new Dog[3]`**).
- Use the Length **method** on an array to get its length (like kennel.Length).
- Access an array value using its **index** in square brackets (like `bool[3]` or `Dog[0]`). Array indexes start at 0.
- **`null`** means a reference points to nothing. The `null` keyword is useful for testing if a reference is null, or clearing a reference variable so an object will get marked for garbage collection.
- Use **collection initializers** to initialize an array by setting the array equal to the `new` keyword followed by the array type followed by a comma-delimited list in curly braces (like `new int[] { 8, 6, 7, 5, 3, 0, 9 }`). The array type is optional when setting a variable or field value in the same statement where it's declared.
- You can pass a **format parameter** to an object or value's ToString method. If you're calling a numeric type's ToString method, passing it a value of `"c"` formats the value as a local currency.
- The .NET Random class is a pseudo-random number generator seeded by the system clock. Use a single instance of Random to avoid multiple instances with the same seed generating the same sequence of numbers.

Unity Lab #2

Write C# Code for Unity

Unity isn't *just* a powerful, cross-platform engine and editor for building 2D and 3D games and simulations. It's also a **great way to get practice writing C# code**.

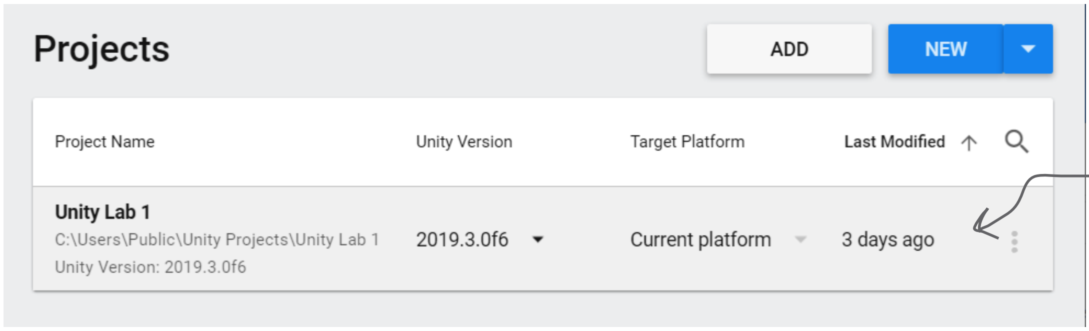
In the last Unity Lab, you learned how to navigate around Unity and your 3D space, and started to create and explore GameObjects. Now it's time to write some code to take control of your GameObjects. The whole goal of that lab was to get you oriented in the Unity editor (and give you an easy way to remind yourself of how to navigate around it if you need it).

In this Unity Lab, you'll start writing code to control your GameObjects. You'll write C# code to explore concepts you'll use in the rest of the Unity Labs, starting with adding a method that rotates the 8 Ball GameObject that you created in the last Unity Lab. You'll also start using the Visual Studio debugger with Unity to sleuth out problems in your games.

C# scripts add behavior to your GameObjects

Now that you can add a GameObject to your scene, you need a way to make it, well, do stuff. That's where your C# skills come in. Unity uses **C# scripts** to define the behavior of everything in the game.

This Unity Lab will introduce tools that you'll use to work with C# and Unity. You're going to build a simple "game" that's really just a little bit of visual eye candy: you'll make your 8 ball fly around the scene. Start by going to Unity Hub and **opening the same project** that you created in the first Unity Lab.



Project Name	Unity Version	Target Platform	Last Modified
Unity Lab 1 C:\Users\Public\Unity Projects\Unity Lab 1 Unity Version: 2019.3.0f6	2019.3.0f6	Current platform	3 days ago

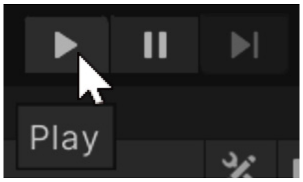
This Unity Lab picks up where the first one left off, so go to Unity Hub and open the project you created in the last lab.

Here's what you'll do in this Unity Lab:

1. **Attach a C# script to your GameObject.** You'll add a Script component to your Sphere GameObject. When you add it, Unity will create a class for you. You'll modify that class so that it drives the 8 ball sphere's behavior.

2. **Use Visual Studio to edit the script.** Remember how you set the Unity editor's preferences to make Visual Studio the script editor? That means you can just double-click on the script in the Unity editor and it will open up in Visual Studio.

3. **Play your game in Unity.** There's a Play button at the top of the screen. When you press it, it starts executing all of the scripts attached to the GameObjects in your scene. You'll use that button to run the script that you added to the sphere.



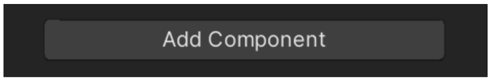
The Play button does not save your game! So make sure you save early and save often. A lot of people get in the habit of saving the scene every time they run the game.

4. **Use Unity and Visual Studio together to debug your script.** You've already seen how valuable the Visual Studio debugger is when you're trying to track down problems in your C# code. Unity and Visual Studio work together seamlessly so you can add breakpoints, use the Locals window, and work with the other familiar tools in the Visual Studio debugger while your game is running.

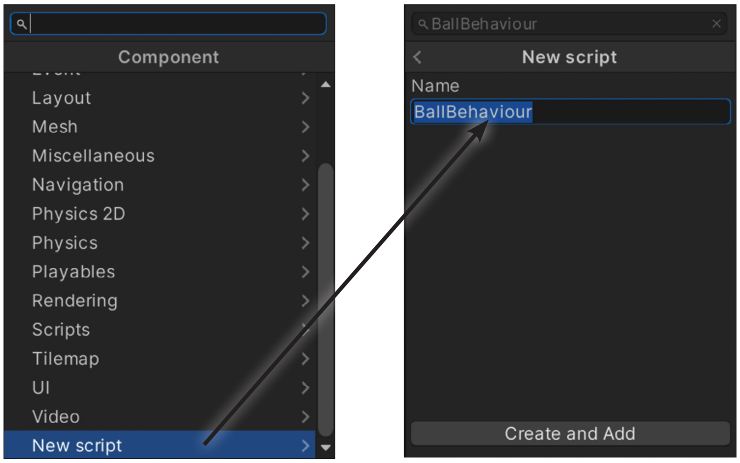
Add a C# script to your GameObject

Unity is more than an amazing platform for building 2D and 3D games. Many people use it for artistic work, data visualization, augmented reality, and more. It's especially valuable to you, as a C# learner, because you can write code to control everything that you see in a Unity game. That makes Unity ***a great tool for learning and exploring C#***.

Let's start using C# and Unity right now. Make sure the Sphere GameObject is selected, then **click the Add Component button** at the bottom of the Inspector window.



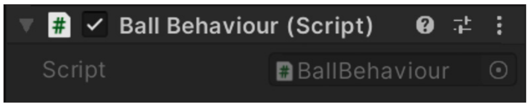
When you click it, Unity pops up a window with all of the different kinds of components that you can add—and there are ***a lot*** of them. **Choose "New script"** to add a new C# script to your Sphere GameObject. You'll be prompted for a name. **Name your script `BallBehaviour`**.



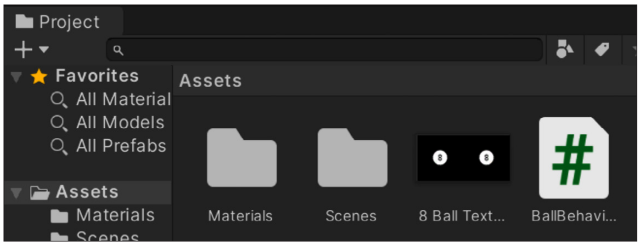
Unity code uses British spelling.

*If you're American (like us), or if you're used to the US spelling of the word **behavior**, you'll need to be careful when you work with Unity scripts because the class names often feature the British spelling **behaviour**.*

Click the "Create and Add" button to add the script. You'll see a component called *Ball Behaviour (Script)* appear in the Inspector window.



You'll also see the C# script in the Project window.



The Project window gives you a folder-based view of your project. Your Unity project is made up of files: media files, data files, C# scripts, textures, and more. Unity calls these files assets. The Project window was displaying a folder called Assets when you right-clicked inside it to import your texture, so Unity added it to that folder.

Did you notice a folder called Materials appeared in the Project window as soon as you dragged the 8 ball texture onto your sphere?

Write C# code to rotate your sphere

In the first lab, you told Unity to use Visual Studio as its external script editor. So go ahead and **double-click 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()
    {

    }
}
```

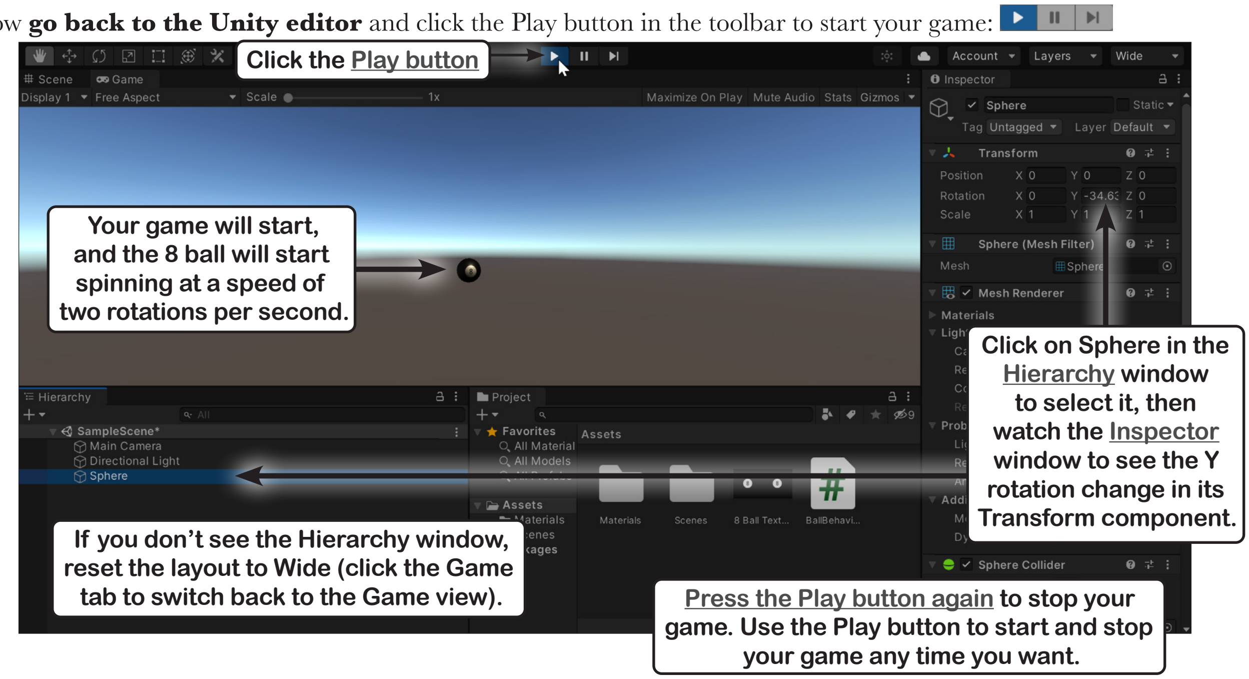
You opened your C# script in Visual Studio by clicking on it in the Hierarchy window, which shows you a list of every GameObject in the current scene. When Unity created your project, it added a scene called SampleScene with a camera and a light. You added a sphere to it, so your Hierarchy window will show all of those things.

If Unity didn't launch Visual Studio and open your C# script in it, go back to the beginning of Unity Lab 1 and make sure you followed the steps to set the External Tools preferences.

Here's a line of code that will rotate your sphere. **Add it to your Update method**:

```
    transform.Rotate(Vector3.up, 180 * Time.deltaTime);
```

Now **go back to the Unity editor** and click the Play button in the toolbar to start your game:


Click the Play button
Your game will start, and the 8 ball will start spinning at a speed of two rotations per second.
Click on Sphere in the Hierarchy window to select it, then watch the Inspector window to see the Y rotation change in its Transform component.
If you don't see the Hierarchy window, reset the layout to Wide (click the Game tab to switch back to the Game view).
Press the Play button again to stop your game. Use the Play button to start and stop your game any time you want.
Scene
Game
Hierarchy
SampleScene*
Main Camera
Directional Light
Sphere
Project
Inspector
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y -34.63 Z 0
Scale X 1 Y 1 Z 1
Sphere (Mesh Filter)
Mesh Renderer
Sphere Collider
Materials
Scenes
8 Ball Text...
BallBehavi...

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);
  }
}
```

You learned about namespaces in Chapter 2. When Unity created the file with the C# script, it added `using` lines at 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 draws the next one very quickly, and your eye interprets changes in these frames as movement. Unity calls the Update method for every GameObject before each frame so it can move, rotate, or make any other changes that it needs to make. A faster computer will run at a higher frame rate—or number of frames per second (FPS)—than a slower one.

The transform.Rotate method causes a GameObject to rotate. The first parameter is the axis to rotate around. In this case, your code used Vector3.up, which tells it to rotate around the Y axis. The second parameter is the number of degrees to rotate.

Different computers will run your game at different frame rates. If it's running at 30 FPS, we want one rotation every 60 frames. If it's running at 120 FPS, it should rotate once every 240 frames. Your game's frame rate may even change if it needs to run more or less complex code.

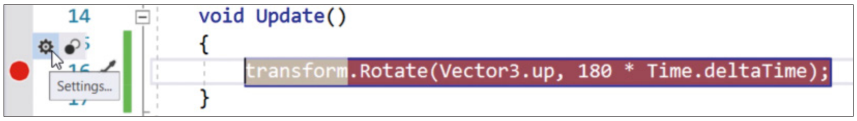
That's where the Time.deltaTime value comes in handy. Every time the Unity engine calls a GameObject's Update method—once per frame—it sets Time.deltaTime to the fraction of a second since the last frame. Since we want our ball to do a full rotation every two seconds, or 180 degrees per second, all we need to do is multiply it by Time.deltaTime to make sure that it rotates exactly as much as it needs to for that frame.

Inside your Update method, multiplying any value by Time.deltaTime turns it into that value per second.

Time.deltaTime is static—and like we saw in Chapter 3, you don't need an instance of the Time class to use it.

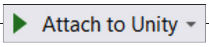
Add a breakpoint and debug your game

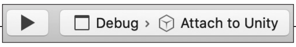
Let's debug your Unity game. First **stop your game** if it's still running (by pressing the Play button again). Then switch over to Visual Studio, and **add a breakpoint** on the line that you added to the Update method.



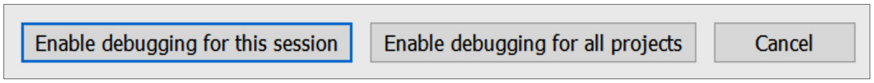
```
void Update()
{
    transform.Rotate(Vector3.up, 180 * Time.deltaTime);
}
```

Now find the button at the top of Visual Studio that starts the debugger:

★ In Windows it looks like this— —or choose Debug >> Start Debugging (F5) from the menu

★ In macOS it looks like this— —or choose Run >> Start Debugging (⌘↩)

Click that button to **start the debugger**. Now switch back to the Unity editor. If this is the first time you're debugging this project, the Unity editor will pop up a dialog window with these buttons:


Enable debugging for this session
Enable debugging for all projects
Cancel

Press the "Enable debugging for this session" button (or if you want to keep that pop-up from appearing again, press "Enable debugging for all projects"). Visual Studio is now ***attached*** to Unity, which means it can debug your game.

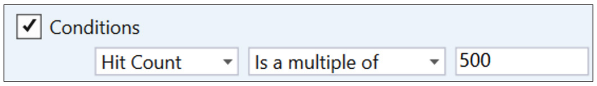
Now **press the Play button in Unity** to start your game. Since Visual Studio is attached to Unity, it ***breaks immediately*** on the breakpoint that you added, just like with any other breakpoint you've set.

Congratulations, you're now debugging a game!

Use a hit count to skip frames

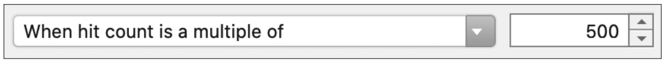
Sometimes it's useful to let your game run for a while before your breakpoint stops it. For example, you might want your game to spawn and move its enemies before your breakpoint hits. Let's tell your breakpoint to break every 500 frames. You can do that by adding a **Hit Count condition** to your breakpoint:

★ On Windows, right-click on the breakpoint dot (●) at the left side of the line, choose **Conditions** from the pop-up menu, select *Hit Count* and *Is a multiple of* from the dropdowns, and enter 500 in the box:



☑ Conditions
Hit Count | Is a multiple of | 500

★ On macOS, right-click on the breakpoint dot (◎), choose **Edit breakpoint…** from the menu, then choose *When hit count is a multiple of* from the dropdown and enter 500 in the box:



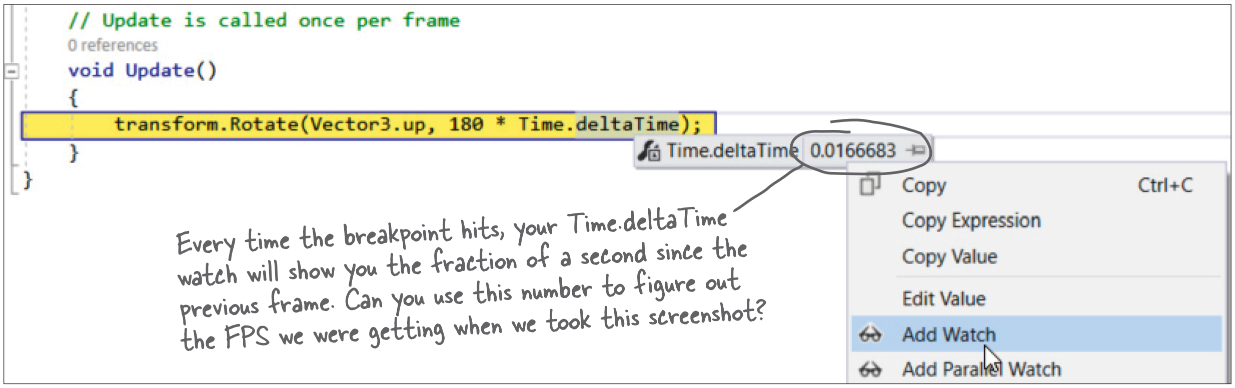
When hit count is a multiple of | 500

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 60 FPS, that means when you press Continue the game will run for a little over 8 seconds before it breaks again. **Press Continue, then switch back to Unity** and watch the ball spin until the breakpoint breaks.

Use the debugger to understand Time.deltaTime

You're going to be using Time.deltaTime in many of the Unity Labs projects. Let's take advantage of your breakpoint and use the debugger to really understand what's going on with this value.

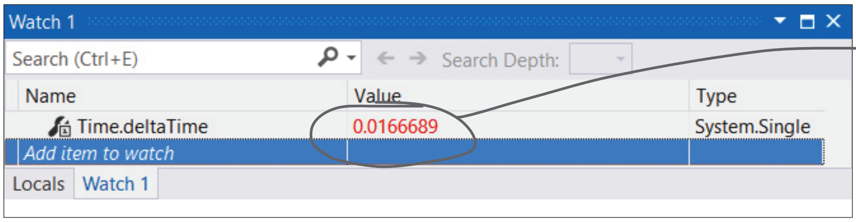
While your game is paused on the breakpoint in Visual Studio, **hover over Time.deltaTime** to see the fraction of a second that elapsed since the previous frame (you'll need to put your mouse cursor over `deltaTime`). Then **add a watch for Time.deltaTime** by selecting Time.deltaTime and choosing Add Watch from the right-mouse menu.



```
    // Update is called once per frame
    0 references
    void Update()
    {
        transform.Rotate(Vector3.up, 180 * Time.deltaTime);
    }
}
```

Every time the breakpoint hits, your Time.deltaTime watch will show you the fraction of a second since the previous frame. Can you use this number to figure out the FPS we were getting when we took this screenshot?

Continue debugging (F5 on Windows, ⇧⌘↩ on macOS), just like with the other apps you've debugged), to resume your game. The ball will start rotating again, and after another 500 frames the breakpoint will trigger again. You can keep running the game for 500 frames at a time. Keep your eye on the Watch window each time it breaks.



Name	Value	Type
Time.deltaTime	0.0166689	System.Single
Add item to watch		

Press the Continue button to get another Time.deltaTime value, then another. You can get your approximate FPS by dividing 1 ÷ Time.deltaTime.

Stop debugging (Shift+F5 on Windows, ⇧⌘↩ on macOS) to stop your program. Then **start debugging again**.Since your game is still running, the breakpoint will <u>continue to work</u> when you reattach Visual Studio to Unity. Once you're done debugging, **toggle your breakpoint again** so the IDE will still keep track of it but not break when it's hit. **Stop debugging** one more time to detach from Unity.

Go back to Unity and **stop your game**—and save it, because the Play button <u>doesn't</u> automatically save the game.

The Play button in Unity starts and stops your game. Visual Studio will stay attached to Unity even when the game is stopped.

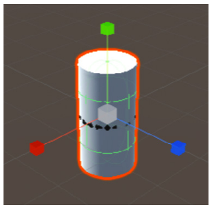
BRAIN POWER

Debug your game again and hover over "Vector3.up" to inspect its value—you'll have to put your mouse cursor over up. It has a value of (0.0, 1.0, 0.0). What do you think that means?

Add a cylinder to show where the Y axis is

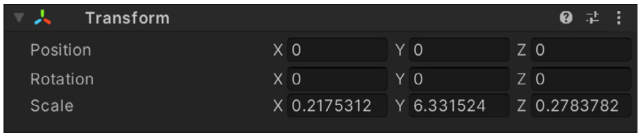
Your sphere is rotating around the Y axis at the very center of the scene. Let's add a very tall and very skinny cylinder to make it visible. ***Create a new cylinder*** by choosing *3D Object >> Cylinder* from the GameObject menu. Make sure it's selected in the Hierarchy window, then look at the Inspector window and check that Unity created it at position (0, 0, 0)—if not, use the context menu (⋮) to reset it.

Let's make the cylinder tall and skinny. Choose the Scale tool from the toolbar: either click on it (⤢) or press the R key. You should see the Scale Gizmo appear on your cylinder:



The Scale Gizmo looks a lot like the Move Gizmo, except that it has cubes instead of cones at the end of each axis. Your new cylinder is sitting on top of the sphere—you might see just a little of the sphere showing through the middle of the cylinder. When you make the cylinder narrower by changing its scale along the X and Z axes, the sphere will get uncovered.

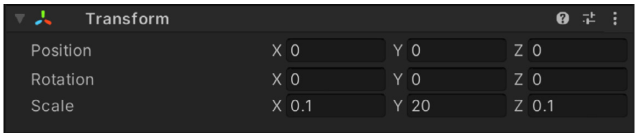
Click and drag the green cube up to elongate your cylinder along the Y axis. Then click on the red cube and drag it toward the cylinder to make it very narrow along the X axis, and do the same with the blue cube to make it very narrow along the Z axis. Watch the Transform panel in the Inspector as you change the cylinder's scale—the Y scale will get larger, and the X and Z values will get much smaller.



Transform			
Position	X 0	Y 0	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 0.2175312	Y 6.331524	Z 0.2783782

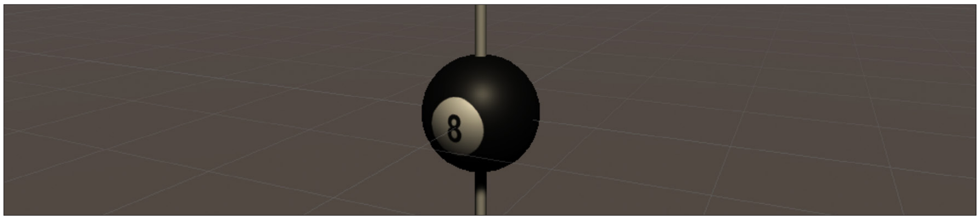
Click on the X label in the Scale row in the Transform panel and drag up and down. Make sure you click the actual X label to the left of the input box with the number. When you click the label it turns blue, and a blue box appears around the X value. As you drag your mouse up and down, the number in the box goes up and down, and the Scene view updates the scale in as you change it. Look closely as you drag—the scale can be positive and negative.

Now **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.



Transform			
Position	X 0	Y 0	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 0.1	Y 20	Z 0.1

Now your sphere has a very long cylinder going through it that shows the Y axis where Y = 0.



Add fields to your class for the rotation angle and speed

In Chapter 3 you learned how C# classes can have **fields** that store values methods can use. Let's modify your code to use fields. Add these four lines just under the class declaration, **immediately after the first curly brace {:**

```
public class BallBehaviour : MonoBehaviour
{
    public float XRotation = 0;
    public float YRotation = 1;
    public float ZRotation = 0;
    public float DegreesPerSecond = 180;
```

These are just like the fields that you added to the projects in Chapters 3 and 4. They're variables that keep track of their values—each time Update is called it reuses the same field over and over again.

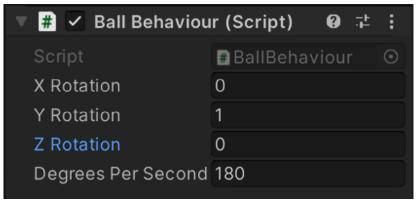
The XRotation, YRotation, and ZRotation fields each contain a value between 0 and 1, which you'll combine to create a **vector** that determines the direction that the ball will rotate:

```
    new Vector3(XRotation, YRotation, ZRotation)
```

The DegreesPerSecond field contains the number of degrees to rotate per second, which you'll multiply by Time.deltaTime just like before. **Modify your Update method to use the fields.** This new code creates a Vector3 variable called `axis` and passes it to the transform.Rotate method:

```
    void Update()
    {
        Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
        transform.Rotate(axis, DegreesPerSecond * Time.deltaTime);
    }
```

Select the Sphere in the Hierarchy window. Your fields now show up in the Script component. When the Script component renders fields, it adds spaces between the capital letters to make them easier to read.

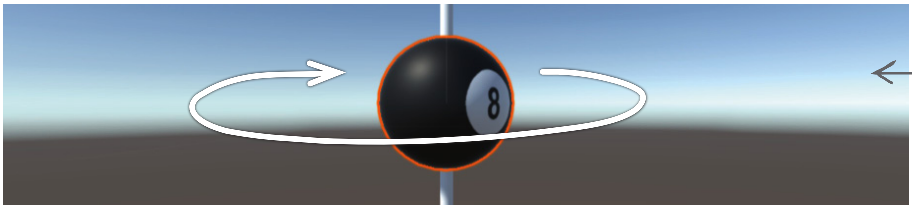

Ball Behaviour (Script)

Script	BallBehaviour
X Rotation	0
Y Rotation	1
Z Rotation	0
Degrees Per Second	180

When you add public fields to a class in your Unity script, the Script component displays input boxes that let you modify those fields. If you modify them while the game is not running, the updated values will get saved with your scene. You can also modify them while the game is running, but they'll revert when you stop the game.

Run your game again. ***While it's running***, select the Sphere in the Hierarchy window and change the degrees per second to 360 or 90—the ball starts to spin at twice or half the speed. Stop your game—and the field will reset to 180.

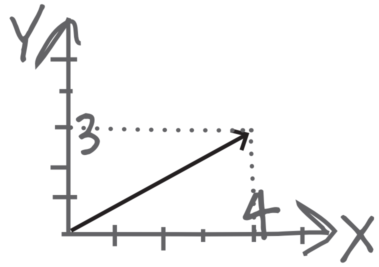
While the game is stopped, use the Unity editor to change the X Rotation to field to 1 and the Y Rotation field to 0. Start your game—the ball will rotate away from you. Click the X Rotation label and drag it up and down to change the value while the game is running. As soon as the number turns negative, the ball starts rotating toward you. Make it positive again and it starts rotating away from you.



When you use the Unity editor to set the Y Rotation field to 1 and then start your game, the ball rotates clockwise around the Y axis.

Use Debug.DrawRay to explore how 3D vectors work

A **vector** is a value with a **length** (or magnitude) and a **direction**. If you ever learned about vectors in a math class, you probably saw lots of diagrams like this one of a 2D vector:



Here's a diagram of a two-dimensional vector. You can represent it with two numbers: its value on the X axis (4) and its value on the Y axis (3), which you'd typically write as (4, 3).

That's not hard to understand…on an intellectual level. But even those of us who took a math class that covered vectors don't always have an ***intuitive*** grasp of how vectors work, especially in 3D. Here's another area where we can use C# and Unity as a tool for learning and exploration.

Use Unity to visualize vectors in 3D

You're going to add code to your game to help you really "get" how 3D vectors work. Start by having a closer look at the first line of your Update method:

```
Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
```

What does this line tell us about the vector?

- **It has a type: Vector3.** Every variable declaration starts with a type. Instead of using string, int, or bool, you're declaring it with the type Vector3. This is a type that Unity uses for 3D vectors.
- **It has a variable name: `axis`.**
- **It uses the `new` keyword to create a Vector3.** It uses the XRotation, YRotation, and ZRotation fields to create a vector with those values.

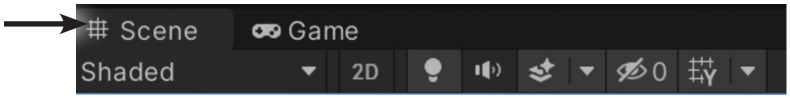
So what does that 3D vector look like? There's no need to guess—we can use one of Unity's useful debugging tools to draw the vector for us. **Add this line of code to the end of your Update method:**

```
void Update()
{
    Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
    transform.Rotate(axis, DegreesPerSecond * Time.deltaTime);
    Debug.DrawRay(Vector3.zero, axis, Color.yellow);
}
```

The Debug.DrawRay method is a special method that Unity provides to help you debug your games. It draws a **ray**—which is a vector that goes from one point to another—and takes parameters for its start point, end point, and color. There's one catch: ***the ray <u>only</u> appears in the Scene view***. The methods in Unity's Debug class are designed so that they don't interfere with your game. They typically only affect how your game interacts with the Unity editor.

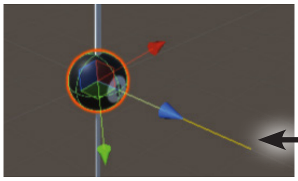
Run the game to see the ray in the Scene view

Now run your game again. You won't see anything different in the Game view because Debug.DrawRay is a tool for debugging that doesn't affect gameplay at all. Use the Scene tab to **switch to the Scene view**. You may also need to **reset the Wide layout** by choosing Wide from the Layout dropdown.



Now you're back in the familiar Scene view. Do these things to get a real sense of how 3D vectors work:

- Use the Inspector to **modify the BallBehaviour script's fields**. Set the X Rotation to 0, Y Rotation to 0, and **Z Rotation to 3**. You should now see a yellow ray coming directly out of the Z axis and the ball rotating around it (remember, the ray only shows up in the Scene view).



The vector (0, 0, 3) extends 3 units along the Z axis. Look closely at the grid in the Unity editor—the vector is exactly 3 units long. Try clicking and dragging the Z Rotation label in the Script component in the Inspector. The ray will get larger or smaller as you drag. When the Z value in the vector is negative, the ball rotates in the other direction.

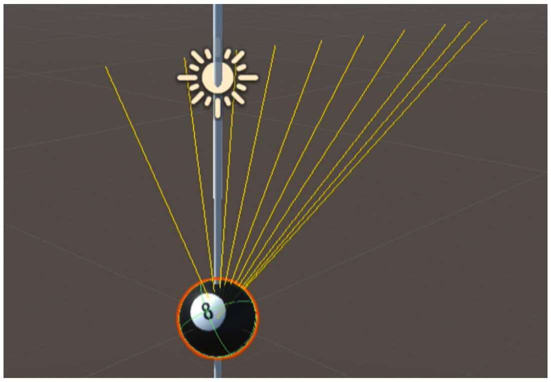
- Set the Z Rotation back to 3. Experiment with dragging the X Rotation and Y Rotation values to see what they do to the ray. Make sure to reset the Transform component each time you change them.
- Use the Hand tool and the Scene Gizmo to get a better view. Click the X cone on the Scene Gizmo to set it to the view from the right. Keep clicking the cones on the Scene Gizmo until you see the view from the front. It's easy to get lost—you can **reset the Wide layout to get back to a familiar view**.

Add a duration to the ray so it leaves a trail

You can add a fourth argument to your Debug.DrawRay method call that specifies the number of seconds the ray should stay on the screen. Add **.5f** to make each ray stay on screen for half a second:

```
    Debug.DrawRay(Vector3.zero, axis, Color.yellow, .5f);
```

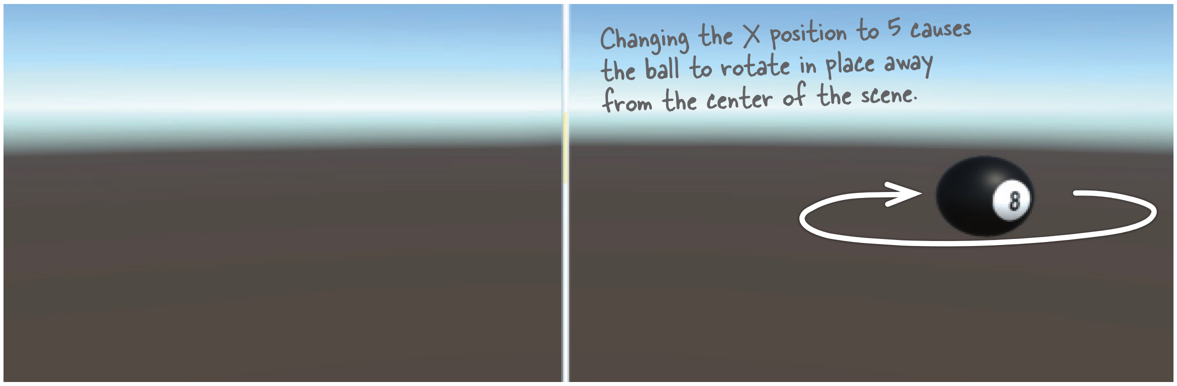
Now run the game again and switch to the Scene view. Now when you drag the numbers up and down, you'll see a trail of rays left behind. This looks really interesting, but more importantly, it's a great tool to visualize 3D vectors.



Making your ray leave a trail is a good way to help you develop an intuitive sense of how 3D vectors work.

Rotate your ball around a point in the scene

Your code calls the transform.Rotate method to rotate your ball around its center, which changes its X, Y, and Z rotation values. **Select Sphere in the Hierarchy window and change its X position to 5** in the Transform component. Then **use the context menu (⋮) in the BallBehaviour Script component** to reset its fields. Run the game again—now the ball will be at position (5, 0, 0) and rotating around its own Y axis.


Changing the X position to 5 causes the ball to rotate in place away from the center of the scene.

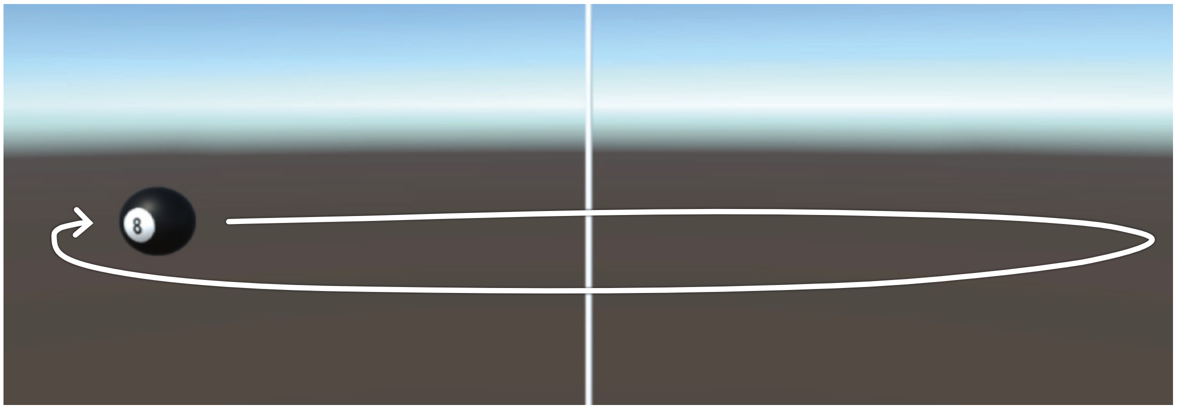
Let's modify the Update method to use a different kind of rotation. Now we'll make the ball rotate around the center point of the scene, coordinate (0, 0, 0), using the **transform.RotateAround method**, which rotates a GameObject around a point in the scene. (This is *different* from the transform.Rotate method you used earlier, which rotates a GameObject around its center.) Its first parameter is the point to rotate around. We'll use **Vector3.zero** for that parameter, which is a shortcut for writing `new Vector3(0, 0, 0)`.

Here's the new Update method:

This new Update method rotates the ball around the point (0, 0, 0) in the scene.

```
void Update()
{
    Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
    transform.RotateAround(Vector3.zero, axis, DegreesPerSecond * Time.deltaTime);
    Debug.DrawRay(Vector3.zero, axis, Color.yellow, .5f);
}
```

Now run your code. This time it rotates the ball in a big circle around the center point:



Use Unity to take a closer look at rotation and vectors

You're going to be working with 3D objects and scenes in the rest of the Unity Labs throughout the book. Even those of us who spend a lot of time playing 3D video games don't have a perfect feel for how vectors and 3D objects work, and how to move and rotate in a 3D space. Luckily, Unity is a great tool to **explore how 3D objects work**. Let's start experimenting right now.

While your code is running, try changing parameters to experiment with the rotation:

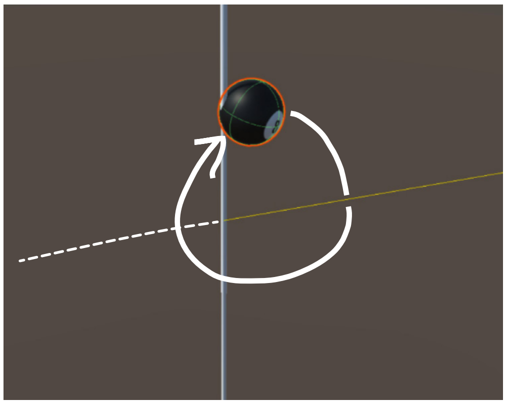
★ **Switch back to the Scene view** so you can see the yellow ray that Debug.DrawRay renders in your BallBehaviour.Update method.

★ Use the Hierarchy window to **select the Sphere**. You should see its components in the Inspector window.

★ Change the **X Rotation, Y Rotation, and Z Rotation values** in the Script component to **10** so you see the vector rendered as a long ray. Use the Hand tool (Q) to rotate the Scene view until you can clearly see the ray.

★ Use the Transform component's context menu (⋮) to **reset the Transform component**. Since the center of the sphere is now at the zero point in the scene, (0, 0, 0), it will rotate around its own center.

★ Then **change the X position in** the Transform component to **2**. The ball should now be rotating around the vector. You'll see the ball cast a shadow on the Y axis cylinder as it flies by.



While the game is running, set the X, Y, and Z Rotation fields in the BallBehaviour Script component to 10, reset the sphere's Transform component, and change its X position to 2—as soon as you do, it starts rotating around the ray.

Try **repeating the last three steps** for different values of X, Y, and Z rotation, resetting the Transform component each time so you start from a fixed point. Then try clicking the rotation field labels and dragging them up and down—see if you can get a feel for how the rotation works.

Unity is a great tool to explore how 3D objects work by modifying properties on your GameObjects in real time.

Get creative!

This is your chance to **experiment on your own with C# and Unity**. You've seen the basics of how you combine C# and Unity GameObjects. Take some time and play around with the different Unity tools and methods that you've learned about in the first two Unity Labs. Here are some ideas:

- ★ Add cubes, cylinders, or capsules to your scene. Attach new scripts to them—make sure you give each script a unique name!—and make them rotate in different ways.
- ★ Try putting your rotating GameObjects in different positions around the scene. See if you can make interesting visual patterns out of multiple rotating GameObjects.
- ★ Try adding a light to the scene. What happens when you use transform.rotateAround to rotate the new light around various axes?
- ★ Here's a quick coding challenge: try using += to add a value to one of the fields in your BallBehaviour script. Make sure you multiply that value by Time.deltaTime. Try adding an `if` statement that resets the field to 0 if it gets too large.

Before you run the code, try to figure out what it will do. Does it act the way you expected it to act? Trying to predict how the code you added will act is a great technique for getting better at C#.

Take the time to experiment with the tools and techniques you just learned. This is a great way to take advantage of Unity and Visual Studio as tools for exploration and learning.

BULLET POINTS

- The **Scene Gizmo** always displays the camera's orientation.
- You can **attach a C# script** to any GameObject. The script's Update method will be called once per frame.
- The **transform.Rotate method** causes a GameObject to rotate a number of degrees around an axis.
- Inside your Update method, multiplying any value by **Time.deltaTime** turns it into that value per second.
- You can **attach** the Visual Studio debugger to Unity to debug your game while it's running. It will stay attached to Unity even when your game is not running.
- Adding a **Hit Count condition** to a breakpoint to makes it break after the statement has executed a certain number of times.
- A **field** is a variable that lives inside of a class outside of its methods, and it retains its value between method calls.
- Adding public fields to the class in your Unity script makes the Script component show **input boxes that let you modify those fields**. It adds spaces between capital letters in the field names to make them easier to read.
- You can create 3D vectors using **`new Vector3`**. (You learned about the `new` keyword in Chapter 3.)
- The **Debug.DrawRay method** draws a vector in the Scene view (but <u>not</u> 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.

i appendix i: ASP.NET Core Blazor projects

Visual Studio for Mac Learner's Guide

Your Mac is a first-class citizen of the C# and .NET world.

We wrote *Head First C#* with our Mac readers in mind, and that's why we created this special **learner's guide** just for you. Most projects in this book are .NET Core console apps, which work on **both Windows and Mac**. Some chapters have a project built with a technology used for desktop Windows apps. This learner's guide has **replacements** for all of those projects—including a *complete replacement for Chapter 1*—that use **C# to create Blazor WebAssembly apps** that run in your browser and are equivalent to the Windows apps. You'll do it all with **Visual Studio for Mac**, a great tool for writing code and a **valuable learning tool** for exploring C#. Let's dive right in and get coding!

Why you should learn C#

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

Visual Studio is your gateway to C#

If you haven't installed Visual Studio 2019 yet, this is the time to do it.

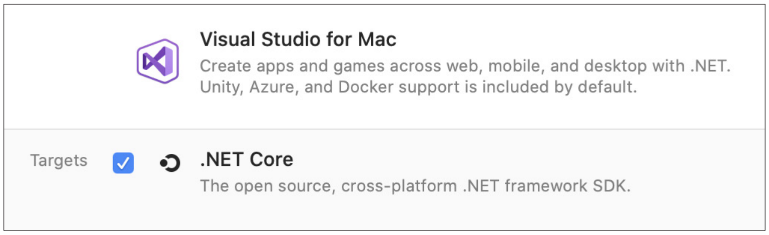
Go to https://visualstudio.microsoft.com and **download Visual Studio for Mac**. (If it's already installed, run the Visual Studio for Mac Installer to update your installed options.)

Install .NET Core

Once you've downloaded the Visual Studio for Mac installer, run it to install Visual Studio. Make sure the **.NET Core** target is checked.

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

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



Visual Studio for Mac
Create apps and games across web, mobile, and desktop with .NET. Unity, Azure, and Docker support is included by default.

Targets ✓ .NET Core
The open source, cross-platform .NET framework SDK.

You can also use Visual Studio for Windows to build Blazor web applications

Most of the projects in *Head First C#* are .NET Core console apps, which you can create using either macOS or Windows. Some of the chapters also include a Windows desktop app project that's built using Windows Presentation Foundation (WPF). Since WPF is a technology that only works with Windows, we wrote this *Visual Studio for Mac Learner's Guide* so you can create equivalent projects on your Mac using web technologies—specifically, ASP.NET Core Blazor WebAssembly projects.

What if you're a Windows reader and want to learn to build rich web applications using Blazor? Then you're in luck! ***You can build the projects in this guide using Visual Studio for Windows***. Go to the Visual Studio installer and make sure that **"ASP.NET and web development" option is checked**. Your IDE screenshots won't exactly match the ones in this guide, but all of the code will be the same.

You'll be using HTML and CSS in the web application projects throughout this guide, but you don't need to know HTML or CSS.

This book is about learning C#. In the projects in this guide, you'll be creating Blazor web applications that include pages designed using HTML and CSS. Don't worry if you haven't used HTML or CSS before—you don't need any prior web design knowledge to use this book. We'll give you everything you need to create all of the pages for the web application projects. However, be warned: you might pick up a little HTML knowledge along the way.

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

You could use TextEdit 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. **Write and run your C# code.** The Visual Studio IDE is one of the easiest-to-use tools out there for writing code. The team at Microsoft who develop it have put a huge amount of work into making your job of writing code as easy as possible.

3. **Build visually stunning web applications.** In this Visual Studio for Mac Learner's Guide, you'll be building web applications that run in your browser. You'll use **Blazor**, a technology that lets you build interactive web apps using C#. When you **combine C# with HTML and CSS**, you've got an impressive toolkit for web development.

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.

We'll often refer to Visual Studio as just "the IDE" throughout this book.

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 for Mac

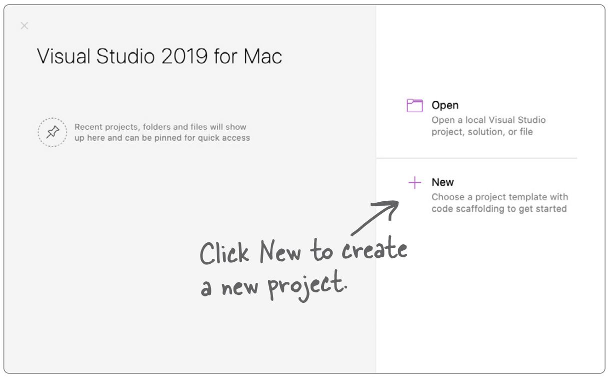
The best way to learn C# is to start writing code, so we're going to use Visual Studio to **create a new project**... and start writing code immediately!

Do this!

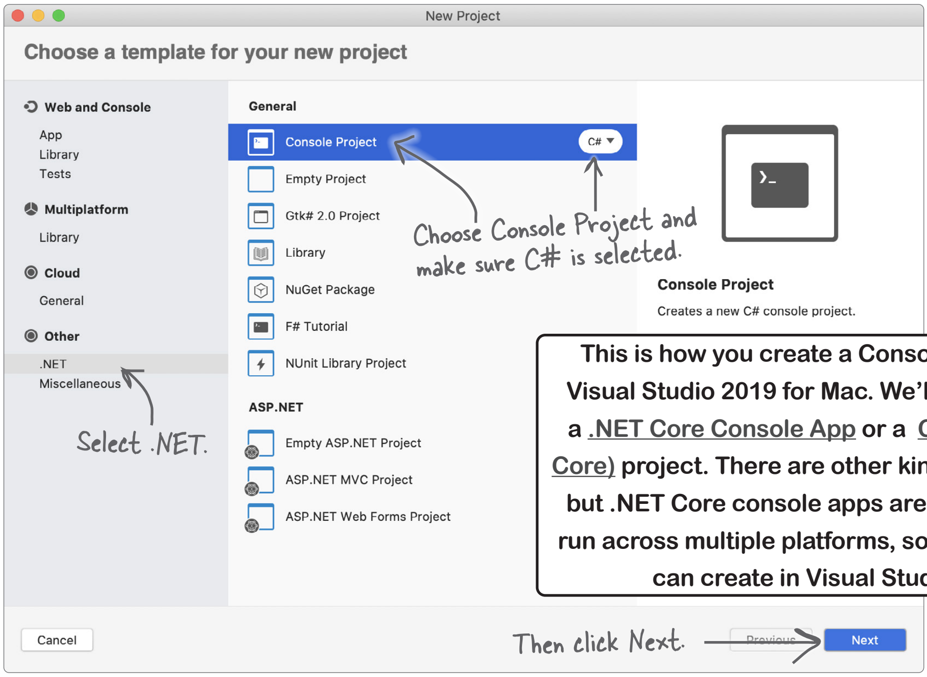
When you see Do this! (or Now do this!, or Debug this!, etc.), go to Visual Studio and follow along. We'll tell you exactly what to do, and point out what to look for to get the most out of the example we show you.

1 **Create a new Console Project.**

Start up Visual Studio 2019 for Mac. When it starts, it shows you a window that lets you create a new project or open an existing one. **Click New** to create a new project. Don't worry if you dismiss the window—you can always get it back by choosing *File >> New Solution... (⇧⌘N)* from the menu.


Visual Studio 2019 for Mac
Recent projects, folders and files will show up here and can be pinned for quick access
Open
Open a local Visual Studio project, solution, or file
New
Choose a project template with code scaffolding to get started
Click New to create a new project.

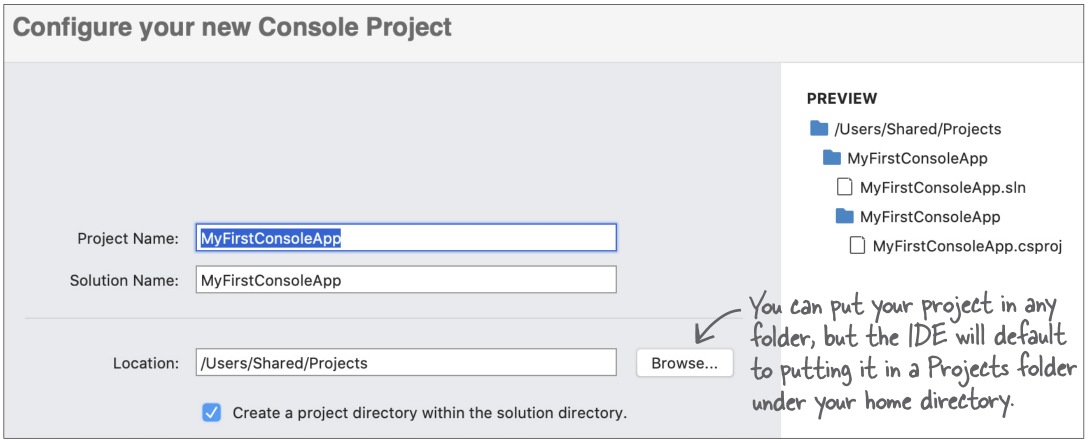
Select .NET from the panel on the left, then choose **Console Project**:



This is how you create a Console App project in Visual Studio 2019 for Mac. We'll sometimes call it a .NET Core Console App or a Console App (.NET Core) project. There are other kinds of console apps, but .NET Core console apps are the only ones that run across multiple platforms, so that's the kind you can create in Visual Studio for Mac.

2 Name your project MyFirstConsoleApp.

Enter **MyFirstConsoleApp** in the Project Name box and click the **Create button** to create the project.



Configure your new Console Project

Project Name: MyFirstConsoleApp

Solution Name: MyFirstConsoleApp

Location: /Users/Shared/Projects

Browse...

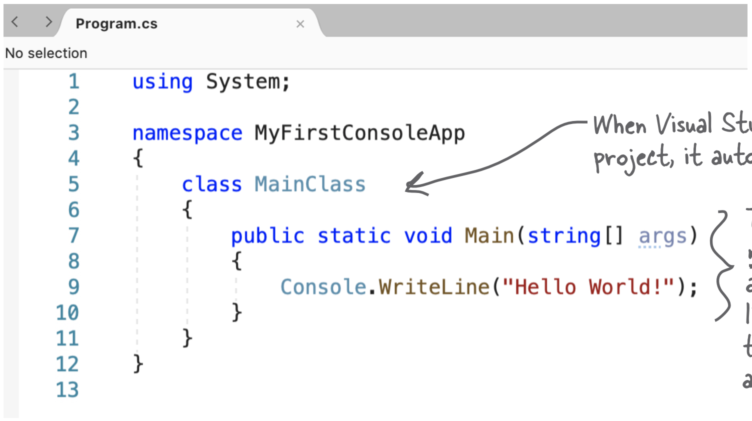
✓ Create a project directory within the solution directory.

PREVIEW
/Users/Shared/Projects
MyFirstConsoleApp
MyFirstConsoleApp.sln
MyFirstConsoleApp
MyFirstConsoleApp.csproj

You can put your project in any folder, but the IDE will default to putting it in a Projects folder under your home directory.

3 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 opens and displays a file called *Program.cs* with this code:



Program.cs

No selection

```
using System;

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

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

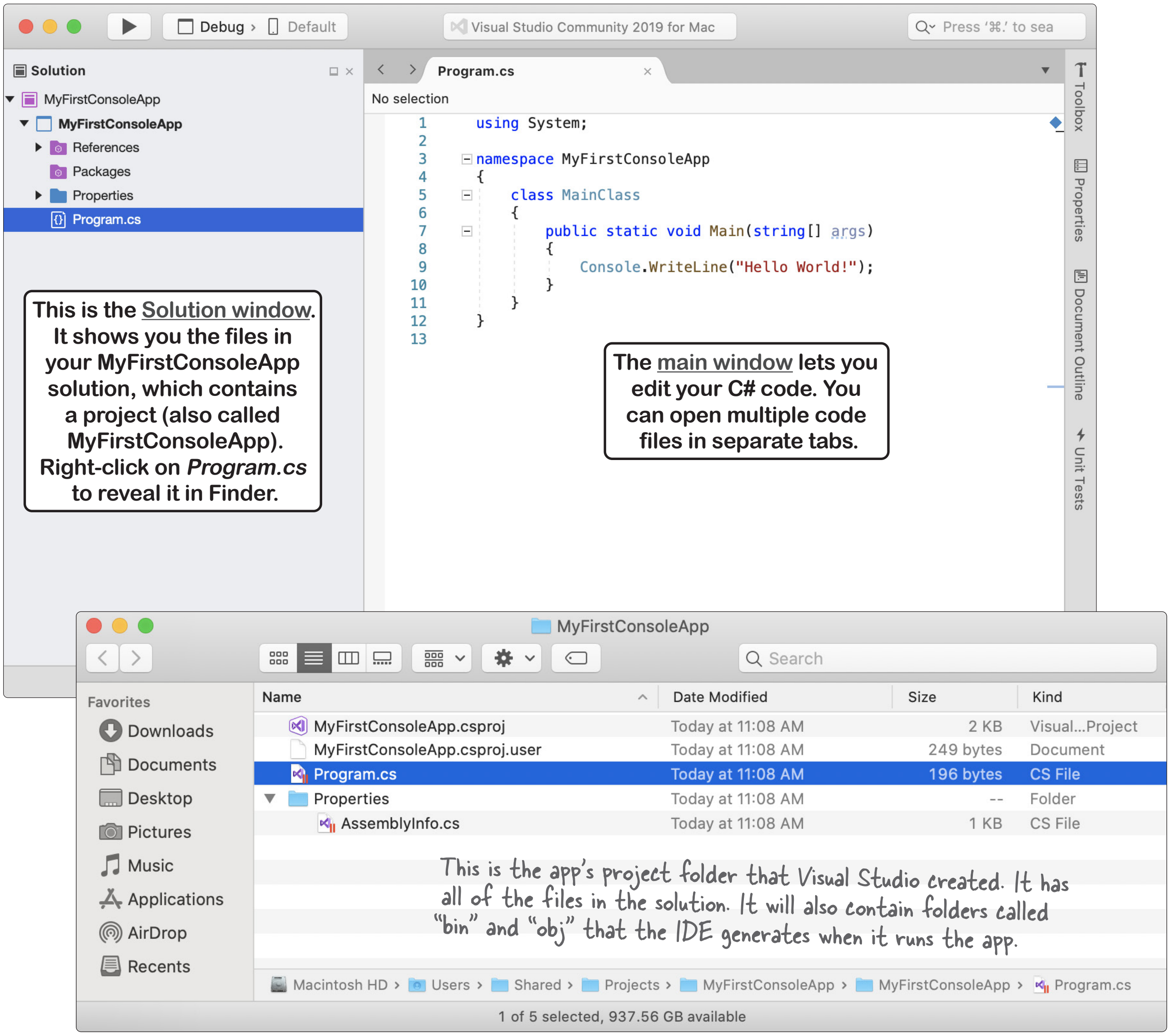
The class starts out with a method called Main, which contains a single statement that writes a line of text to the console. We'll take a much closer look at classes and methods in Chapter 2.

The main class's name is different in Windows.

When you create a console app in Visual Studio for Windows, it generates almost exactly the same code as Visual Studio for Mac. The one difference is that on a Mac the main class is called MainClass, while on Windows the main class is called Program. That won't make a difference in most of the projects in this book. We'll make sure to point out any place that it does.

Use the Visual Studio IDE to explore your app

1. **Explore the Visual Studio IDE—and the files that it created for you.**
When you created the new project, Visual Studio created several files for you automatically and bundled them into a **solution**. The Solution window on the left side of the IDE shows you these files, with the solution (MyFirstConsoleApp) at the top. The solution contains a **project** that has the same name as the solution.



```
using System;

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

Name	Date Modified	Size	Kind
MyFirstConsoleApp.csproj	Today at 11:08 AM	2 KB	Visual...Project
MyFirstConsoleApp.csproj.user	Today at 11:08 AM	249 bytes	Document
Program.cs	Today at 11:08 AM	196 bytes	CS File
Properties	Today at 11:08 AM	--	Folder
AssemblyInfo.cs	Today at 11:08 AM	1 KB	CS File

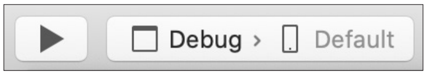
This is the Solution window. It shows you the files in your MyFirstConsoleApp solution, which contains a project (also called MyFirstConsoleApp). Right-click on *Program.cs* to reveal it in Finder.

The main window lets you edit your C# code. You can open multiple code files in separate tabs.

This is the app's project folder that Visual Studio created. It has all of the files in the solution. It will also contain folders called "bin" and "obj" that the IDE generates when it runs the app.

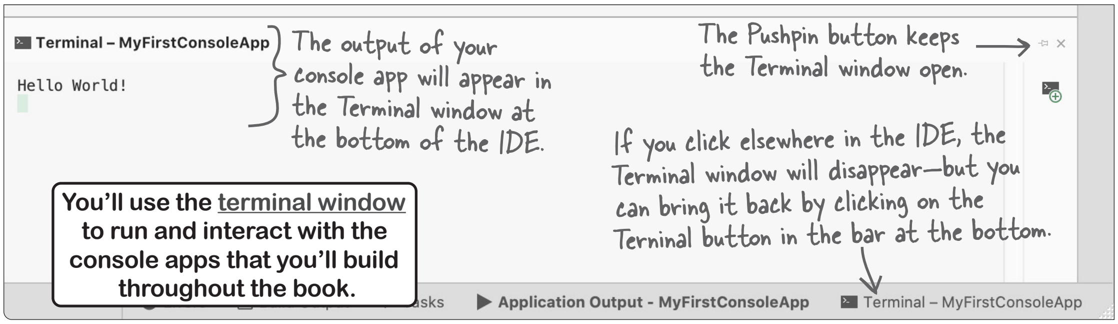
2 **Run your new app.**

The app that Visual Studio for Mac created for you is ready to run. At the top of the Visual Studio IDE, find the Run button (with a "play" triangle). **Click that button** to run your app:



3 **Look at your program's output.**

When you run your program, the **Terminal window** will appear at the bottom of the IDE and display the output of the program:


Terminal – MyFirstConsoleApp
The output of your console app will appear in the Terminal window at the bottom of the IDE.
The Pushpin button keeps the Terminal window open.
If you click elsewhere in the IDE, the Terminal window will disappear—but you can bring it back by clicking on the Terminal button in the bar at the bottom.
You'll use the terminal window to run and interact with the console apps that you'll build throughout the book.
Application Output - MyFirstConsoleApp
Terminal – MyFirstConsoleApp

```
Hello World!
```

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

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

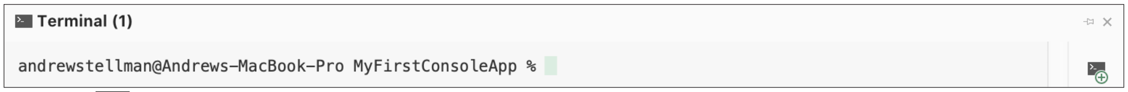
```
Hello World!
```

Click anywhere in the code to hide the Terminal window. Then press the Terminal – MyFirstConsoleApp button at the bottom of the IDE to open it again—you'll see the same output from your program. The IDE automatically hides the Terminal window after your app exits.

Press the Run button to run your program again. Then choose Start Debugging from the Run menu, or use its shortcut (⌘↩). This is how you'll run all of the Console App projects throughout the book.

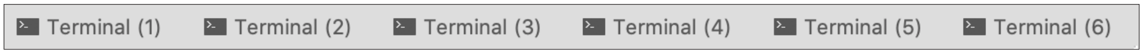
IDE Tip: Open a terminal inside the IDE

The Terminal window shows the output of your console apps—but it does more than that. Click the button at the right of the Terminal window, or choose *View >> Terminal* from the menu when your app isn't running. You'll see a macOS Terminal shell right inside the IDE, which you can use to execute macOS shell commands:



```
andrewstellman@Andrews-MacBook-Pro MyFirstConsoleApp %
```

Click the button a few more times—the IDE will open several Terminals at once. You can switch between them using the *View >> Other Windows* menu item or the buttons on the bar at the bottom of the IDE:

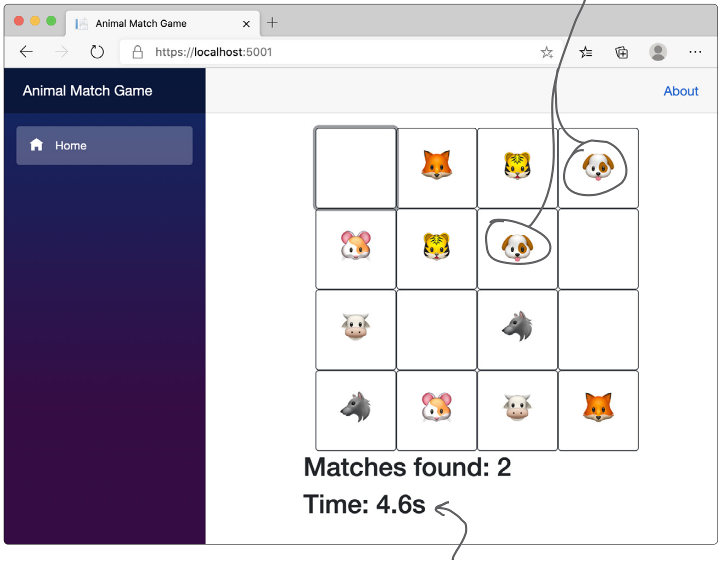


Let's build a game!

You've built your first C# app, and that's great! Now that you've done that, let's build something a little more complex. We're going to build an **animal matching game**, where a player is shown a grid of 16 animals and needs to click on pairs to make them disappear.

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

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



Matches found: 2

Time: 4.6s

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

Building different kinds of projects is an important tool in your C# learning toolbox. We chose Blazor for the Mac projects in this book because it gives you tools to design rich web applications that run on any modern browser.

But C# isn't just for web development and console apps! Every project in this Mac learner's guide has an equivalent Windows project.

Are you a Windows user, but still want to learn Blazor and build web applications with C#? Well then, you're in luck! All of the projects in the Mac Learner's Guide can also be done with Visual Studio for Windows.

Your animal matching game is a Blazor WebAssembly app

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

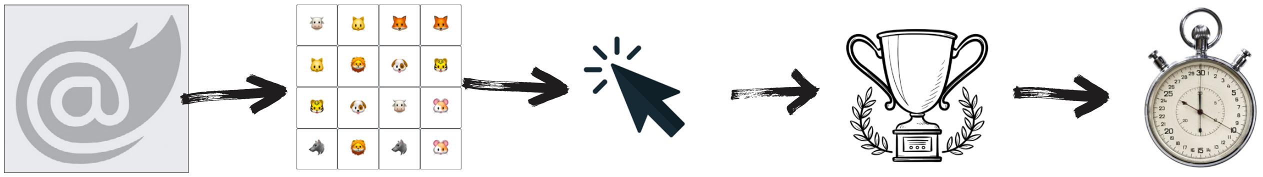
By the time you're done with this project, you'll be a lot more familiar with the tools that you'll rely on throughout this book to learn and explore C#.

Here's how you'll build your game

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

1. First you'll create a new Blazor WebAssembly App project in Visual Studio.
2. Then you'll lay out the page and write C# code to shuffle the animals.
3. The game needs to let the user click on pairs of emoji to match them.
4. You'll write more C# code to detect when the player has won the game.
5. Finally, you'll make the game more exciting by adding a timer.

This project can take anywhere from 15 minutes to an hour, depending on how quickly you type. We learn better when we don't feel rushed, so give yourself plenty of time.



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 <u>any kind of project</u>, not just video games.

Game design... and beyond

What is a game?

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

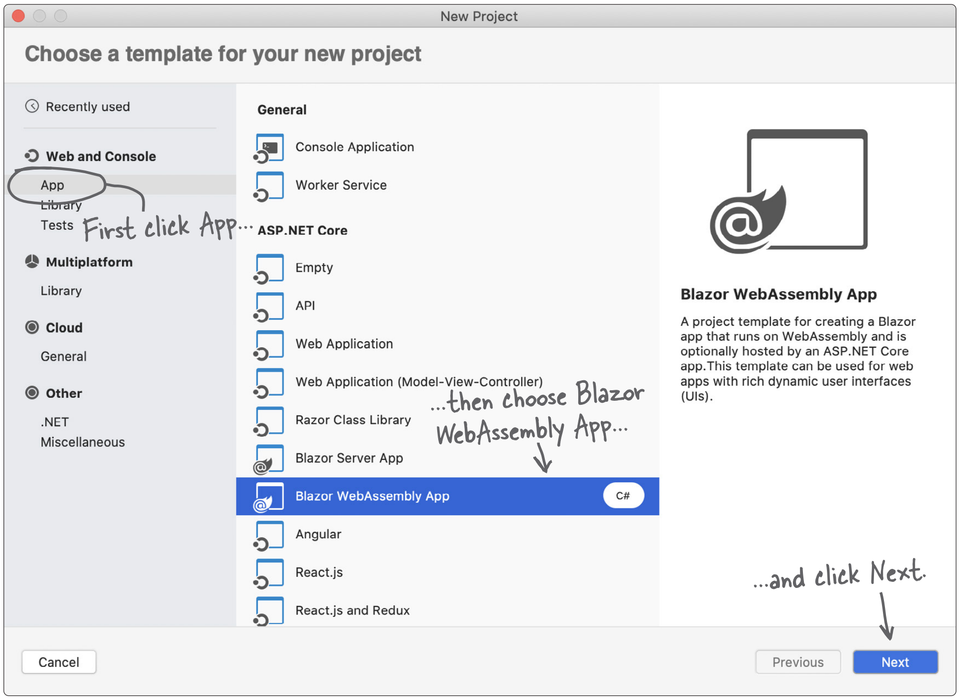
- Do all games have a **winner**? Do they always end? Not necessarily. What about a flight simulator? A game where you design an amusement park? What about a game like The Sims?
- Are games always **fun**? Not for everyone. Some players like a "grind" where they do the same thing over and over again;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

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

Create a Blazor WebAssembly App in Visual Studio

The first step in building your game is to create a new project in Visual Studio.

1. Choose **File >> New Solution...** (⇧⌘N) from the menu to bring up the New Project window. It's the same way you started out your Console App project.

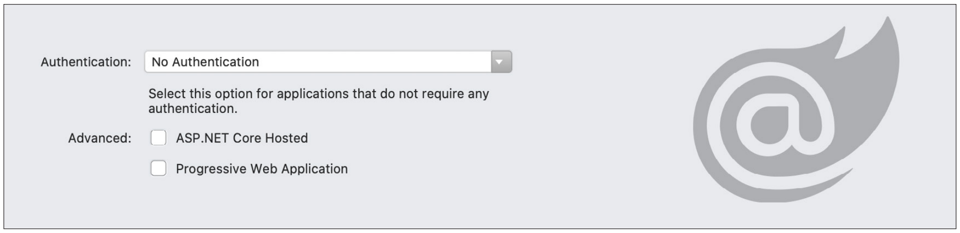


Blazor WebAssembly App

A project template for creating a Blazor app that runs on WebAssembly and is optionally hosted by an ASP.NET Core app.This template can be used for web apps with rich dynamic user interfaces (UIs).

Click **App** under "Web and Console" on the left, then choose **Blazor WebAssembly App** and click **Next**.

2. The IDE will give you a page with options.



Authentication: No Authentication

Select this option for applications that do not require any authentication.

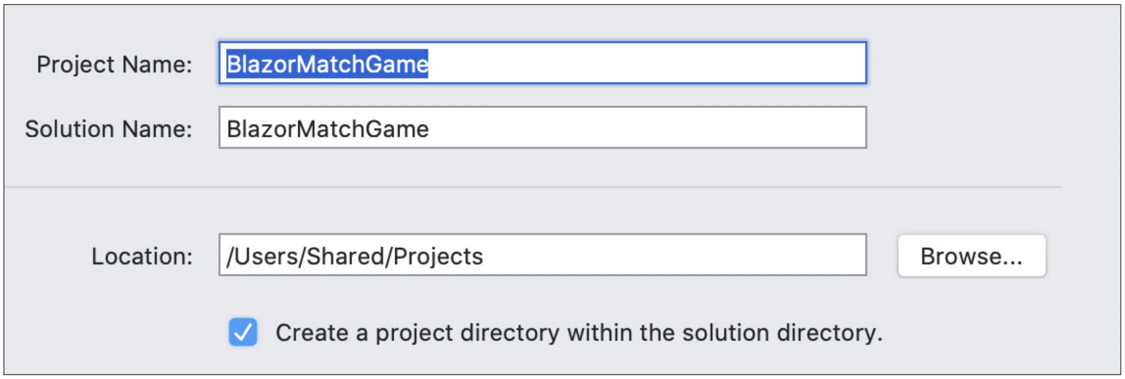
Advanced: ☐ ASP.NET Core Hosted

☐ Progressive Web Application

Leave all of the options set to their default values and click **Next**.

If you run into any trouble with this project, go to our GitHub page and look for a link to a video walkthrough: *https://github.com/head-first-csharp/fourth-edition*

3. Enter **BlazorMatchGame** as the project name, just like you did with your Console App project.



Project Name: BlazorMatchGame

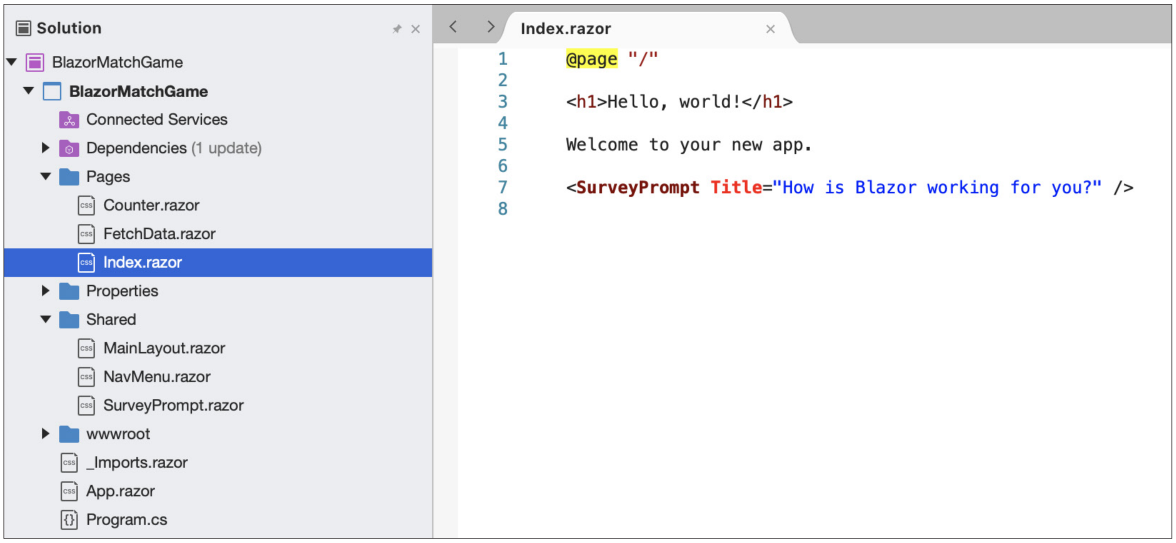
Solution Name: BlazorMatchGame

Location: /Users/Shared/Projects

✓ Create a project directory within the solution directory.

Then **click Create** to create the project solution.

4. The IDE will create a new BlazorMatchGame project and show you its contents, just like it did with your first console app. **Expand the Pages folder** in the Solution window to view its contents, then **double-click *Index.razor*** to open it in an editor.



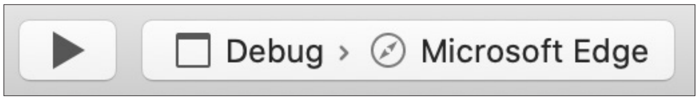
```
@page "/"

<h1>Hello, world!</h1>

Welcome to your new app.

<SurveyPrompt Title="How is Blazor working for you?" />
```

Run your Blazor web app in a browser

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

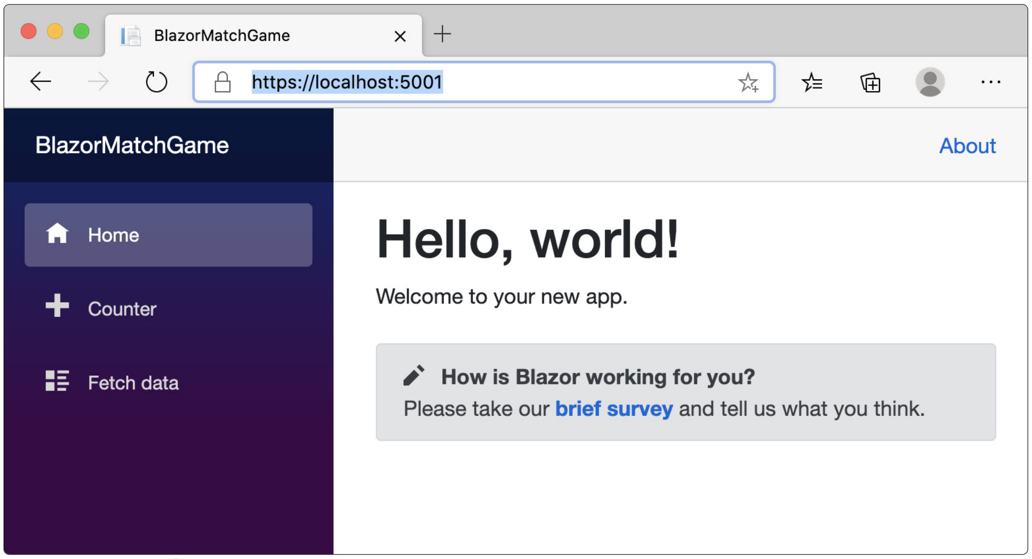
Do this!

1. **Choose the browser to run your web application.**
Find the triangle-shaped Run button at the top of the Visual Studio IDE:



Your default browser should be listed next to Debug >. Click the browser name to see a dropdown of installed browsers and **choose either Microsoft Edge or Google Chrome**.

2. **Run your web application.**
Click the **Run button** to start your application. You can also choose Start Debugging (⌘↩) from the Run menu. The IDE will first open a Build Output window (at the bottom, just like it opened the Terminal window), and then an Application Output window. After that, it will pop up a browser running your app.



Hello, world!

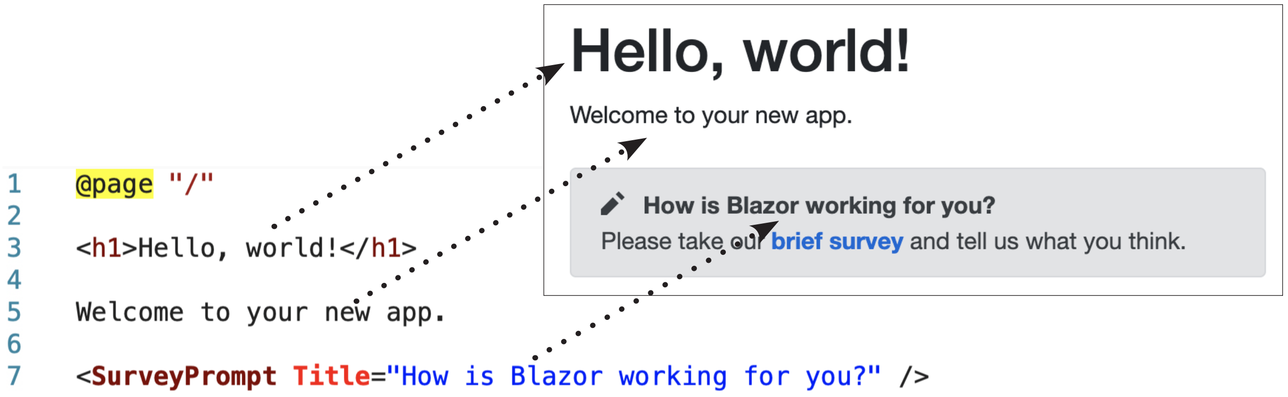
Welcome to your new app.

How is Blazor working for you?
Please take our **brief survey** and tell us what you think.

Run your web apps in Microsoft Edge or Google Chrome.

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

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



```
@page "/"

<h1>Hello, world!</h1>

Welcome to your new app.

<SurveyPrompt Title="How is Blazor working for you?" />
```

Hello, world!

Welcome to your new app.

How is Blazor working for you?
Please take our brief survey and tell us what you think.

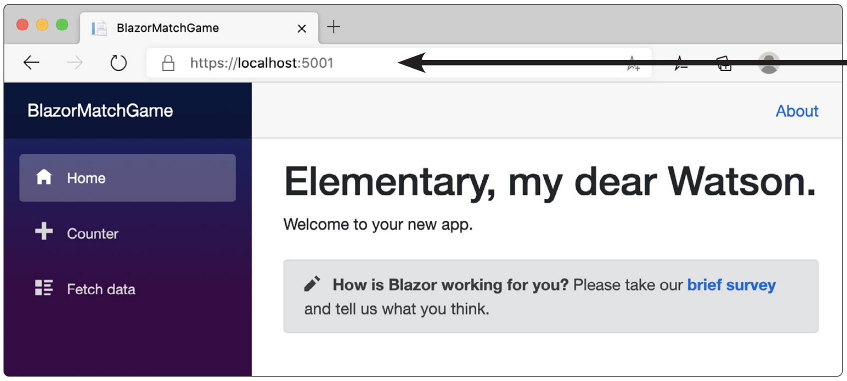
4 **Change "Hello, world!" to something else.**
Change the third line of the *Index.razor* file so it says something else:

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

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

Click the Stop button ■ or choose Stop (⇧⌘↩) from the Run menu. Now go back and reload your browser—since you stopped your app, it displays its "Site can't be reached" page.

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



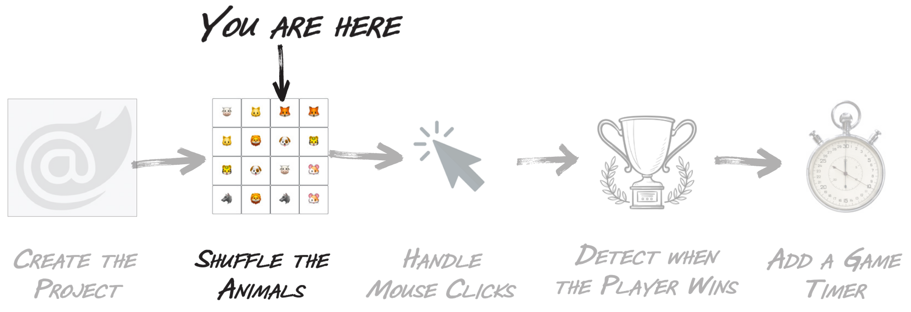
Elementary, my dear Watson.

Welcome to your new app.

How is Blazor working for you? Please take our brief survey and tell us what you think.

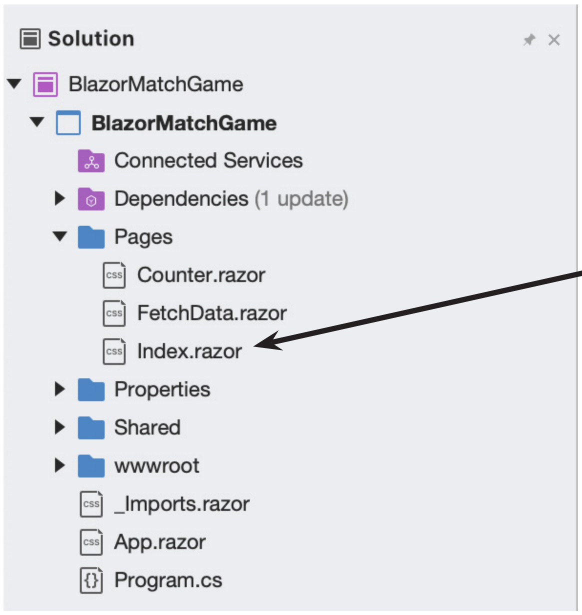
Try copying the URL from your browser, opening a new Safari window, and pasting it in. Your application will run there, too. Now you have two different browsers connecting to the same server.

Do you have extra instances of your browser open? Visual Studio opens a new browser each time you run your Blazor web app. Get in the habit of closing the browser (⌘Q) before you stop your app (⇧⌘↩).



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

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



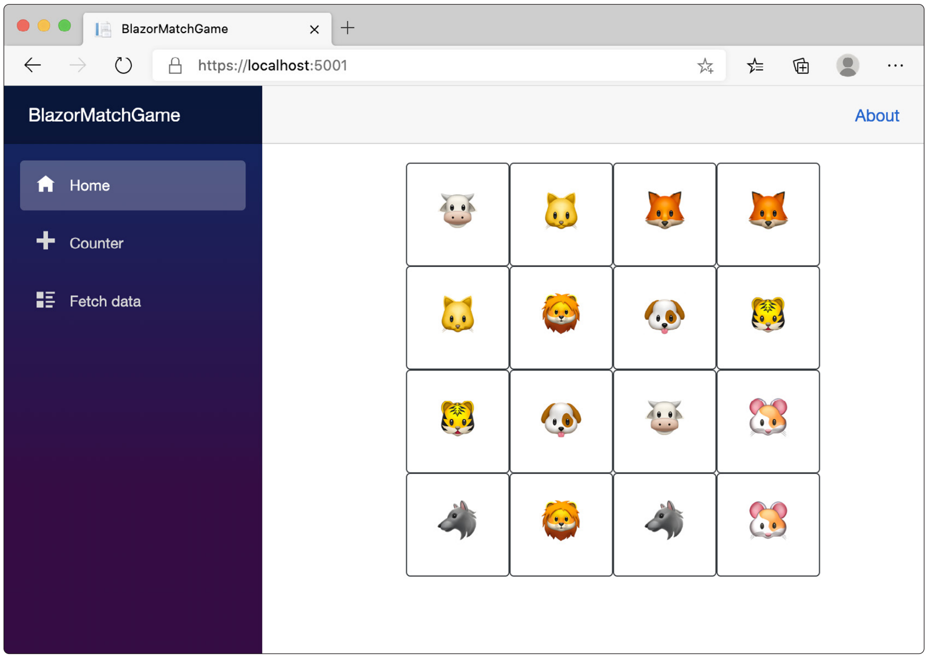
Solution
- BlazorMatchGame
  - BlazorMatchGame
    - Connected Services
    - Dependencies (1 update)
    - Pages
      - Counter.razor
      - FetchData.razor
      - Index.razor
    - Properties
    - Shared
    - wwwroot
    - _Imports.razor
    - App.razor
    - Program.cs

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

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

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.*



How the page layout in your animal matching game will work

Your animal matching game is laid out in a grid—or, at least, that's how it looks. It's actually made up of 16 square buttons. If you make your browser very narrow, it will rearrange them so they're in one long column.

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

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

The @ is how you tell a Razor page to include C# code. This is a `foreach` loop that runs the same code over and over again to generate a button for each emoji in a list of animal emoji.

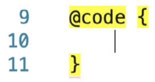
The `foreach` loop causes everything between the { and } to be repeated once for each emoji in a list of animal emoji, replacing `@animal` with each of the emoji in the list one by one. Since the list has 16 emoji, the result is a series of 16 buttons.

Visual Studio helps you write C# code

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

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

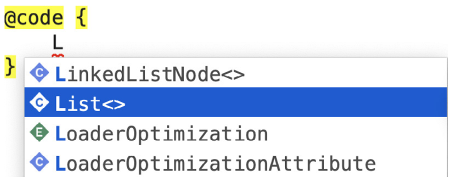
Start by **adding a `@code` block** to the end of your *Index.razor* file. (Keep the existing contents of the file there for now—you'll delete them later.) Go to the last line of the file and type `@code {`. The IDE will fill in the closing curly bracket `}` for you. Press Enter to add a line between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



```
@code {
    |
}
```

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

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



```
@code {
    L
}
```

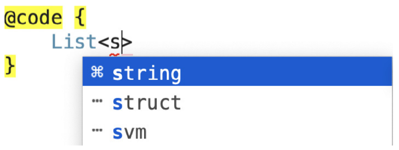
LinkedListNode<>
List<>
LoaderOptimization
LoaderOptimizationAttribute

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

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

③ Start creating a List to store your animal emoji.

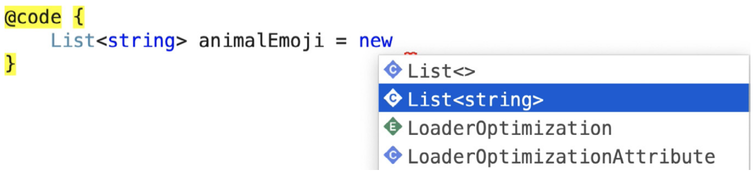
Type s to bring up another IntelliSense window:



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

string
struct
svm

Choose `string`—the IDE will add it between the brackets. Press the **right arrow and then the space bar**, then **type `animalEmoji = new`**. Press the space bar again to pop up another IntelliSense window. **Press Enter** to choose the default value, `List<string>`, from the options.



```
@code {
    List<string> animalEmoji = new
}
```

List<>
List<string>
LoaderOptimization
LoaderOptimizationAttribute

Your code should now look like this: `List<string> animalEmoji = new List<string>`

④ Finish creating the List of animal emoji.

Start by **adding a @code block** to the end of your *Index.razor* file. Go to the last line and **type @code {**. The IDE will fill in the closing curly bracket } for you. Press Enter to add a line between the brackets, then:

- ★ Type an **opening parenthesis (**—the IDE will fill in the closing one.
- ★ **Press the right arrow** to move past the parentheses.
- ★ Type an **opening curly bracket {**—again, the IDE will fill in the closing one.
- ★ Press Enter to add a line between the brackets, then **add a semicolon ;** after the closing bracket.

The last six lines at the very bottom of your *Index.razor* file should now look like this:

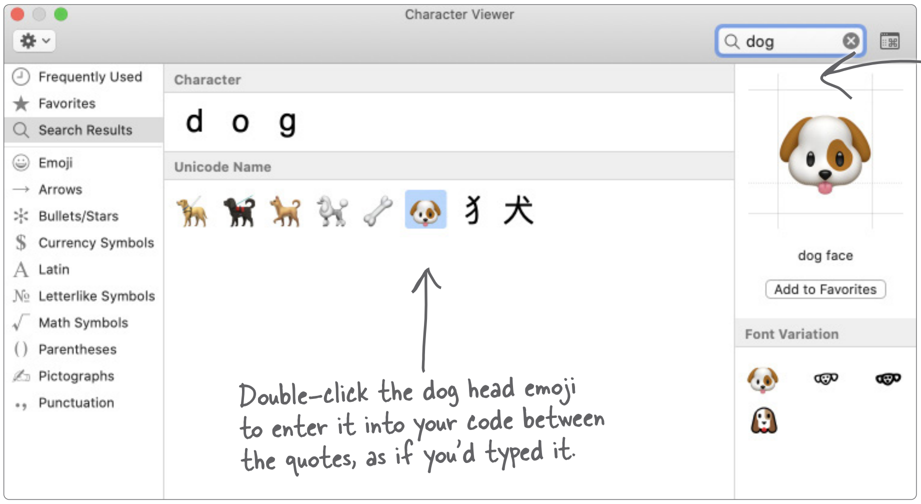
```
@code {
    List<string> animalEmoji = new List<string>()
    {

    };
}
```

For more about statements, refer to Chapter 2.

Congratulations—you've just created your first C# **statement**. But you're not done yet! You've created a list to hold the emoji to match. **Enter a quote "** on the blank line—the IDE will add a close quote.

⑤ Use the Character Viewer to enter emoji.

Next, **choose Edit >> Emoji & Symbols** (^⌘Space) from the menu to bring up the macOS Character Viewer. Position your cursor between the quotes, then go to the Character Viewer and **search for "dog"**:



Use the search box to search for the dog emoji. The Character Viewer will show several potential matches.

Double-click the dog head emoji to enter it into your code between the quotes, as if you'd typed it.

The last six lines at the bottom of your *Index.razor* file should now look like this:

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

See Chapter 8 to learn more about how a List works.

Finish creating your emoji list and display it in the app

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

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

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

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

IDE Tip: Indent lines

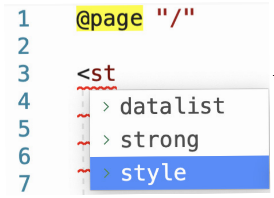
The IDE automatically indents your C# code for you as you enter it. But when you're entering the emoji or HTML tags, you might find that it doesn't quite indent them the way you want. You can easily fix that by selecting the text you want to indent and pressing ⇥ (Tab) to indent, or ⇧⇥ (Shift+Tab) to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

Replace the contents of the page

Delete these lines from the top of the page:

~~`<h1>Elementary, my dear Watson.</h1>`~~
~~`Welcome to your new app.`~~
~~`<SurveyPrompt Title="How is Blazor working for you?" />`~~

Then put your cursor on the third line of the page and **type `<st`**—the IDE will pop up an IntelliSense window:



```
@page "/"

<st
```

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

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

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

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

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

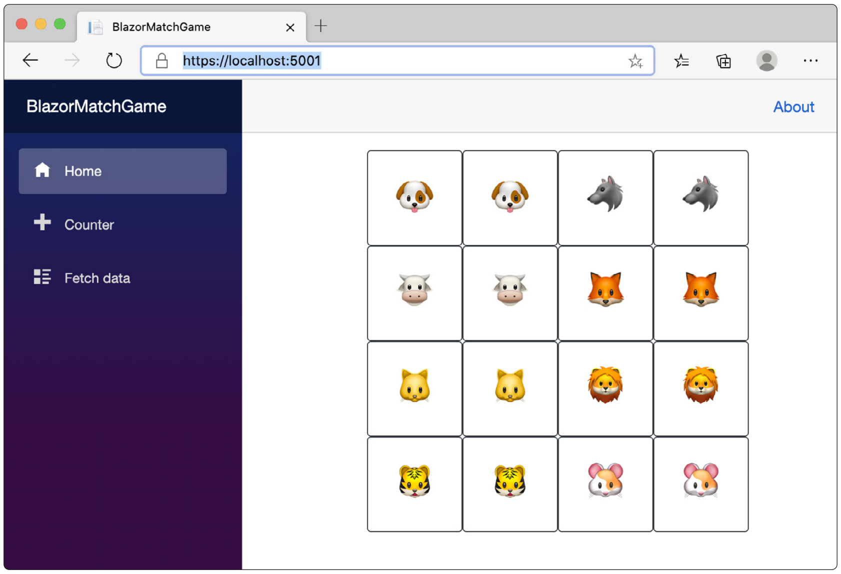
The matching game is made up of a series of buttons. This is a really simple CSS stylesheet to set the total width of the container, and the height and width of each button. Since the container is 400 pixels wide and each button is 100 pixels wide, the page will only allow four columns in a row before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

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

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

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

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



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

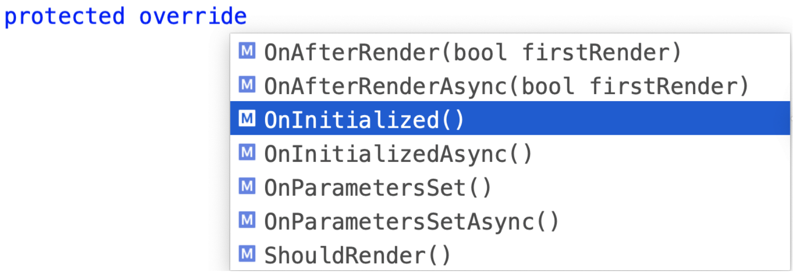
Shuffle the animals so they're in a random order

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

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

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

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



```
protected override
    OnAfterRender(bool firstRender)
    OnAfterRenderAsync(bool firstRender)
    OnInitialized()
    OnInitializedAsync()
    OnParametersSet()
    OnParametersSetAsync()
    ShouldRender()
```

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

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

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

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

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

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

You'll learn more about methods in Chapter 2.

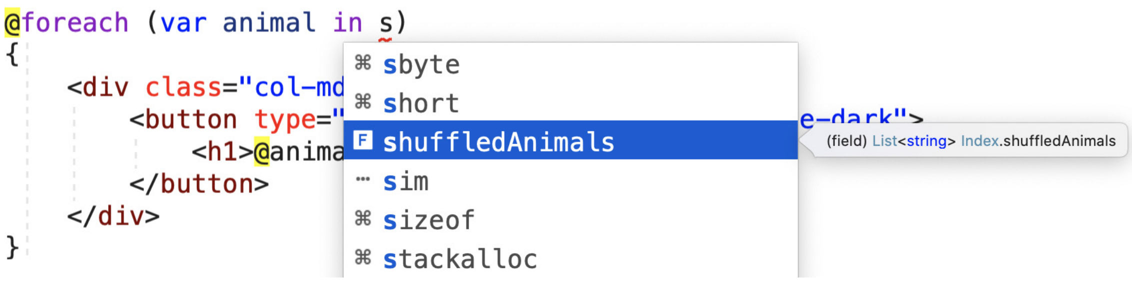
You'll learn a lot more about C# methods soon.

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

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

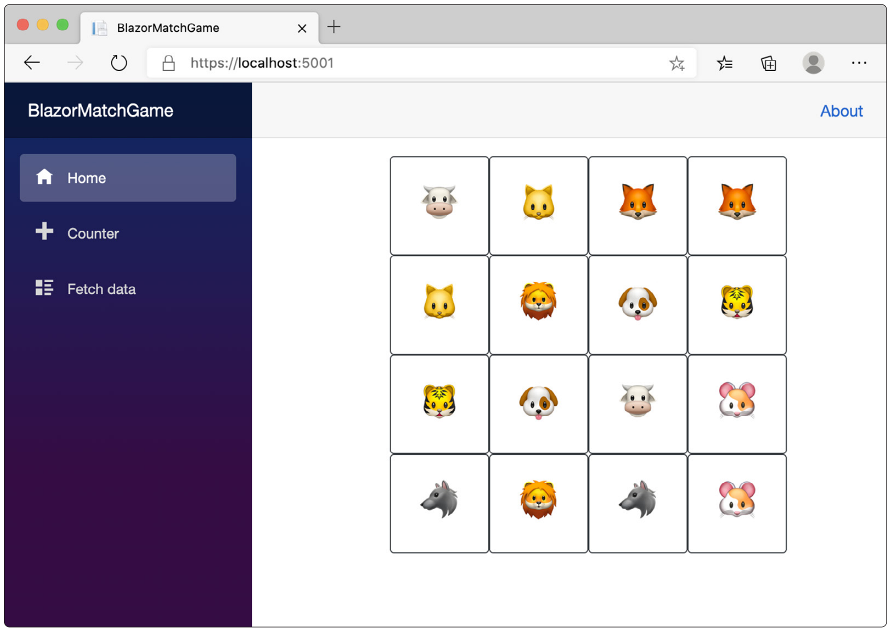
4 Scroll back up to the HTML and find this code: `@foreach (var animal in animalEmoji)`

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



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

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



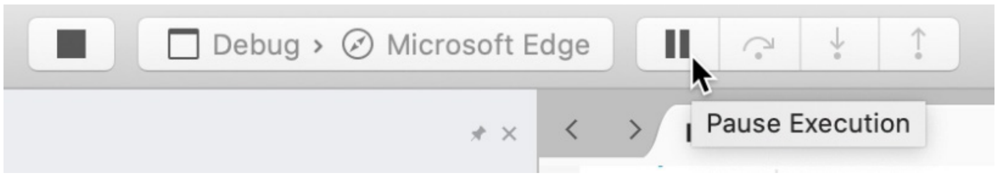
Again, make sure your app looks like this screenshot when you run it. Once it does, you'll know you entered all of the code without any typos. Don't move on until your game is reshuffling the animals every time you reload the browser page.

You're running your game in the debugger

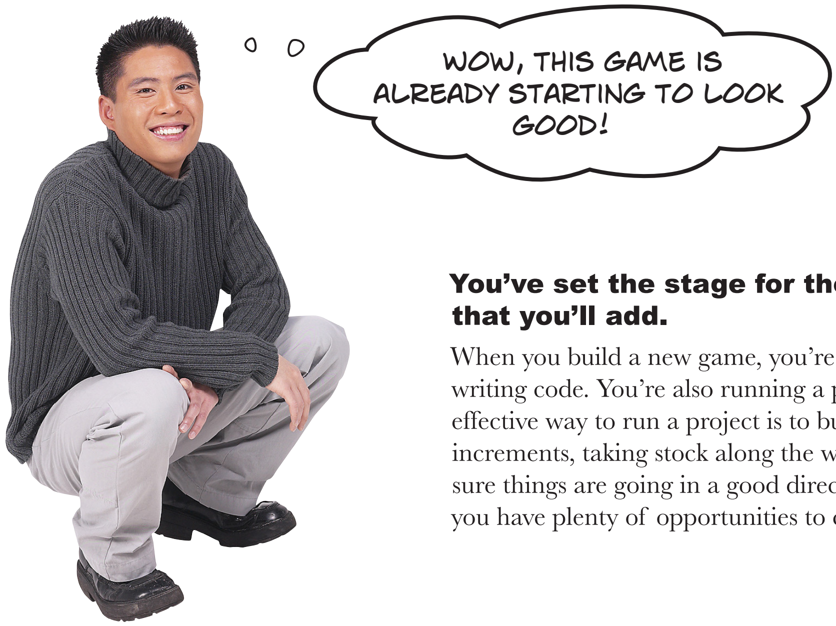
When you click the Run button ▶ or choose Start Debugging (⌘↩) from the Run menu to start your program running, you're putting Visual Studio into **debugging mode**.

You can tell that you're debugging an app when you see the **debug controls** appear in the toolbar. The Start button has been replaced with the square Stop button ■, the dropdown to choose which browser to launch is grayed out, and an extra set of controls has appeared.

Hover your mouse cursor over the Pause Execution button to see its tooltip:



You can stop your app clicking the Stop button or choosing Stop (⇧⌘↩) from the Run menu.



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

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

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

WHO DOES WHAT?

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

C# statement

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


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


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


private void SetUpGame()
{


    Random random = new Random();


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


}
```

What it does

Create a second list to store the shuffled emoji

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

The beginning of a method that sets up the game

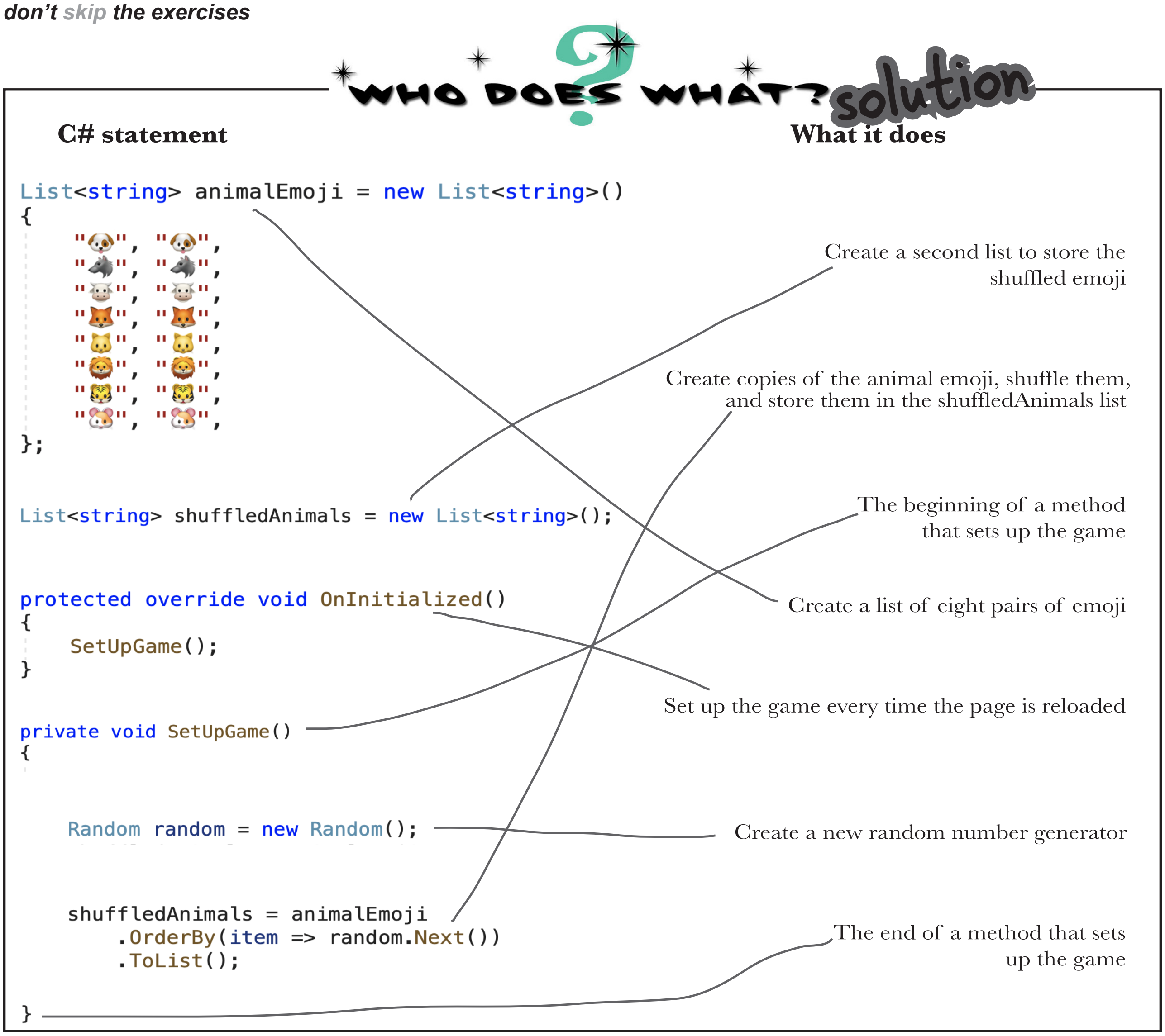
Create a list of eight pairs of emoji

Set up the game every time the page is reloaded

Create a new random number generator

The end of a method that sets up the game

WHO DOES WHAT? solution



C# statement

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

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

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

private void SetUpGame()
{

    Random random = new Random();

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

}
```

What it does

Create a second list to store the shuffled emoji

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

The beginning of a method that sets up the game

Create a list of eight pairs of emoji

Set up the game every time the page is reloaded

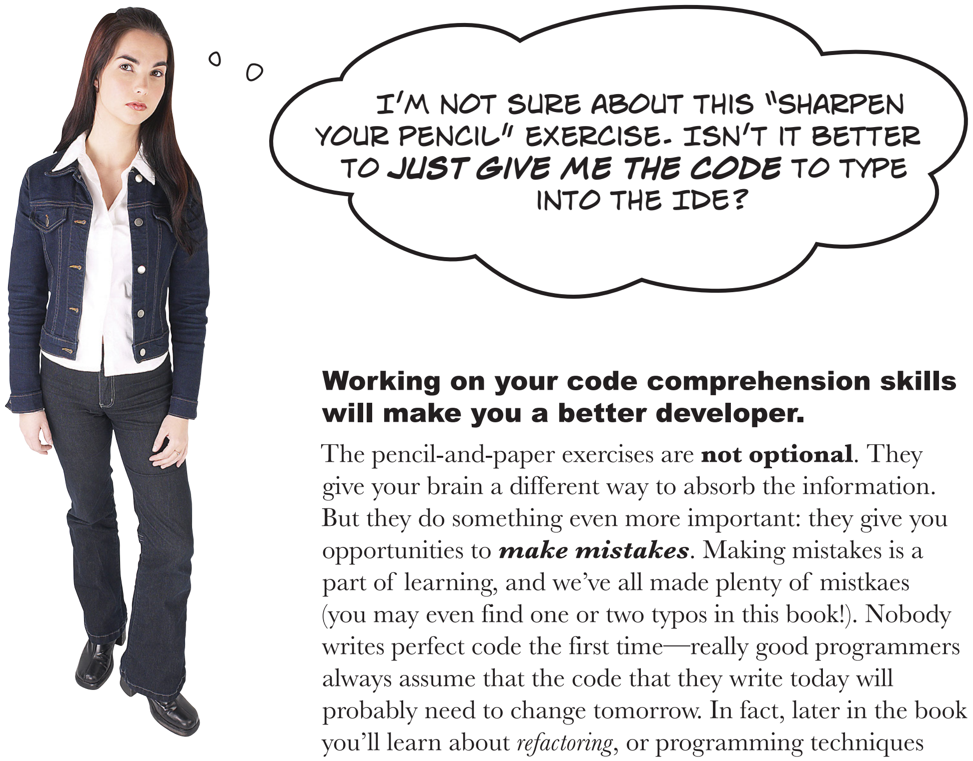
Create a new random number generator

The end of a method that sets up the game

MINI Sharpen your pencil

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

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


I'M NOT SURE ABOUT THIS "SHARPEN YOUR PENCIL" EXERCISE. ISN'T IT BETTER TO JUST GIVE ME THE CODE TO TYPE INTO THE IDE?

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

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

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

BULLET POINTS

- Visual Studio is **Microsoft's IDE**—or **integrated development environment**—that simplifies and assists in editing and managing your C# code files.
- **.NET Core console apps** are cross-platform apps that use text for input and output.
- **Blazor WebAssembly apps** let you build rich interactive web applications using C# code and HTML markup.
- The IDE's **AI-assisted IntelliSense** helps you enter code more quickly and accurately.
- Visual Studio can **run your Blazor app** in debugging mode, opening a browser to display your app.
- User interfaces for Blazor apps are designed in **HTML,** the markup language used to design web pages.
- **Razor** lets you add C# code directly into your HTML markup. Razor page files end with the *.razor* extension.
- Use an **@** to embed your C# code in a Razor page.
- A **`foreach` loop** in a Razor page lets you repeat a block of HTML code for each element in a list.

Add your new project to source control

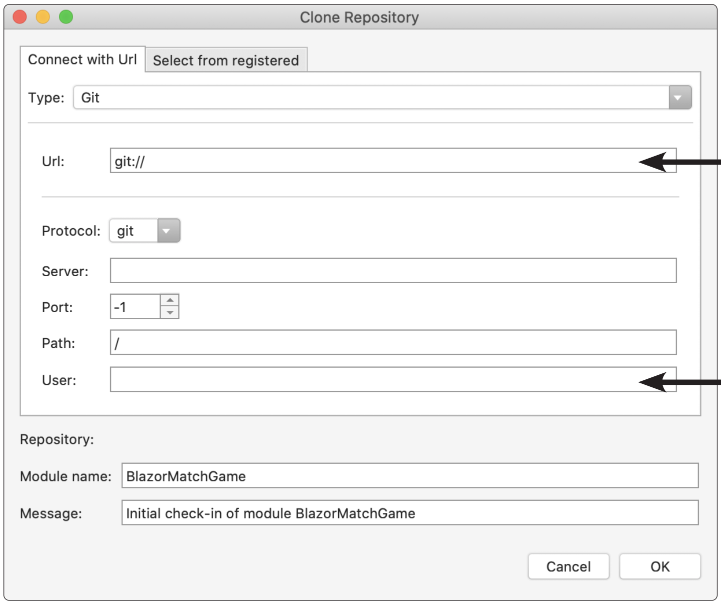
You're going to be building a lot of different projects in this book. Wouldn't it be great if there was an easy way to back them up and access them from anywhere? What if you make a mistake—wouldn't it be super convenient if you could roll back to a previous version of your code? Well, you're in luck! 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. Visual Studio makes it easy for you to add your projects to source control.

Git is a popular version control system, and Visual Studio will publish your source to any Git **repository** (or **repo**). We think **GitHub** is one of the easiest Git providers to use. You'll need a GitHub account to push code to it, so if you don't already have one, go to https://github.com and create it now.

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, the Git provider we're recommending. Maybe you just don't feel like doing it. Whatever the reason, you can skip this step—or, alternatively, you can publish it to a private repository if you want to keep a backup but don't want other people to find it.

Once you have your GitHub account set up, you can use the built-in version control features of the IDE. **Choose *Version Control* >> *Publish in Version Control...* from the menu** to bring up the Clone Repository window:



Type: Git

Url: git://

Protocol: git

Server:

Port: -1

Path: /

User:

Repository:

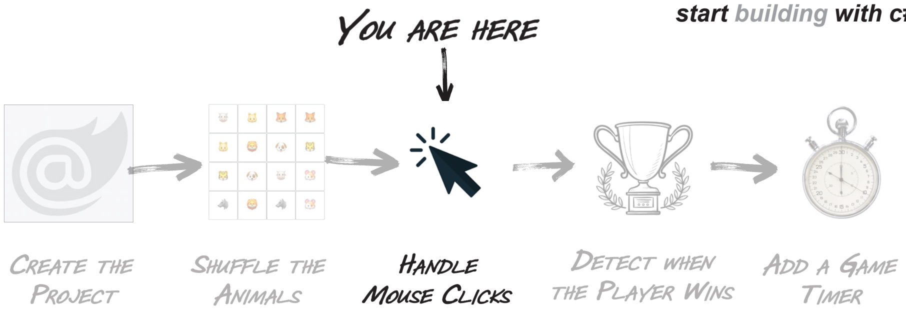
Module name: BlazorMatchGame

Message: Initial check-in of module BlazorMatchGame

Once you create a remote repo on GitHub, you can paste its URL here.

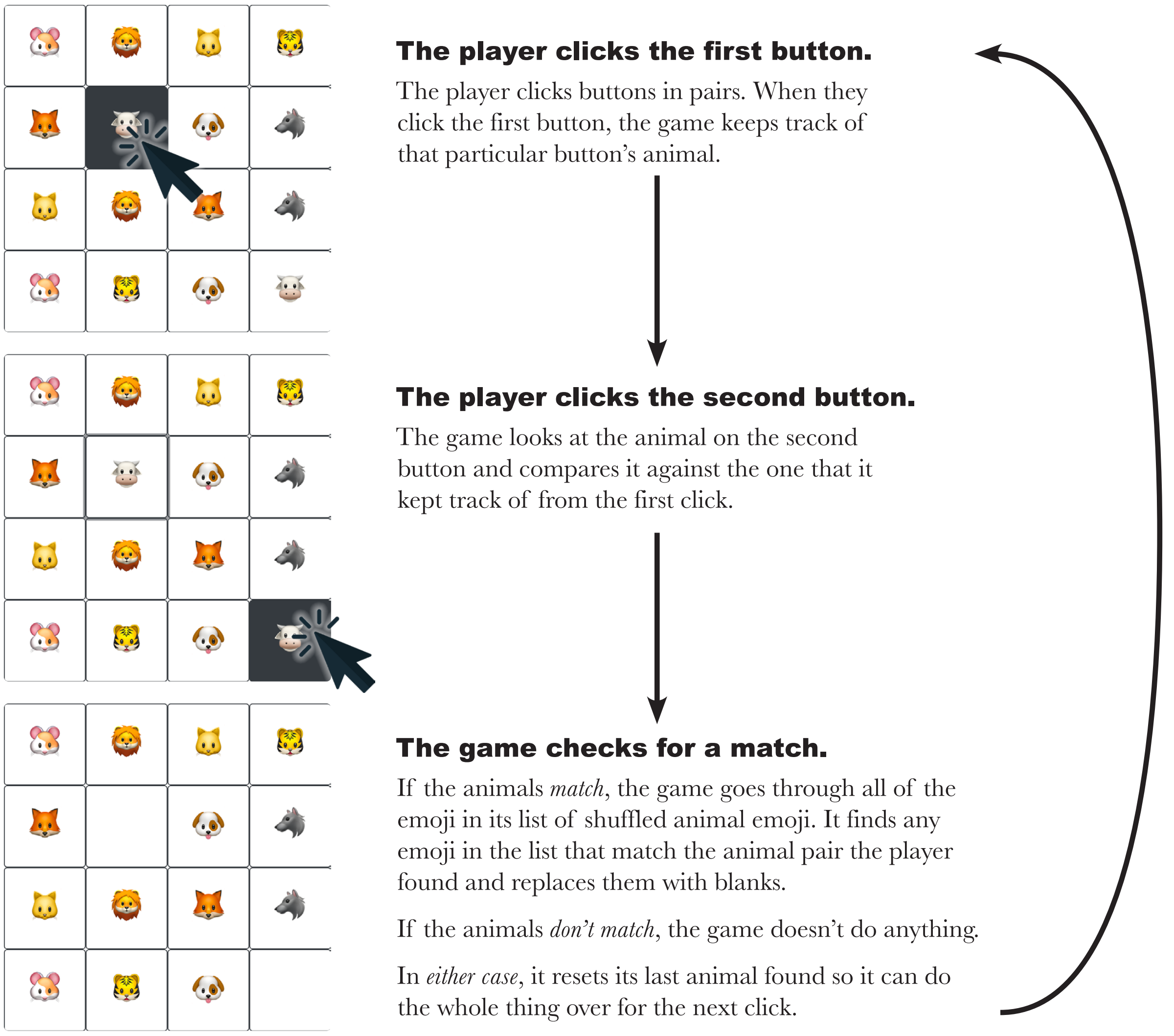
You'll enter your username here.

The Visual Studio for Mac documentation has a complete guide to creating projects on GitHub and publishing them from Visual Studio. It includes step-by-step instructions for creating a remote repo on GitHub and publishing your projects to Git directly from Visual Studio. We think it's a great idea to publish all of your *Head First C#* projects to GitHub—that way you can easily go back to them later. https://docs.microsoft.com/en-us/visualstudio/mac/set-up-git-repository.



Add C# code to handle mouse clicks

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



The player clicks the first button.

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

The player clicks the second button.

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

The game checks for a match.

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

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

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

Add click event handlers to your buttons

When you click a button, it needs to do something. In web pages, a click is an **event**. Web pages have other events, too, like when a page finishes loading, or when an input changes. An **event handler** is C# code that gets executed any time a specific event happens. We'll add an event handler that implements the button functionality.

Don't worry if you're not 100% clear on what all of the C# code does yet! For now just concentrate on making sure your code matches ours exactly.

Here's the code for the event handler

Add this code to the bottom of your Razor page, just above the closing **}** at the bottom:

```
string lastAnimalFound = string.Empty;

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

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

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

This is a LINQ query. There's more on LINQ in Chapter 9.

Hook up your event handler to the buttons

Now you just need to modify your buttons to call the ButtonClick method when clicked:

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

Add this @onclick attribute to the HTML inside your foreach. Be really careful with the parentheses.

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

Your Event Handler Up Close

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

The player clicks the first button.

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

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

The player clicks the second button.

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

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



}
```

The game checks for a match.

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

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

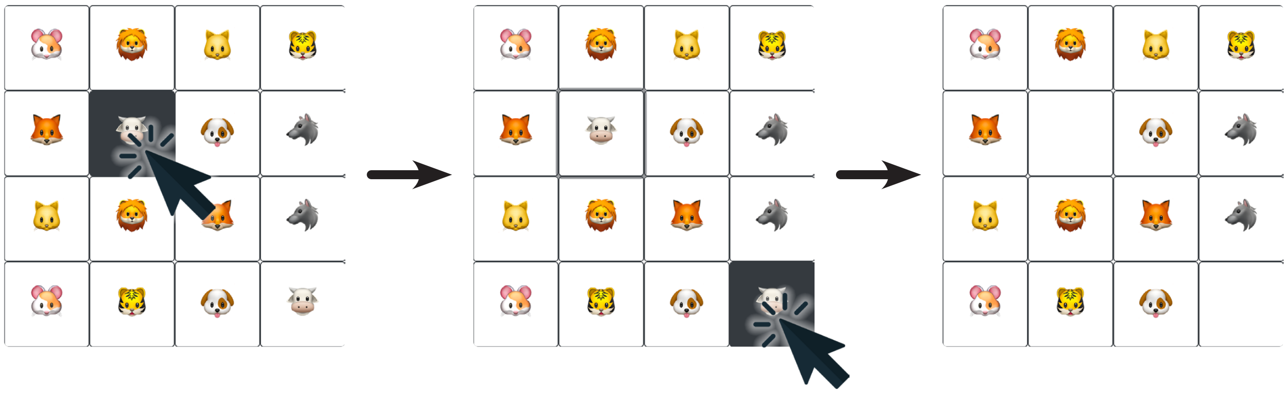
You'll find this statement in the code twice: in the section that's run if the second animal the player clicked matches the first, and in the section that's run if the second animal doesn't match. It blanks out the last animal found to reset the game so the next button click is the first of the pair.

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

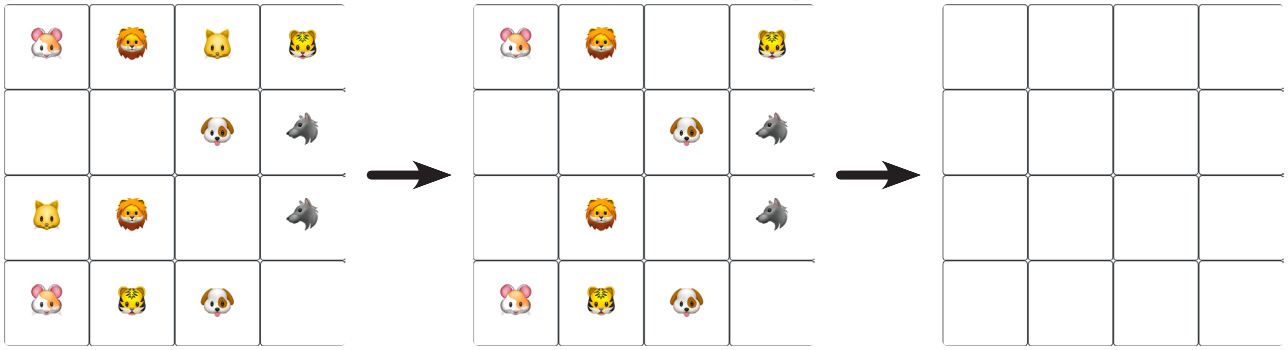
Uh-oh—there's a bug in this code! Can you spot it?
We'll track it down and fix it in the next section.

Test your event handler

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

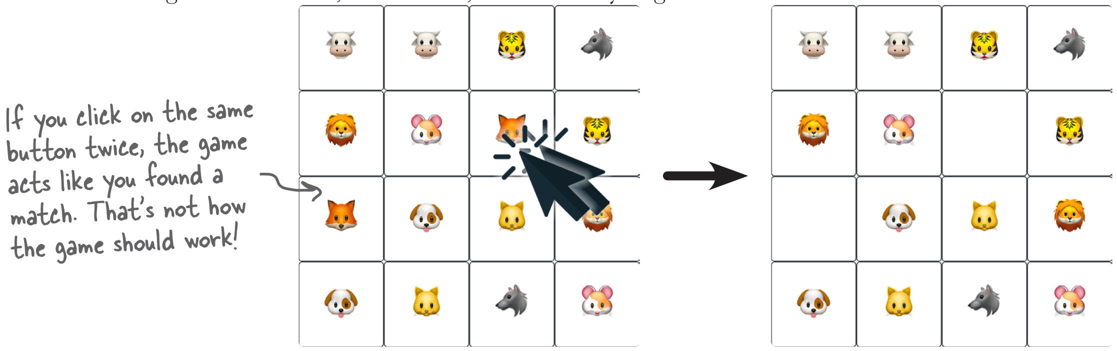


Click on another, then another, then another. You should be able to keep clicking on pairs until all of the buttons are blank. Congratulations, you found all the pairs!



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

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


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

Use the debugger to troubleshoot the problem

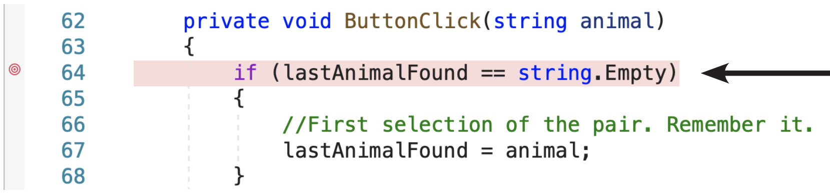
You might have heard the word “bug” before. You might have even said something like this to your friends at some point in the past: “That game is really buggy, it has so many glitches.” Every bug has an explanation—everything in your program happens for a reason—but not every bug is easy to track down.

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

1. **Think about what’s going wrong.**
   The first thing to notice is that your bug is **reproducible**: any time you click on the same button twice, it always acts like you clicked a matching pair.

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

2. **Add a breakpoint to the Click event handler code that you just wrote.**
   Click on the first line of the ButtonClick method and **choose Run >> Toggle Breakpoint** (⌘\\) from the menu. The line will change color and you’ll see a dot in the left margin:

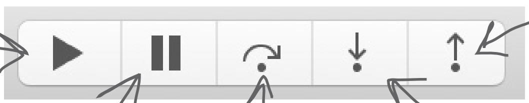


```
        private void ButtonClick(string animal)
        {
            if (lastAnimalFound == string.Empty)
            {
                //First selection of the pair. Remember it.
                lastAnimalFound = animal;
            }
```

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

Anatomy of the Debugger

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



The Continue Execution button starts your app running again.

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

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

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

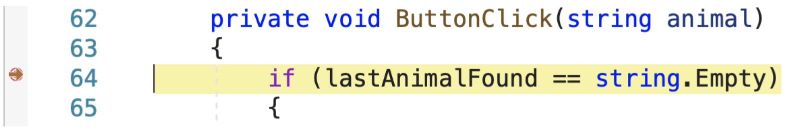
The Step Out button finishes executing the current method and breaks on the line after the one that called it.

Keep debugging your event handler

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

3 **Click on an animal to trigger the breakpoint.**

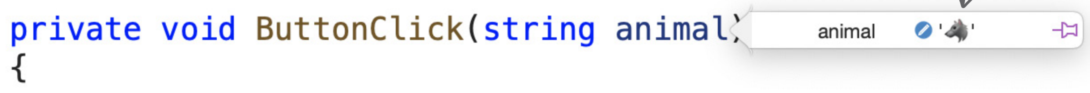
If your app is already running, stop it and close all browser windows. Then **run your app** again and **click any animal button**. Visual Studio should pop into the foreground. The line where you toggled the breakpoint should now be highlighted in a different color:



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

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

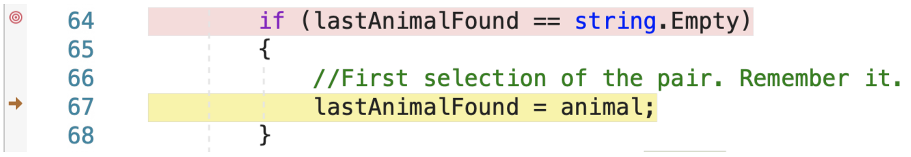
Hover over "animal" to see the emoji that you clicked.



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

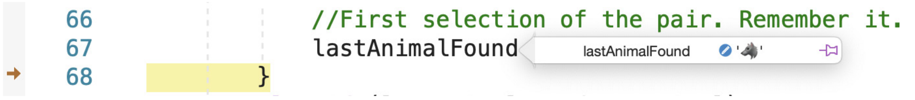
```
animal  '🐺'
```

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



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

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



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

```
lastAnimalFound  '🐺'
```

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

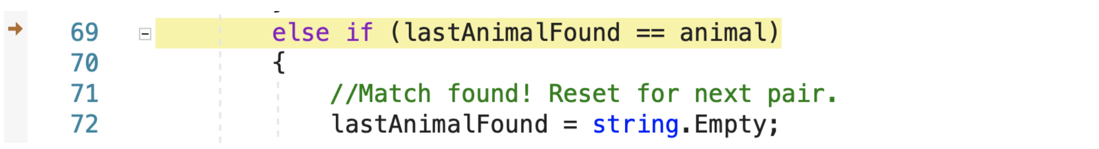
That's how the code keeps track of the first animal that the player clicked.

4 **Continue execution.**

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

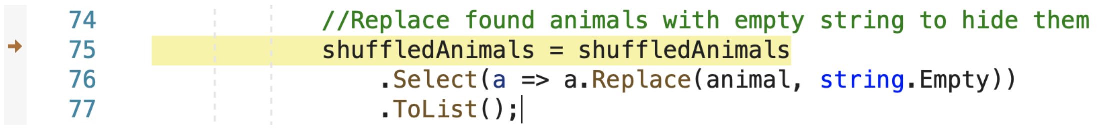
5 Click the matching animal in the pair.

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



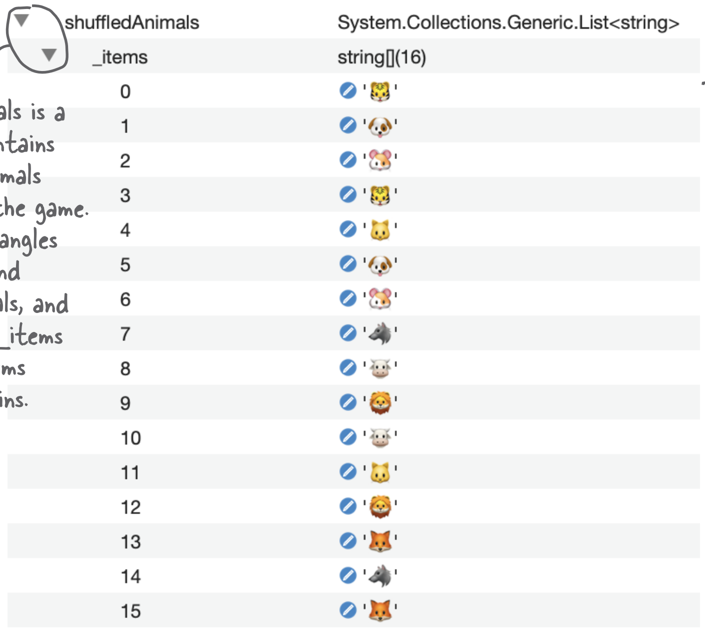
```
            else if (lastAnimalFound == animal)
            {
                //Match found! Reset for next pair.
                lastAnimalFound = string.Empty;
```

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



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

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

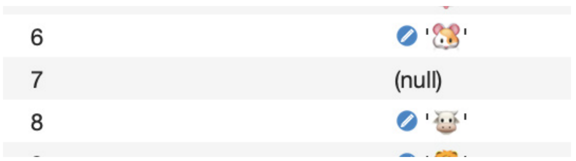


shuffledAnimals	System.Collections.Generic.List<string>
_items	string[](16)
0	'🐯'
1	'🐶'
2	'🐹'
3	'🐯'
4	'🐱'
5	'🐶'
6	'🐹'
7	'🐺'
8	'🐮'
9	'🦁'
10	'🐮'
11	'🐱'
12	'🦁'
13	'🦊'
14	'🐺'
15	'🦊'

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

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

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



6	'🐹'
7	(null)
8	'🐮'

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

Track down the bug that's causing the problem...

Sleuth it out

It's time to put on our Sherlock Holmes caps and sleuth out the problem. We've gathered a lot of evidence. Here's what we know so far:

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

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

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

...and fix the bug!

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

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

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

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

Now each button gets a description as well as an animal, and the event handler uses lastDescription to keep track of it.

Now it makes sure the animals match and the descriptions also match.

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

```
<div class="row">

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

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

Replace the foreach loop with a for loop. We cover loops in Chapter 2.

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

there are no Dumb Questions

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

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

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

Q: What web server is my browser connecting to?

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

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

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

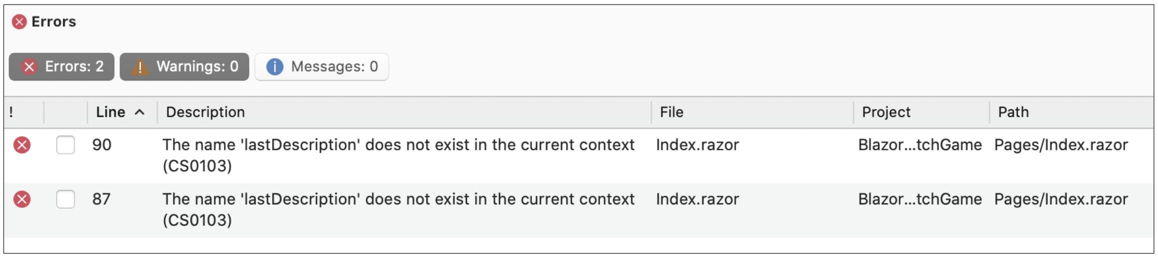
A: Every time you stop and restart your app in Visual Studio, it launches a new instance of the browser because it needs to establish a separate connection for debugging. You can connect other instances of a browser, but you can only debug with the browser that the IDE launched. You can test this yourself: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

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

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

IDE Tip: The Errors Window

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


Errors

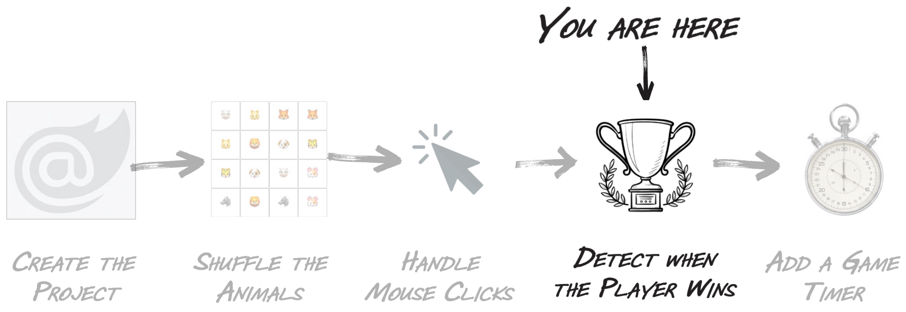
Errors: 2 | Warnings: 0 | Messages: 0

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

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

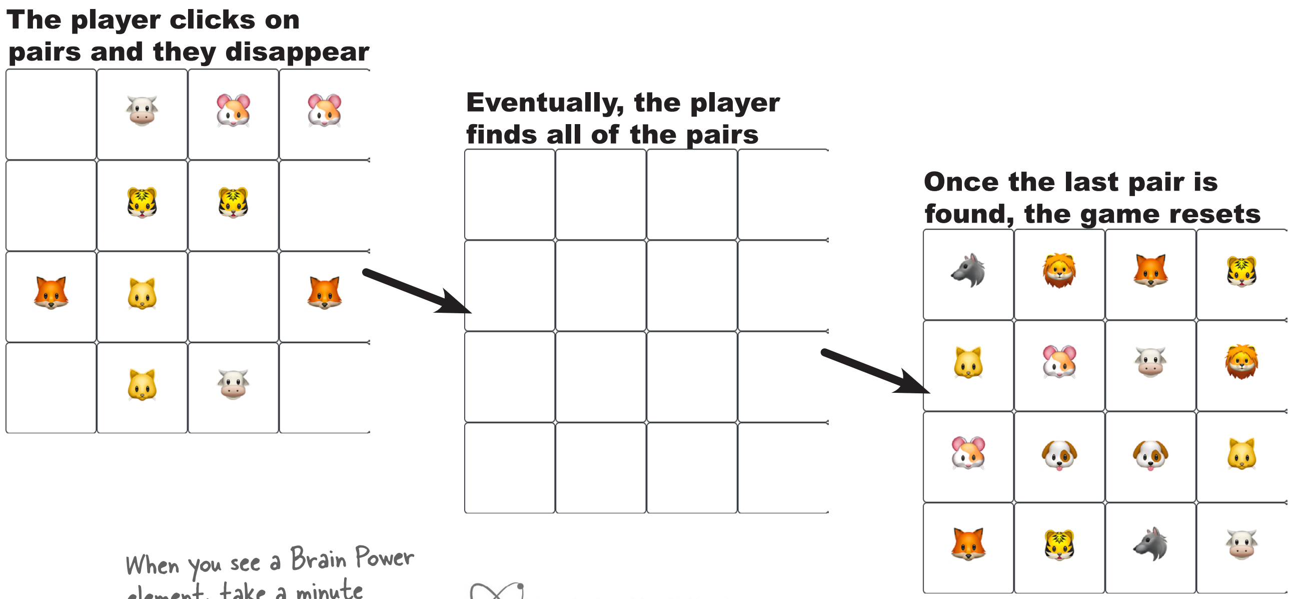
Let's add an error to your code. Go to the first line in your SetUpGame method, then add this on its own line: `Xyz`

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



Add code to reset the game when the player wins

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


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

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

BRAIN POWER

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

Exercise

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

```
int matchesFound = 0;
```

```
matchesFound = 0;
```

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

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

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

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

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

Which of the four blocks of code above goes in this box?

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

```
private void SetUpGame()
{


    Random random = new Random();
    shuffledAnimals = animalEmoji
        .OrderBy(item => random.Next())
        .ToList();
    [                                                            ]
}
```

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

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

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

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

Exercise Solution

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

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

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

This Razor markup uses @matchesFound to make the page display the number of matches found underneath the grid of buttons.

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

This is where the game keeps track of the number of matches the player's found so far.

```
private void SetUpGame()
{


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

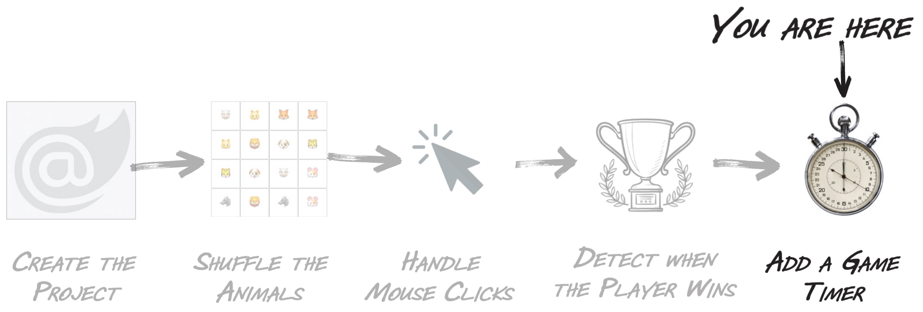
When the game is set up or reset, it resets the number of matches found back to zero.

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

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

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

Every time the player finds a match this block adds 1 to matchesFound. If all 8 matches are found, it resets the game.

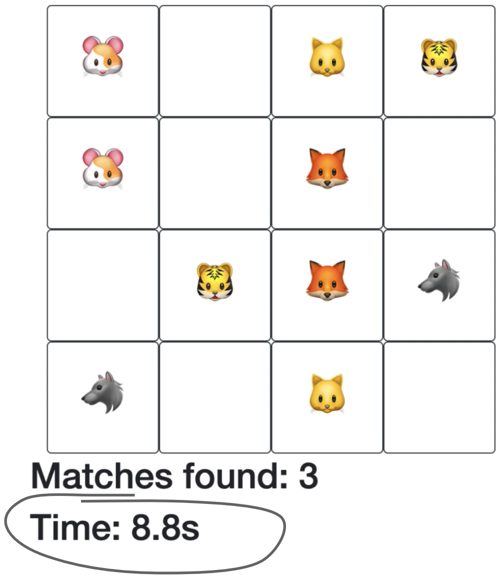
BRAIN POWER

You've reached another checkpoint in your project! Your game might not be finished yet, 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

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



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

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

Add a timer to your game's code

Add this!

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

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

```
@using System.Timers
```

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

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

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

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

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

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

    SetUpGame();
}
```

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

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

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

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

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

    timer.Start();
}
```

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

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

    SetUpGame();
}
```

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

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

```
private void Timer_Tick(Object source, ElapsedEventArgs e)
{
    InvokeAsync(() =>
    {
        tenthsOfSecondsElapsed++;
        timeDisplay = (tenthsOfSecondsElapsed / 10F)
            .ToString("0.0s");
        StateHasChanged();
    });
}
```

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

Clean up the navigation menu

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

Start by expanding the **wwwroot folder** and editing *index.html*. Find the line that starts `<title>` and **modify it** so it looks like this: `<title>`**`Animal Matching Game`**`</title>`

Next, expand the **Shared folder** in the solution and **double-click *NavMenu.razor***. Find this line:

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

and **replace it with this**:

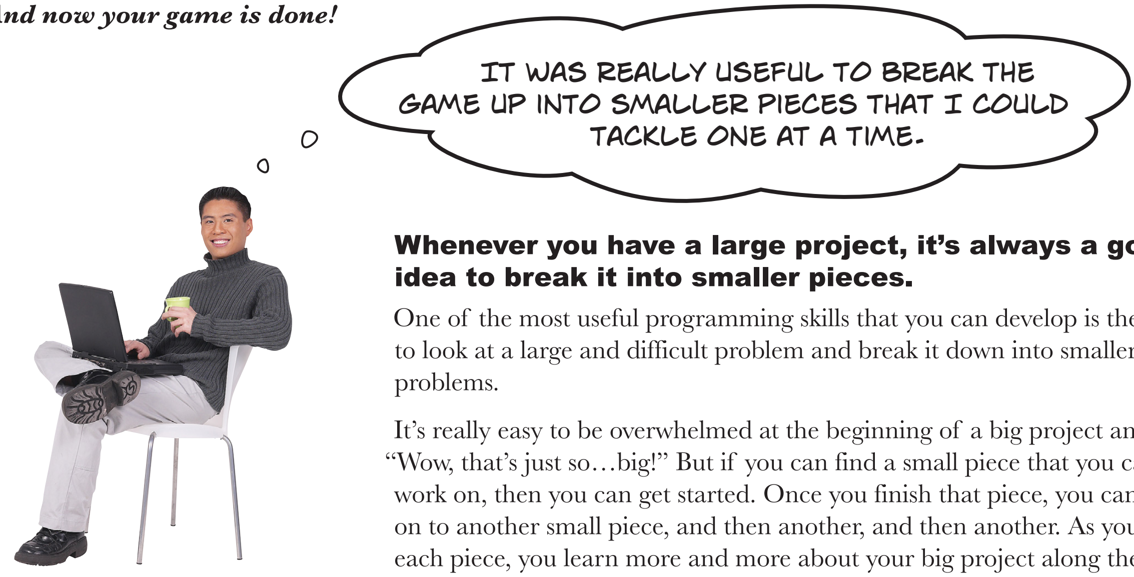
```
<a class="navbar-brand" href="">Animal Matching Game</a>
```

Then **delete these lines**:

~~`<li class="nav-item px-3">`~~
~~`    <NavLink class="nav-link" href="counter">`~~
~~`        <span class="oi oi-plus" aria-hidden="true"></span> Counter`~~
~~`    </NavLink>`~~
~~`</li>`~~
~~`<li class="nav-item px-3">`~~
~~`    <NavLink class="nav-link" href="fetchdata">`~~
~~`        <span class="oi oi-list-rich" aria-hidden="true"></span> Fetch data`~~
~~`    </NavLink>`~~
~~`</li>`~~

Finally, hold down ⌘ (Command) and **click to multiselect these files** in the Solution window: ***Counter.razor*** and ***FetchData.razor*** in the Pages folder, ***SurveyPrompt.razor*** in the Shared folder, and the **entire sample-data** folder inside the wwwroot folder. Once they're all selected, right-click on one of them and **choose Delete** (⌘⌫) from the menu to delete them.

And now your game is done!


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

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

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

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

Even better ifs...

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

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

MINI Sharpen your pencil

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

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

BULLET POINTS

- An **event handler** is a method that your application calls when a specific event like a mouse click, page reload, or timer tick happens.
- The IDE's **Errors window** shows any errors that prevent your code from building.
- **Timers** execute Tick event handler methods over and over again on a specified interval.
- `foreach` is a kind of loop that iterates through a collection of items.
- `for` is a kind of loop that can be used for counting.
- When your program has a **bug**, gather evidence and try to figure out what's causing it.
- Bugs are easier to fix when they're **reproducible**.
- The IDE's **debugger** lets you pause your app on a specific statement to help track down problems.
- Setting a **breakpoint** makes the debugger pause on the statement where the breakpoint is set.
- Visual Studio makes it really easy to use **source control** to back up your code and keep track of all changes that you've made.
- You can commit your code to a **remote Git repository**. We use GitHub for the repository with the source code for all of the projects in this book.

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

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

from Chapter 2 dive into C#

This is the Blazor version of the Windows desktop project in Chapter 2.

The last part of Chapter 2 is a Windows project to experiment with different kinds of controls. We'll use Blazor to build a similar project to experiment with web controls.

Controls drive the mechanics of your user interfaces

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

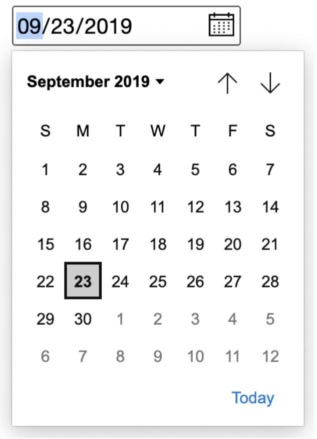
- A **text box** lets a user enter any text they want. But we need a way to make sure they're only entering numbers and not just any text.

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

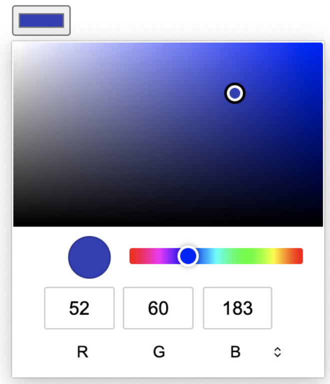
- **Sliders** are used exclusively to choose a number. Phone numbers are just numbers, too, so *technically* you could use a slider to choose a phone number. Do you think that's a good choice?

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.



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



Create a new Blazor WebAssembly App project

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

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

1. **Create a new Blazor WebAssembly App project.**
Either start up Visual Studio 2019 for Mac or choose *File >> New Solution... (⇧⌘N)* from the menu to **bring up the New Project window**. **Click New** to create a new project. Name it **ExperimentWithControlsBlazor**.

2. **Change the title and navigation menu.**
At the end of the animal matching game project, you modified the title and navigation bar text. Do the same for this project. Expand the **wwwroot folder** and edit *Index.html*. Find the line that starts `<title>` and **modify it** so it looks like this: `<title>`**`Experiment with Controls`**`</title>`

Expand the **Shared folder** in the solution and **double-click on *NavMenu.razor***. Find this line:

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

and **replace it with this**:

```
<a class="navbar-brand" href="">Experiment With Controls</a>
```

3. **Remove the extra navigation menu options and their corresponding files.**
This is just like what you did at the end of the animal matching game project. **Double-click on *NavMenu.razor*** and **delete these lines**:

~~`<li class="nav-item px-3">`~~
~~`    <NavLink class="nav-link" href="counter">`~~
~~`        <span class="oi oi-plus" aria-hidden="true"></span> Counter`~~
~~`    </NavLink>`~~
~~`</li>`~~
~~`<li class="nav-item px-3">`~~
~~`    <NavLink class="nav-link" href="fetchdata">`~~
~~`        <span class="oi oi-list-rich" aria-hidden="true"></span> Fetch data`~~
~~`    </NavLink>`~~
~~`</li>`~~

Then hold down ⌘ (Command) and **click to multiselect these files** in the Solution window: ***Counter.razor*** and ***FetchData.razor*** in the Pages folder, ***SurveyPrompt.razor*** in the Shared folder, and the **entire sample-data** folder inside the wwwroot folder. Once they're all selected, right-click on one of them and **choose Delete** (⌘⌫) from the menu to delete them.

Create a page with a slider control

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

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

1 Replace the Index.razor page.

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

```
@page "/"

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

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

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

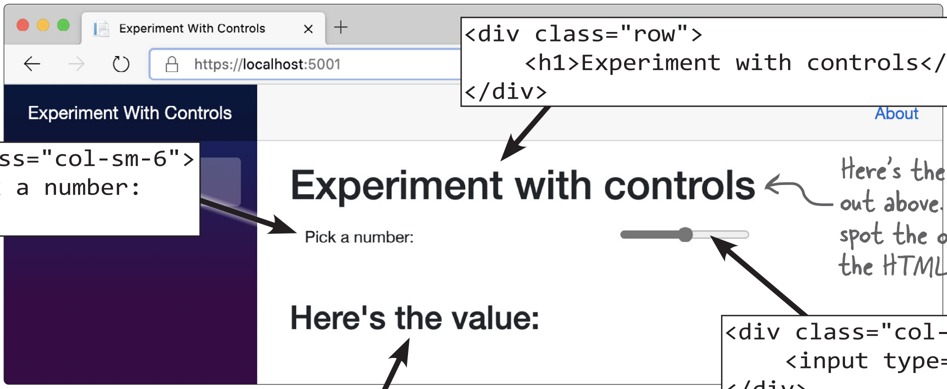
This is an input tag. It has a type attribute that determines what kind of input control appears on the page. When you set the `type` to `range`, it displays a slider:

`<input type="range"/>`

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

2 Run your app.

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



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

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

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

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

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

3 **Add C# code to your page.**

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

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

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

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

The change event handler updates DisplayValue any time it's called with a value.

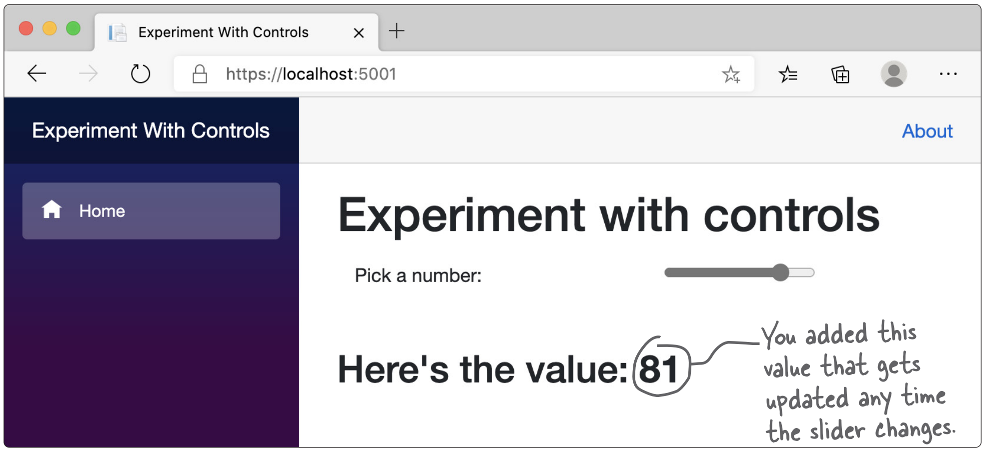
4 **Hook up your range control to the Change event handler you just added.**

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

```
@page "/"

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

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

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



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

Add a text input to your app

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

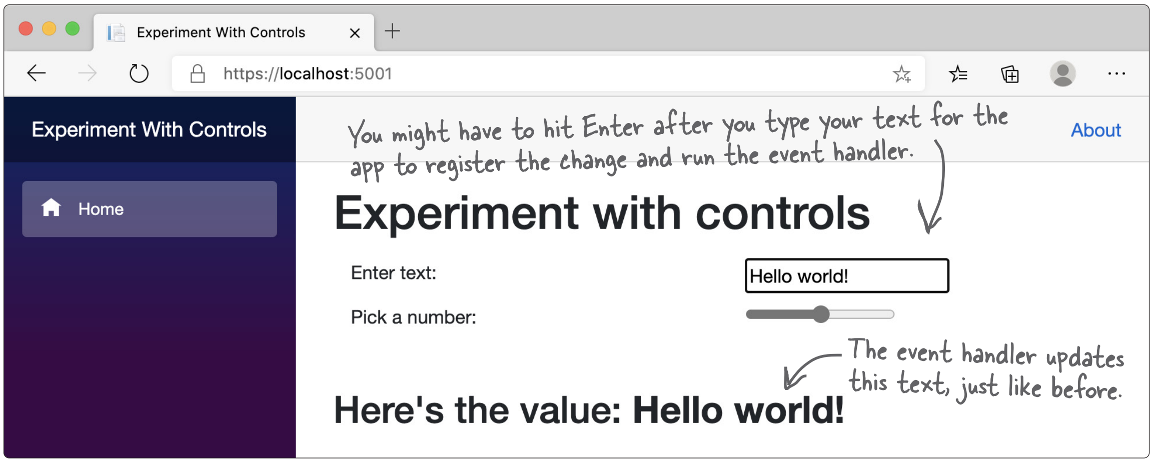
1. **Add a text input control to your page's HTML markup.**
   Add an `<input ... />` tag that's almost identical to the one you added for the slider. The only difference is that you'll set the `type` attribute to `"text"` instead of `"range"`. Here's the HTML markup:

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

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

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

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



Here's the value: Hello world!

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

The event handler updates this text, just like before.

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

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

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

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

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

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

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

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

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

Exercise

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

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

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

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

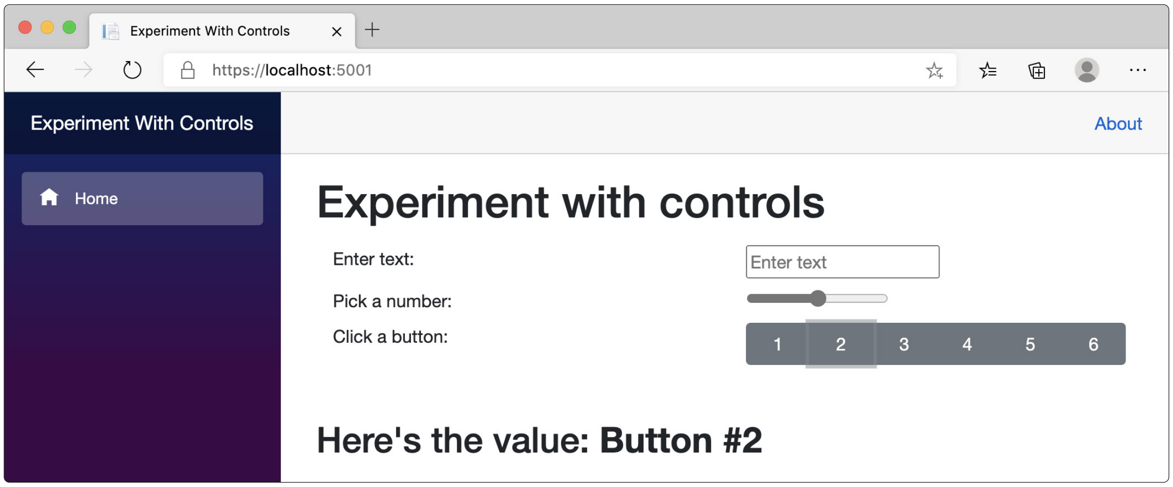
Here's the line of code that makes the Razor markup add six buttons to the page. It's a `for` loop, and it works just like the other `for` loops you learned about in Chapter 2:

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

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

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

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



Experiment with controls

Enter text: Enter text

Pick a number:

Click a button:

Here's the value: Button #2

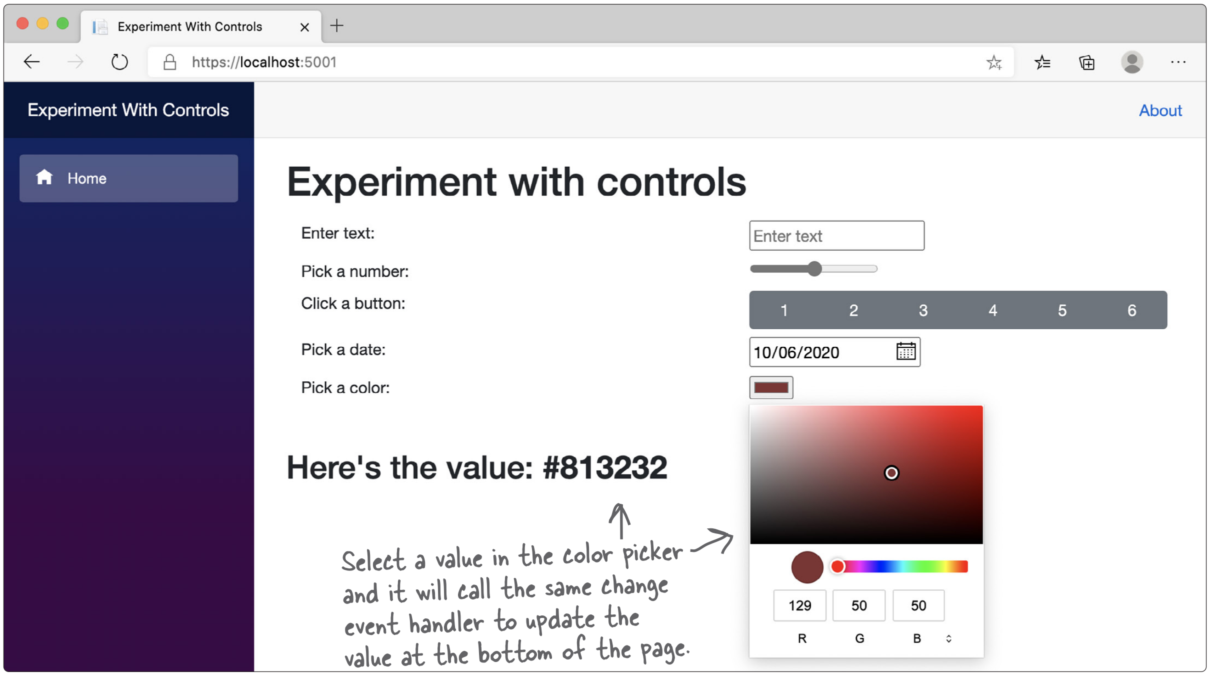
Add color and date pickers to your app

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

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

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

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



Experiment with controls

Enter text: Enter text

Pick a number:

Click a button: 1 2 3 4 5 6

Pick a date: 10/06/2020

Pick a color:

Here's the value: #813232

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

That's the end of the project—great job! You can pick up Chapter 2 at the very end, where there's a person in a chair thinking this:

THERE ARE SO MANY DIFFERENT WAYS FOR USERS TO CHOOSE NUMBERS!

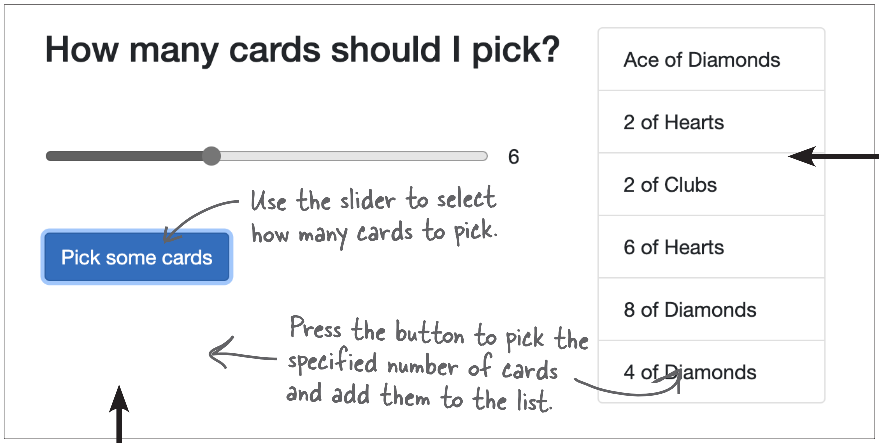
from Chapter 3 objects...get oriented!

This is the Blazor version of the Windows desktop project in Chapter 3.

Partway through Chapter 3, there's a project where you build a Windows version of the card picker app. We'll use Blazor to build a web-based version of the same app.

Up next: build a Blazor version of your card picking app

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



How many cards should I pick?
6
Pick some cards
Ace of Diamonds
2 of Hearts
2 of Clubs
6 of Hearts
8 of Diamonds
4 of Diamonds
Use the slider to select how many cards to pick.
Press the button to pick the specified number of cards and add them to the list.

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

This button's event handler will call a method in your class that returns a list of cards, then it will add each card to an array.

Reuse this!

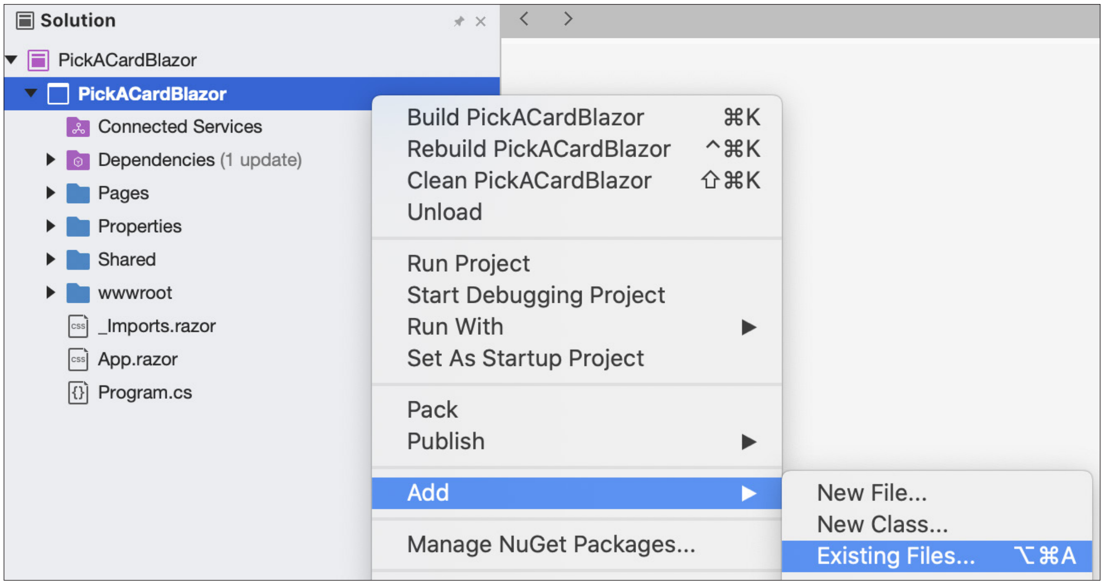
Reuse your CardPicker class in a new Blazor app

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

① **Create a new Blazor WebAssembly App project called PickACardBlazor.**
You'll follow exactly the same steps you used to create your animal matching game in Chapter 1:

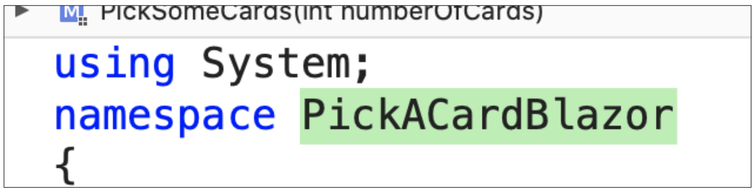
- ★ Open Visual Studio and create a new project.
- ★ Select **Blazor WebAssembly App**, just like you did with your previous Blazor apps.
- ★ Name your new app **PickACardBlazor**. Visual Studio will create the project.

② **Add the CardPicker class that you created for your Console App project.**
Right-click on the project name and choose **Add >> Existing Files...** from the menu:



Navigate to the folder with your console app and **click on *CardPicker.cs*** to add it to your project. Visual Studio will ask you if you want to copy, move, or link to the file. Tell Visual Studio to **copy the file**. Your project should now have a copy of the *CardPicker.cs* file from your console app.

③ **Change the namespace for the CardPicker class.**
Double-click on *CardPicker.cs* in the Solution window. It still has the namespace from the console app. **Change the namespace** to match your project name:



```
PickSomeCards(int numberOfCards)
using System;
namespace PickACardBlazor
{
```

You're changing the namespace in the CardPicker.cs file to match the namespace that Visual Studio used when it created the files in your new project so you can use your CardPicker class in your new project's code. For example, if you open Program.cs you'll see that it's in the same namespace.

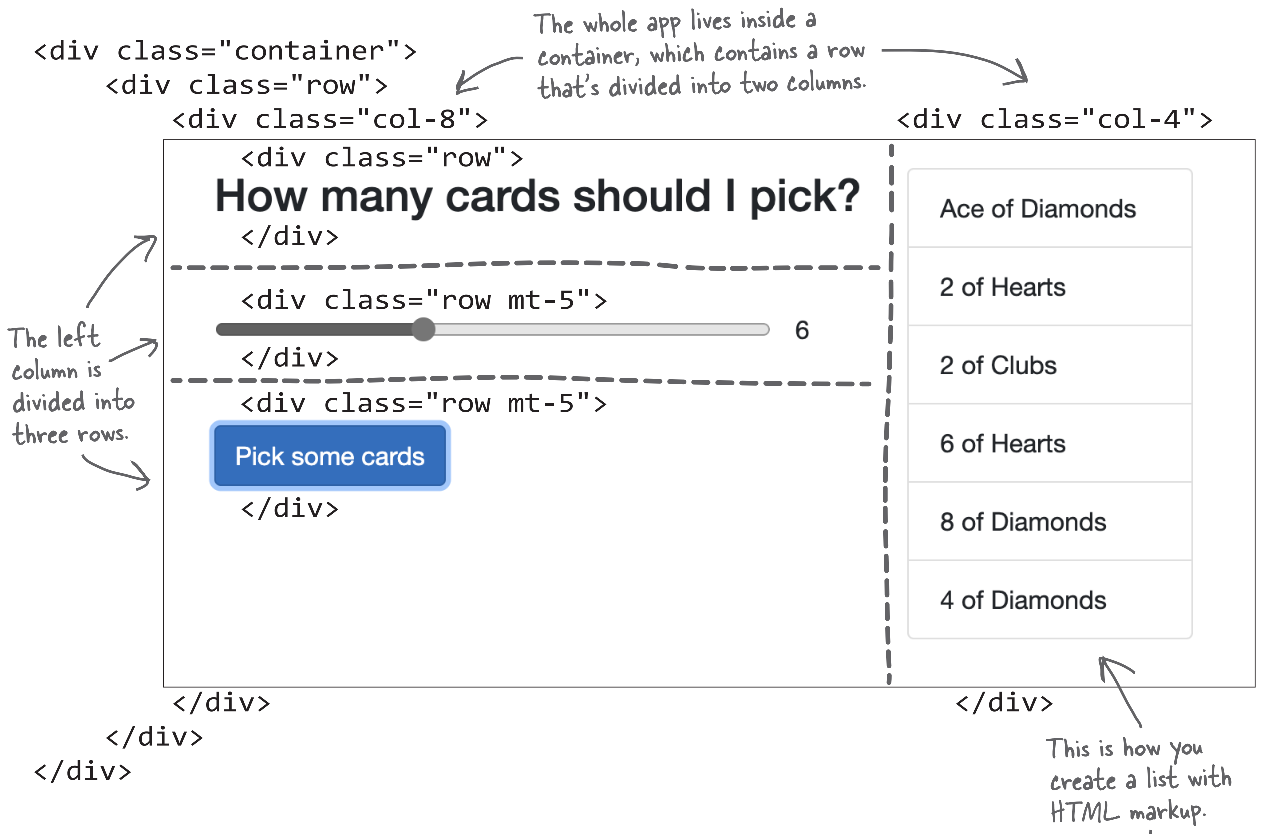
Now your CardPicker class should be in the PickACardBlazor namespace:

```
namespace PickACardBlazor
{
    class CardPicker
    {
```

Congratulations, you've reused your CardPicker class! You should see the class in the Solution window, and you'll be able to use it in the code for your Blazor app.

The page is laid out with rows and columns

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



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

The whole app lives inside a container, which contains a row that's divided into two columns.
The left column is divided into three rows.
This is how you create a list with HTML markup.

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

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

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

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

The slider uses data binding to update a variable

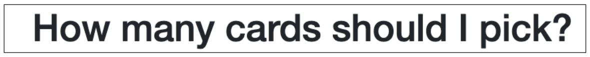
The code at the bottom of the page will start with a variable called `numberOfCards`:

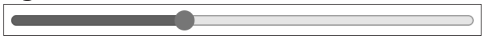
```
@code {
    int numberOfCards = 5;
```

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

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

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


How many cards should I pick?



6

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

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

Exercise

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

Step 1: Finish the HTML markup

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

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

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

Step 2: Finish the code

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

- Add a string array field called `pickedCards`: `string[] pickedCards = new string[0];`
- Add the UpdateCards event handler method called by the button. It calls CardPicker.PickSomeCards and assigns the result to the `pickedCards` field.

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

```
@page "/"

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

@code {
    int numberOfCards = 5;

    string[] pickedCards = new string[0];

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

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

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

numberOfCards and pickedCards are special kinds of variables called fields. You'll learn about them later in Chapter 3.

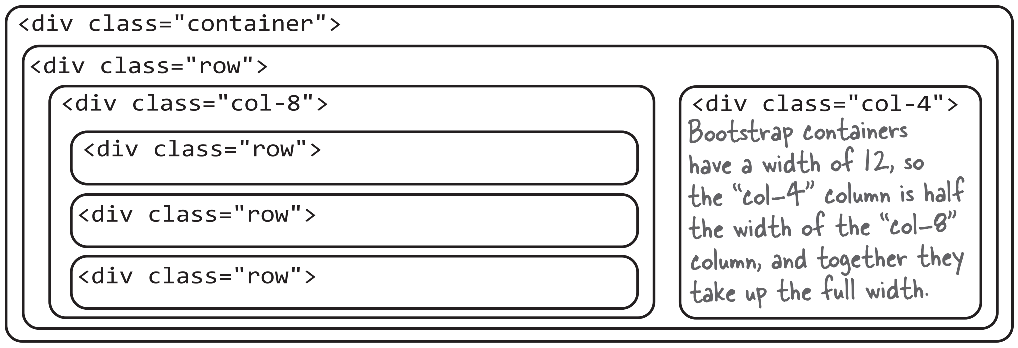
The button's Click event handler method calls the PickSomeCards method in the CardPicker class that you wrote earlier in the chapter.

Behind the Scenes

Your Blazor web apps use Bootstrap for page layout.

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

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



```
<div class="container">
<div class="row">
<div class="col-8">
<div class="row">
<div class="row">
<div class="row">
<div class="col-4">
```

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

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

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

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

BULLET POINTS

- Classes have methods that contain statements that perform actions. Well-designed classes have sensible method names.
- Some methods have a **return type**. You set a method's return type in its declaration. A method with a declaration that starts with the `int` keyword returns an int value. Here's an example of a statement that returns an int value: `return 37;`
- When a method has a return type, it **must** have a `return` statement that returns a value that matches a return type. A method declaration with a string return type has a `return` statement that returns a string.
- As soon as a `return` statement in a method executes, your program jumps back to the statement that called the method.
- Not all methods have a return type. A method with a declaration that starts `public void` doesn't return anything at all. You can still use a `return` statement to exit a void method: `if (finishedEarly) { return; }`
- Developers often want to **reuse** the same code in multiple programs. Classes can help you make your code more reusable.

That's the end of the project—nice work! You can go back to Chapter 3 and resume at the section with this heading: *Ana's prototypes look great...*

from Chapter 4 types and references

This is the Blazor version of the Windows desktop project in Chapter 4.

At the end of Chapter 4 there's a Windows project. We'll build a Blazor version of it.

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

Sloppy Joe has a pile of meat, a whole lotta bread, and more condiments than you can shake a stick at. What he doesn't have is a menu! Can you build a program that makes a new *random* menu for him every day? You definitely can... with a **new Blazor WebAssembly App project**, some arrays, and a couple of useful new techniques.

Do this!

MenuItem
Randomizer Proteins Condiments Breads Description Price
Generate

1. **Add a new MenuItem class to your project and add its fields.**
Have a look at the class diagram. It has six fields: an instance of Random, three arrays to hold the various sandwich parts, and string fields to hold the description and price. The array fields use **collection initializers**, which let you define the items in an array by putting them inside curly braces.

```
class MenuItem
{
    public Random Randomizer = new Random();
    public string[] Proteins = { "Roast beef", "Salami", "Turkey",
            "Ham", "Pastrami", "Tofu" };
    public string[] Condiments = { "yellow mustard", "brown mustard",
           "honey mustard", "mayo", "relish", "french dressing" };
    public string[] Breads = { "rye", "white", "wheat", "pumpernickel", "a roll" };

    public string Description = "";
    public string Price;
}
```

2. **Add the Generate method to the MenuItem class.**
This method uses the same Random.Next method you've seen many times to pick random items from the arrays in the Proteins, Condiments, and Breads fields and concatenate them together into a string.

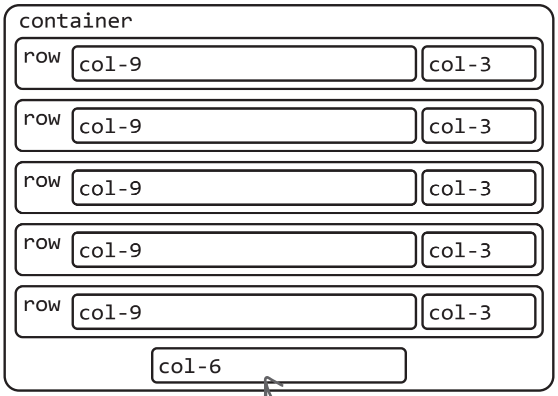
```
    public void Generate()
    {
        string randomProtein = Proteins[Randomizer.Next(Proteins.Length)];
        string randomCondiment = Condiments[Randomizer.Next(Condiments.Length)];
        string randomBread = Breads[Randomizer.Next(Breads.Length)];
        Description = randomProtein + " with " + randomCondiment + " on " + randomBread;

        decimal bucks = Randomizer.Next(2, 5);
        decimal cents = Randomizer.Next(1, 98);
        decimal price = bucks + (cents * .01M);
        Price = price.ToString("c");
    }
```

The Generate method makes a random price between 2.01 and 5.97 by converting two random ints to decimals. Have a close look at the last line—it returns `price.ToString("c")`. The parameter to the ToString method is a format. In this case, the `"c"` format tells ToString to format the value with the local currency: if you're in the United States you'll see a $; in the UK you'll get a £, in the EU you'll see €, etc.

3 **Add the page layout to your Index.razor file.**

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

```
@page "/"

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

@code {
    MenuItem[] menuItems = new MenuItem[5];
    string guacamolePrice;
}
```



The bottom row has one column that's half the width of the container. This row has the justify-content-center class, which causes the bottom row to be centered on the page.

Exercise

Add the `@code` section to the bottom of your *Index.razor* file. It adds five MenuItem objects to the `menuItems` field and sets the `guacamolePrice` field.

Step 1: Add an OnInitialized method

You used an OnInitialized method to shuffle the animals in your animal matching game. Add this line of code:

```
protected override void OnInitialized()
```

Step 2: Replace the OnInitialized body with code to create the MenuItem objects

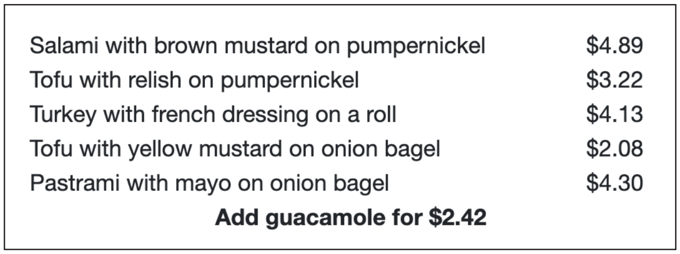
The IDE will automatically fill in the body (`base.OnInitialized();`) like it did when you created your animal matching game. Delete that statement. Replace it with code that sets the `menuItems` and `guacamolePrice` fields.

- Add a `for` loop that adds five MenuItem objects to the `menuItems` array field and calls their Generate methods.
- The last two items on the menu should be bagel sandwiches, so set their Breads field to a new string array:
```
new string[] { "plain bagel", "onion bagel",
               "pumpernickel bagel", "everything bagel" }
```
- Create a new MenuItem instance, call its Generate method, and use its Price field to set `guacamolePrice`.

Exercise Solution

Here's the entire code for the *Index.razor* file. Everything up to `string guacamolePrice;` matches the code that we gave you—your job was to fill in the rest of the `@code` block.

```
@page "/"

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



Salami with brown mustard on pumpernickel $4.89
Tofu with relish on pumpernickel $3.22
Turkey with french dressing on a roll $4.13
Tofu with yellow mustard on onion bagel $2.08
Pastrami with mayo on onion bagel $4.30
Add guacamole for $2.42

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

The page uses these two fields for data binding. The menuItems field is used to generate the five rows, while guacamolePrice has the price for the guacamole line at the bottom of the page.

```
    protected override void OnInitialized()
    {
        for (int i = 0; i < 5; i++)
        {
            menuItems[i] = new MenuItem();
```

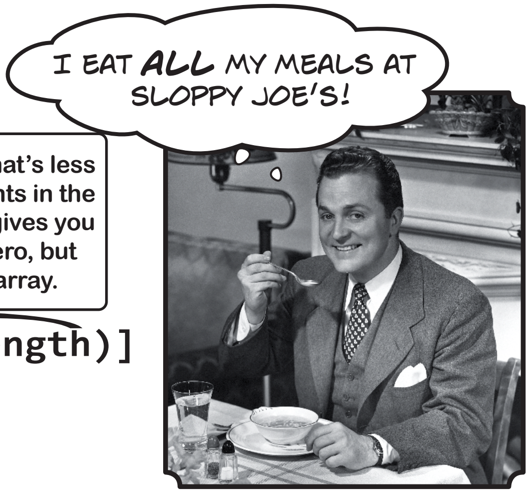
We assigned directly to the array element. You could use a separate variable to create the new MenuItem and then assign it to the array element at the end of the for loop.

```
            if (i >= 3)
            {
                menuItems[i].Breads = new string[] {
                    "plain bagel",
                    "onion bagel",
                    "pumpernickel bagel",
                    "everything bagel"
                };
            }
            menuItems[i].Generate();
```

Make sure you call the Generate method; otherwise the MenuItem's fields will be empty and your page will be mostly blank.

```
        }

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

How it works...



The Randomizer.Next(7) method gets a random int that's less than 7. Breads.Length returns the number of elements in the Breads array. So Randomizer.Next(Breads.Length) gives you a random number that's greater than or equal to zero, but less than the number of elements in the Breads array.

```
Breads[Randomizer.Next(Breads.Length)]
```

Breads is an array of strings. It's got five elements, numbered from 0 to 4. So Breads[0] equals "rye", and Breads[3] equals "a roll".

If your computer is fast enough, your program may not run into this problem. If you run it on a much slower computer, you'll see it.

4 **Run your program and behold the new randomly generated menu.**

Uh…something's wrong. The prices on the menu are all the same, and the menu items are weird—the first three are the same, so are the next two, and they all seem to have the same protein. What's going on?

```
Salami with brown mustard on pumpernickel          $4.89
Salami with brown mustard on pumpernickel          $4.89
Salami with brown mustard on pumpernickel          $4.89
Salami with brown mustard on everything bagel      $4.89
Salami with brown mustard on everything bagel      $4.89
              Add guacamole for $2.42
```

Why aren't the menu items and prices getting randomized?

It turns out that the .NET Random class is actually a **pseudo-random number** generator, which means that it uses a mathematical formula to generate a sequence of numbers that can pass certain statistical tests for randomness. That makes them good enough to use in any app we'll build (but don't use it as part of a security system that depends on truly random numbers!). That's why the method is called Next—you're getting the next number in the sequence. The formula starts with a "seed value"—it uses that value to find the next one in the sequence. When you create a new instance of Random, it uses the system clock to "seed" the formula, but you can provide your own seed. Try calling **`new Random(12345).Next();`** a bunch of times. You're telling it to create a new instance of Random with the same seed value (12345), so the Next method will give you the same "random" number each time.

```
Salami with brown mustard on pumpernickel          $2.54
Roast beef with mayo on a roll                     $2.59
Salami with honey mustard on a roll                $3.81
Salami with french dressing on plain bagel         $4.52
Turkey with yellow mustard on everything bagel     $2.67
              Add guacamole for $2.76
```

When you see a bunch of different instances of Random give you the same value, it's because they were all seeded close enough that the system clock didn't change time, which means they all have the same seed value. So how do we fix this? Use a single instance of Random by making the Randomizer field static so all MenuItems share a single Random instance:

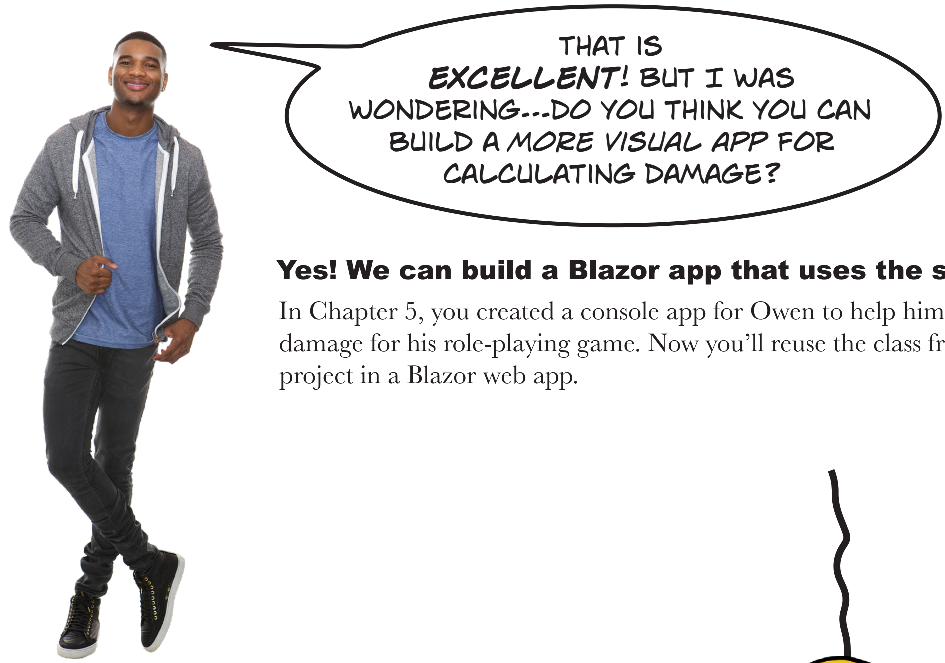
```
public static Random Randomizer = new Random();
```

Run your program again—now the menu will be randomized.

That's the end of the project! Resume at the Bullet Points at the very end of Chapter 4.

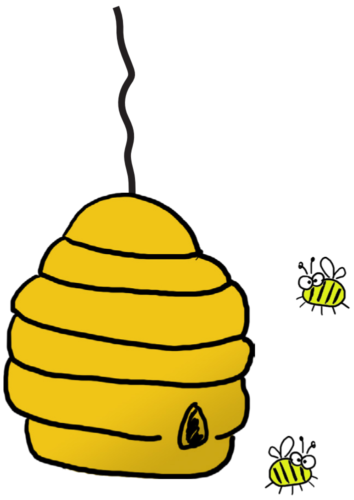
Go to our GitHub page for Chapters 5 and 6

You've probably noticed that the projects in Chapters 5 and 6 are longer and more involved than the ones in the previous chapters. We want to give you the best learning experience possible—but doing that takes more pages than we can fit in this appendix! That's why we've made Chapters 5 and 6 of the Visual Studio for Mac Learner's Guide available as a **PDF download from our GitHub page**: https://github.com/head-first-csharp/fourth-edition.


THAT IS *EXCELLENT!* BUT I WAS WONDERING...DO YOU THINK YOU CAN BUILD A MORE VISUAL APP FOR CALCULATING DAMAGE?

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

In Chapter 5, you created a console app for Owen to help him calculate damage for his role-playing game. Now you'll reuse the class from that project in a Blazor web app.



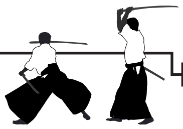
Build a Beehive Management System

The project for Chapter 6 is a serious business application. ***The queen bee needs your help!*** Her hive is out of control, and she needs a program to help manage her honey production business. She's got a beehive full of workers, and a whole bunch of jobs that need to be done around the hive, but somehow she's lost control of which bee is doing what and whether or not she's got the beepower to do the jobs that need to be done. It's up to you to build a **Beehive Management System** to help her keep track of her workers.

The rest of the Windows desktop and Blazor projects after Chapter 6 are all PDF downloads. Look for them on our GitHub page!

ii appendix ii. Code Kata

A learning guide for advanced and/or impatient readers



Code kata

Do you already have experience with another programming language? If so, you might find a **code kata** approach to learning to be an effective, efficient, and satisfying alternative to reading this book cover to cover.

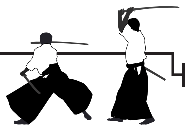
"Kata" is a Japanese word that means "form" or "way," and it's used in many martial arts to describe a training method that involves practicing a series of movements or techniques over and over again. Many developers apply this idea to practice their programming skills by writing specific programs, often more than once. In this book, more experienced developers looking to pick up C# can ***use a code kata approach to take an alternative path through the chapters***. Here's how this will work:

- When you start a new chapter, **flip or scroll through it** until you find the first code kata element (see the following pages for instructions). Read through the nearby Bullet Points element to see what was covered.
- The code kata element will have **instructions for a specific exercise** to do—usually a coding project. Try doing the project, going back to previous sections (especially Bullet Points sections) for extra guidance.
- If you **get stuck** doing the exercise, then you've run into an area where C# is very different from the languages you already know. Go back to the previous code kata element (or the beginning of the chapter, if it's the first one) and start reading the book linearly until you get past the code kata where you got stuck.
- In the **Unity Labs,** you'll get practice writing C# code by doing 3D game development with Unity. These labs are *not required* for the code kata path but strongly recommended. They're also very satisfying, and a really fun way to hone your newly acquired C# skills.

Kata—in martial arts or in code—are meant to be repeated. So if you really want to get C# into your brain, when you finish a chapter go back through it, find the code kata sections, and do them again.

There are no code kata for Chapter 1. It makes sense even for an advanced developer to read this chapter in its entirety and do all of the projects and exercises, even the pencil-and-paper ones. There are foundational ideas in this chapter that the rest of the book will build on. Additionally, if you have experience with another programming language and plan to take the code kata path through this book, **watch this video by Patricia Aas** on learning C# as a second (or 15th) language: https://bit.ly/cs_second_language. This is required for the code kata path.

Do you have a lot of experience with another programming language, and are you looking to learn C# for fun or work? Look for the code kata sections, starting in Chapter 2, which lay out an alternate learning path through the book that's perfect for more advanced (or impatient!) developers. Read this to see if that path will work for you.



Chapter 2: Dive into C#

Chapter 2 is about getting our readers familiar with some basic C# concepts: how code is structured into namespaces, classes, methods, and statements, and some basic syntax. It ends with a project to build a simple UI that takes input: for Windows readers, it's a WPF desktop app; for macOS readers it's a Blazor web app (see the *Visual Studio for Mac Learner's Guide* appendix). The idea behind including projects like these in most of the chapters is to help readers see different approaches to solving similar problems, which can really help when learning a new language.

Get started by skimming through the whole chapter and reading all of the Bullet Points sections. Read through all the code samples. Does everything look pretty familiar? If so, then you're ready to get started on some kata.

Kata #1: Find the Do this! element in the section with the heading **Generate a new method to work with variables**. That's your starting point. Add the code in that section and the following **Add code that uses operators to your method** section to the Console App program you created in Chapter 1. Use the Visual Studio debugger to step through the code—the following section shows you how to do that.

Kata #2: The next few sections have examples of `if` statements and loops to add to your app and debug.

Is everything making sense? If so, you're ready to tackle the WPF project at the end of the chapter or the Blazor project in the *Visual Studio for Mac Learner's Guide* appendix. If you feel comfortable with Visual Studio and can find your way around the IDE and create, run, and debug a .NET Core console app, you're ready for…

Chapter 3: Objects

This chapter is all about introducing the basics of classes, objects, and instances. In the first half, we work with static methods or fields (that means they belong to the type itself, and not to a specific instance); in the second half, we'll create instances of objects. If you do—and understand—these kata, it's safe to move on to the next chapter.

Kata #1: The first project in this chapter is a simple program that generates random playing cards. Find the first Do this! element (next to the **Create your PickRandomCards console app** heading). That's your starting point. You'll create a class and use it in a simple program.

Kata #2: Work through the following sections to finish the CardPicker class. You'll reuse the same class in a WPF desktop app or Blazor web app.

Kata #3: Flip or scroll forward to the "Sharpen your pencil" exercise where you create Clown objects. Type in the code, add comments for the answers, and then step through it.

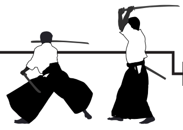
Kata #4: Near the end of the chapter, find the heading **Build a class to work with some guys**. Immediately after that is a class called Guy followed by an exercise. Do that exercise to build a simple app.

Kata #5: Do the exercise immediately after that, where you reuse the Guy class in a simple betting game. Find at least one way to improve on the game: let the player choose different odds and bets, have multiple players, etc.

While you work on these programs, keep an eye out for the places in the code where you might ask these questions—they're answered in later chapters, but noticing them now will help you pick up C# more quickly:

- Doesn't C# have some kind of `switch` statement?
- Don't some people consider multiple `return` statements in a method or function bad practice?
- Why are we using `return` to break out of a loop? Does C# have a better way to break out of a loop without returning from the method?

Chapter 4: Types and references



This chapter is about types and references in C#. Read the first few sections of the chapter to make sure you know the different types—there are some quirks to do with floating-point numbers you should know about. Then glance through the various diagrams to make sure you understand what's going on with references, because you'll see similar ones throughout the book. Did you get all that? OK—go on to these three kata. If you have no trouble with them, it's safe to move on to the next chapter.

Kata #1: The first project in this chapter is a tool to help a role-playing game master calculate ability scores. It starts with the **Let's help Owen experiment with ability scores** heading. You should be able to find and fix both the syntax error and the bug in the program without looking at the answer.

Kata #2: Do the project in the exercise that starts with: Create a program with an Elephant class. Once you have the exercise solved, go through all of the material until you're done with everything under the **Objects use references to talk to each other** heading.

Kata #3: Do the project under the heading **Welcome to Sloppy Joe's Budget House o' Discount Sandwiches**.

Here are some questions that you might think of—they're addressed later in the book:

- Does C# differentiate between value types like double or int and reference types like Elephant, MenuItem, or Random?
- Do we have any control over when the CLR garbage-collects objects that are no longer referenced?
- Does the IDE have tools to track and differentiate specific instances?

Thinking about these questions now will help you pick up C# more quickly.

Chapter 5: Encapsulation

This chapter is about encapsulation, which in C# means restricting access to certain class members to keep other objects from using them in ways that aren't intended. The first project in the chapter features the kind of bug that encapsulation helps to prevent. If you can do the last kata—it involves fixing the bug you create in the first kata—without relying on the solution, it's safe to move on to the next chapter.

Kata #1: The first project in this chapter is a tool to help our role-playing game master roll for damage. It starts at the beginning of the chapter—follow it up through the **Sleuth it out** box. Make sure you understand exactly why the program is broken. Take a quick look at the exercise at the end of the chapter. If you can do it without looking at the answer, then you're ***already*** ready to move on to the next chapter.

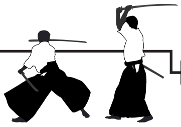
Kata #2: Do the project in the exercise that starts with: Let's get a little practice using the `private` keyword by **creating a small Hi-Lo game**. Make sure you understand how it uses the `const`, `public`, and `private` keywords. Make sure to do the bonus problem.

Kata #3: Do the small project in the "Watch it!" element that starts: **Encapsulation is not the same as security. Private fields are not secure.**

Read the following sections: **Properties make encapsulation easier**, **Auto-implemented properties simplify your code**, and **Use a private setter to create a read-only property.**

Kata #4: Do the last exercise in the chapter, where you use encapsulation to fix the SwordDamage class.

Chapter 6: Inheritance



This chapter is about inheritance, which in C# means creating classes that reuse, extend, and modify behavior in other classes. If you've been following the code kata path, it's likely that you already know an object-oriented language with inheritance. These kata will give you the C# syntax for inheritance.

Kata #1: The first project in this chapter expands the damage calculation app from Chapter 4. It does not use inheritance—the purpose is to help novice readers start to understand reasons why inheritance is valuable. It also introduces the syntax of the C# `switch` statement. This should be a quick exercise for you.

Before you do the rest of the kata, skim the sections: **Any place where you can use a base class, you can use one of its subclasses instead** and **Some members are only implemented in a subclass**. Then skim the section **A subclass can hide methods in the base class** and all of its subsections.

Kata #2: Go back and do the exercise that starts: Let's get some practice extending a base class. This should cover the basic syntax of inheritance in C#.

Kata #3: Do the project in the section **When a base class has a constructor, your subclass needs to call it.**

Kata #4: Do the exercise after the section **It's time to finish the job for Owen**. This is a two-part exercise: the first is a pencil-and-paper exercise to fill in the class diagram, and the second has you write the code to implement it.

If you've finished the first four kata without much trouble, you're ready to move on. However, **we strongly encourage you to build the Beehive Management System app** at the end of the chapter. It's actually a pretty fun project: it teaches some useful lessons about game dynamics, and it's very satisfying when you write just a few lines of code to change it from a turn-based game to a real-time game at the very end of the chapter.

If you get stuck on one of the kata, then it's most efficient for you to switch to working through the chapter linearly, starting after the last kata you finished.

Congratulations on taking the fast path through Head First C#. If you've made it through Chapter 6 following the Code Kata guide, you should be fully ready and ramped up to pick up at Chapter 7.

Index